

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

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

**Abstract**—This paper presents a compositional method with failure-preserving abstraction for scalable asynchronous design verification. It combines efficient state space reductions and novel interface refinement, and can reduce the complexity of state space dramatically while decreasing introduction of false failures. This allows much larger designs to be verified as demonstrated in the experimental results.

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

## I. INTRODUCTION

Compositional methods are essential to address state explosion in model checking. A compositional minimization method is described in [7] 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 overly restrictive, thus resulting in the real design errors escaped. Similar work is also described in [2].

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 a daunting task, and traditionally done by hand. Lately, some automated approaches [4], [6], [3], [1] 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.

In [12], [13], we developed methods that compositionally verify asynchronous designs based on Petri-net reduction. 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, these methods are not general, and limited to certain types of the Petri-nets.

In this paper, we present a compositional method for scalable asynchronous design verification. It uses failure-preserving reduction on state graphs to address state explosion. In this method, a design is modeled as a parallel composition of state graphs derived from the high-level descriptions of the modules in a design. To reduce complexity, state graphs are reduced by removing invisible state transitions on the interface of the modules before they are composed. At the end, a reduced state graph for the entire design is produced for verification. When building the state graph for each module, an environment needs to be found for the module. This method is sound as long as the environment of the module is an over-approximation of the real environment of the module when it is embedded in the design. The reductions on state graphs also preserve the soundness of this method by keeping the paths leading to the failures.

One problem with the above approach is that an over-approximate environment is needed when considering a module locally. This over-approximation can seriously blow up the size of the intermediate state graphs, thus limiting the capability of verification. Maybe more seriously, less accurate environment causes more occurrences of false negatives, and distinguishing the false negatives may incur high computational penalty. To address the above issues, we develop a fully automated interface refinement approach that dynamically refines the state graphs of design modules during composition. Combining the state space reductions and the interface refinement, our method can drastically reduce state graphs while decreasing occurrences of false failures. To the best of our knowledge, there is no other approach similar to ours in terms of module interface refinement in a compositional framework.

This paper is organized as follows. Section II introduces the modeling and verification of asynchronous designs. Section III describes our compositional verification method and the failure-preserving abstraction. Section IV describes our interface constraint derivation method. Section V demonstrates our method on several examples, and concludes the paper.

## II. PRELIMINARIES

Behavior of asynchronous designs is captured by state graphs (SGs). A SG  $G$  is a tuple  $(W, S \cup \{\pi\}, R, s^0)$  where

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

This research is supported by a grant from NISTP at University of South Florida, CAREER Award contract# CCF 0546492 and award contract# CCF 0551621 from the National Science Foundation.

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 has graduated with M.S. in 2007.

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

Copyright (c) 2008 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

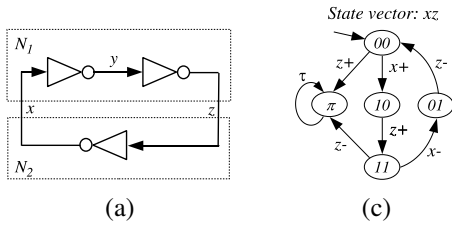


Fig. 1. (a) An inverter chain. (b) The SG for  $N_2$  with its maximal environment.

$W = I \cup O \cup X$  includes all inputs, outputs, and internal wires, and  $T$  is a set of transitions on wires in  $W$ .  $\pi$  is a special state indicating the failure state of  $G$ . After entering the failure state, the design is regarded as having a failure, and what happens afterward does not matter. This is defined by  $(\{\pi\} \times T \times \{\pi\})$ . 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 module  $N_2$  in the example in Fig.1(a) is shown in Fig.1(b).

A *trace* of a SG  $G$ ,  $\sigma = (t_0, t_1, \dots)$ , 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  $i \geq 0$ . The set of all possible valid traces of  $G$  starting from the initial state is denoted by  $\mathcal{P}(G)$ . 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. If a design is composed of multiple modules, the behavior of the entire design is the parallel composition of the SGs of the modules. Given two SGs  $G_1$  and  $G_2$ , the parallel composition is denoted by  $G = G_1 \parallel G_2$ .

In our method, the design correctness is defined as the absence of certain failures. 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 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 *persistency failure* happens if a transition firing causes some enabled transitions on outputs 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. All failure traces in  $G$  are denoted by  $\mathcal{F}(G)$ . Although this paper considers only these properties, verifying other properties in CTL/LTL, for example, can be readily implemented on SGs, and is not considered here.

