Compositional Reachability Analysis for Efficient Modular Verification of Asynchronous Designs

Hao Zheng, *Member, IEEE*

***Abstract*—Compositional verification is essential to address state explosion in model checking. Traditionally, an over-approximate context is needed for each individual component in a system for sound verification. This may cause state explosion for the intermediate results as well as inefficiency for abstraction refinement. This paper presents an opposite approach, a compositional reachability method, that constructs the state space of each component from an under-approximate context gradually until a counter-example is found or a fixpoint in state space is reached. This method has an additional advantage in that counter-examples, if there are any, can be found much earlier, thus leading to faster verification. Furthermore, this modular verification framework does not require complex compositional reasoning rules. The experimental results indicate that this method is promising.**

***Index Terms*—formal verification, model checking, compositional verification, logic verification, circuit verification, abstraction refinement.**

I. Introduction

Although tremendous progresses have been made, model checking still faces the state-explosion problem [7]. Compositional approaches address this problem in a divide-and-conquer manner, and verify the individual components without considering the whole system. When checking each individual component, it is necessary to obtain its appropriate context where it is expected to operate correctly. The purpose of an appropriate context is used to remove from each component the behaviors that do not exist in the complete system.

In the existing compositional approaches [31], [18], [15], [4], [23], [14], [38], [39], an over-approximate context abstraction or assumption is used for each component to find its state space for sound verification. This context abstraction or assumption is needed in order to avoid any false positive results. Ideally, this context should be accurate to avoid excessive number of false counter-examples. However, manually finding such an context with higher accuracy is very difficult, if not impossible, and very time-consuming if the component interfaces are complex. Lately, some researchers proposed automated approaches [9], [3], [1] to generate context assumptions guided by local counter-examples.

Although impossible behavior due to abstract contexts may be reduced by abstraction refinement, some obvious shortcomings of these approaches can be pointed out as follows. By using abstraction, the state space of each individual component of a system needs