Let  $\sigma = (t_0, \dots, t_i, t_{i+1}, \dots)$  be a trace of a SG  $G$ . 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(b), firing  $z+$  in the initial state causes a persistency 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  $G$ ,  $\mathcal{F}(G) \subseteq \mathcal{P}(G)$  holds. A design represented by  $G$  is correct if  $\mathcal{F}(G) = \emptyset$ . Given SGs  $G_1$  and  $G_2$  such that  $I_1 = I_2$ ,  $O_1 = O_2$ ,  $X_1 =$

$X_2$ ,  $G_1$  conforms to  $G_2$ , denoted as  $G_1 \preceq G_2$ , if

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

$\sigma_1 \equiv \sigma_2$  means that transition firings  $t_i^1$  in  $\sigma_1$  and  $t_i^2$  in  $\sigma_2$  are the same or one of them is  $\tau$  for all  $i \geq 0$ . Intuitively, if we verify  $G_2$  correct, we can readily conclude that any  $G_1$  such that  $G_1 \preceq G_2$  is also correct.

### III. VERIFICATION AND REDUCTIONS

In this section, we describe our compositional method of state graph construction for verification, and SG reductions to minimize the size of SGs during composition. In the following, we present our method on designs with two modules to simplify discussion. We assume that a design, modeled in some high-level language, is a parallel composition of the modules,  $N = N_1 \parallel N_2$ .

Let  $i = 1, 2$ . To build the SG for  $N$ , we need to find SGs for  $N_1$  and  $N_2$ . When finding the SG for each module  $N_i$ , it is essential to preserve all possible traces produced by  $N_i$  when it is embedded in the complete design. Let  $\mathcal{E}_i$  and  $\mathcal{E}'_i$  be the exact and an approximate environment of  $N_i$ , respectively. To satisfy the above requirement,  $\mathcal{E}'_i$  must ensure

$$G_i \preceq G'_i \quad (1)$$

where  $G_i$  and  $G'_i$  are the SGs of  $N_i \parallel \mathcal{E}$  and  $N_i \parallel \mathcal{E}'_i$ , respectively. With such  $\mathcal{E}'_i$ , the following property holds.

$$G \preceq (G'_1 \parallel G'_2).$$

where  $G$  is the SG of  $N$ .

Directly composing  $G'_i$  for verification defeats the purpose of compositional construction in that the interleavings of the invisible state transitions in  $G'_i$  may cause the explosion during composition. Our method uses two reductions to minimize SGs while preserving the paths leading to the failure state.

The *state transition abstraction* removes invisible state transitions of a SG, and merges the states connected by these state transitions. Let  $(s_i, \zeta, s_j)$  be an invisible state transition in  $G$ . This reduction removes  $(s_i, \zeta, s_j)$ , and merges  $s_i$  and  $s_j$  to form a new state  $s_{ij}$ . For each  $(s_h, t_h, s_i)$  and  $(s_j, t_j, s_k)$  in  $G$ , they are changed to  $(s_h, t_h, s_{ij})$  and  $(s_{ij}, t_j, s_k)$ . If  $s_j = \pi$ , then  $s_i$  is replaced with  $\pi$ , and all  $(s_i, t_i, s_l)$  are replaced with  $(\pi, \tau, \pi)$ . These cases are illustrated in Fig.2. *Autofailure reduction* is based on the observation that the failures of a design are caused by unacceptable input behavior driven by its environment, even though the failures may occur after some behavior on outputs or internal wires following the triggering input behavior. This is referred to as *autofailure manifestation* in [5]. However, in [5] it is only used to canonicalize trace structures for hierarchical verification. We adopt it to reduce SGs. Autofailure reduction works similarly to the state transition abstraction, but is only applied to  $(s_i, t_i, \pi)$  where  $t_i$  is on non-input wires. It does not remove any state transitions if a SG does not contain the failure state. Moreover, it does not introduce extra failure traces as state transition abstraction does.

Both techniques may remove some state transitions, and therefore cause some states to become unreachable. At the end of reduction, the unreachable states are removed too.

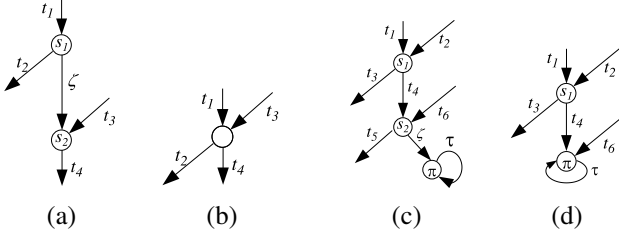


Fig. 2. State transition abstraction for two cases. (a) and (b) shows a case with no failure state, while (c) and (d) shows another case with failure.

#### IV. INTERFACE REFINEMENT

As discussed in the previous section, an over-approximate environment is needed for each module for sound verification. However, such environment may blow up the size of the intermediate SGs, and cause false failures. This may result in higher runtime penalty for removing the extra behavior and distinguishing false failures. In this section, we describe a method to refine the over-approximate interface of design modules. This method derives constraints automatically to restrict the interface behavior of modules during composition. It can be used along with the reduction techniques described in the last section to further contain the peak size of the intermediate SGs during composition.

##### A. Basic Concepts of Constraints

For each transition  $t \in T$ , we use a constraint to impose additional restrictions when it can occur. Given a SG  $G$ , a constraint of  $t \in T$  is a Boolean formula  $b$  over  $I \cup O$ . Let  $c^t$  be a constraint of  $t$ , and  $\text{eval}(s, b)$  be a predicate where  $s$  is a state of a SG  $G$  and  $b$  is a constraint.  $\text{eval}(s, b)$  holds if  $b$  is satisfied in  $s$ .

Given a SG  $G$ , a state transition  $(s_1, t, s'_1) \in R$  is valid with respect to  $c^t$  if  $\text{eval}(s, c^t)$  holds. In other words, a constraint  $c^t$  on a transition  $t$  corresponds to a set of state transitions defined as follows:

$$R_{c^t} = \{(s, t, s') \in R \mid \text{eval}(s, c^t) \text{ holds.}\} \quad (2)$$

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

$$(c_1^t \Rightarrow c_2^t) \Leftrightarrow (R_{c_1^t} \subseteq R_{c_2^t}) \quad (3)$$

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  $c_1^t$ .

Notice that reducing state transitions results in some traces to disappear, including failure traces. In other words, imposing constraints may eliminate some failures. Since imposing stronger constraints on a SG eliminates more state transitions and more traces consequently, the logical implication is reflected as the conformance relation between SGs when the constraints are applied to SGs. Let  $C_1, C_2 : T \rightarrow \{1, 0\}^{I \cup O}$  be two labeling functions that assign each transition of a SG

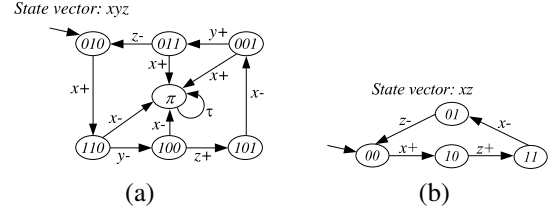


Fig. 3. (a) SG  $G_1$  for  $N_1$  in Fig. 1(a). (b) The SG  $G_2[C_1]$  for  $N_2$  after applying the constraints.

with different Boolean constraints. We use  $C_1 \Rightarrow C_2$  to denote  $\forall t \in T, C_1(t) \Rightarrow C_2(t)$ . Also, we define  $G' = G[C]$  such that

$$R' = \{(s, t, s') \in R \mid \text{eval}(s, C(t)) \text{ holds}\}$$

Given a SG  $G$ ,  $G[C_1] \preceq G[C_2]$  holds if, and only if,  $C_1 \Rightarrow C_2$ ,

##### B. Derivation and Application of Constraints

Given a design consisting of multiple modules, the interface refinement works as follows during composition. At the beginning, there may be no constraint for the first module. However, some output constraints may be derived from its SG. After applying the derived constraints to the inputs of the second module, the outputs of the second module may become more restricted, and the output constraints can be derived to refine the inputs of the following modules during composition. This approach preserves the soundness of our compositional verification method. In the worst case, the derived constraints are useless if they are weaker than the input behavior of a module defined by its environment.

To derive the constraint of a transition on an output wire from a SG  $G$ , all  $(s_i, w+, s'_i) \in R$  are found where  $w \in O$ . a Boolean conjunction on  $I \cup O$  is created from  $s_i$ . Then, the disjunction of all the conjunctions obtained for  $w+$  forms the constraint for  $w+$ . The constraint for  $w-$  is derived similarly. To apply constraints  $C$  to reduce  $G$ , we check each  $(s, t, s') \in R$ . If  $C(t)$  exists but  $\text{eval}(s, C(t))$  does not hold, it is removed from  $R$ .

Next, we give a simple example to illustrate how the refinement works. 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  $G_1$  for  $N_1$  as shown in Fig.3(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  $G_1$ . In this example,  $z+$  is enabled at  $\{100\}$ , and  $z-$  at  $\{011\}$ . The conjunctions 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 in 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  $G_2[C_1]$  is shown in Fig. 3(b) where all state transitions to the failure state are gone. This simple example illustrates how false failures can be removed by imposing the constraints derived.

#### V. EXPERIMENTAL RESULTS

The method described in this paper has been implemented and incorporated into an asynchronous design verification tool FLARE, and we have applied it to several examples. The goal

TABLE I  
EXPERIMENTAL RESULTS (TIME IS IN SECONDS, AND MEMORY IS IN MBs.)

Design	#M	W	Method 2				Method 3				Method 4			
			Time	Mem	S	# $\pi$	Time	Mem	S	$\pi$	Time	Mem	S	$\pi$
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
	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

of these experiments is to show that the capability of verification can be improved dramatically with our method. Also, the peak size of SGs during compositional can be significantly reduced as well as the occurrences of false failures.

FLARE is an explicit model checker for asynchronous circuit and system verification. It can perform flat and compositional verification. Its closest relative is ATACS [9]. Although ATACS also supports compositional verification and failure-directed abstraction, the abstraction is done using Petri-net (PN) reduction, which is not effective on a variant of PNs used in FLARE. In addition, FLARE 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 ours.

According to (1), an over-approximate environment is needed to simulate the actual one. In the experiments, we use the *maximal environment* for each module where the behavior of each input is completely independent to other inputs, and how each input changes is totally arbitrary. Let  $N$  represent a design, and  $\mathcal{E}$  and  $\mathcal{E}^{max}$  be some environment and the maximal environment of  $N$ . The property  $G \preceq G^{max}$  holds for the maximal environment where  $G$  and  $G^{max}$  are SGs of  $N||\mathcal{E}$  and  $N||\mathcal{E}^{max}$ , respectively. 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 [8], a tree arbiter which is a tree of multiple arbiter cells [5], and a distributed mutual exclusion element which is a ring of multiple DME cells in [5]. 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 [10]. This example is an unoptimized version of the actual circuit

used in RAPPID, and has higher complexity, which is more interesting for experiments. The last example is a pipeline controller for an asynchronous processor TITAC2 [11]. In the experiments, DME, arbiter, and FIFO examples are partitioned according to their natural structures. For the tag unit circuit and the pipeline controller, they are partitioned into three and five modules, respectively.

In the experiments, all examples are verified with four different methods: the method in [12], the same method but with the maximal environment for each module, the compositional method described in this paper without automated interface refinement, and the same method with the automated interface refinement. These methods are referred to as method 1, 2, 3, and 4, respectively. The results shown in the tables are obtained on a Linux server with a AMD Opteron dual-core CPU and 4 GB memory.

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

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 design descriptions before verification starts. The Petri-net reduction in [12] is not effective on the kind of PN used in our method. In the experiments, little or no reduction is achieved, and the verification for each example is like the flat one. Therefore, the table does not show the results with method 1. In method 2,  $\mathcal{E}^{max}$  is used for each module.

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

Verification finishes fast, however, the results are not good in that all modules in every example contain failures. This will incur higher runtime later on to distinguish these failures. In method 3, after generating the SG for each module with  $\mathcal{E}^{max}$ , the SGs are composed and form a global reduced SG. For DME and the TAGUNIT, 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 because the SGs are repetitively composed and reduced. In method 4, the automated interface refinement is added 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. The reason is that derived constraints can reduce the size of SGs before they are composed. Also, by reducing the sizes of SGs, the runtime may be shortened due to the less computation needed to compose these SGs. However, deriving and applying constraints may incur some overhead for aforementioned benefits as shown by the results of FIFO with 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. 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.

We also 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 dramatically compared to the results obtained with the fine partitioning. As well known, the state space grows exponentially as the size of designs grows. However, the maximal environment makes it worse since every input is completely unconstrained. These large SGs lead to the large increase in runtime for reduction, refinement, and composition. Although the maximal environment is used in our experiments because we assume 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.

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 a 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.

The experimental results show the effectiveness of our compositional method with interface refinement. The method is general in that it is not limited to any particular high level modeling formalism, and may be combined with other state space reduction approaches such as partial-order reduction to achieve better results.

## REFERENCES

- [1] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. Int. Conf. on Computer Aided Verification*. Springer-Verlag, 2005.
- [2] 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.
- [3] S. Chaki, E. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proc. Int. Conf. on Computer Aided Verification*. Springer-Verlag, 2005.
- [4] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification, 2003.
- [5] David Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.
- [6] 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.
- [7] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In *Computer Aided Verification*, pages 186–196, 1990.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.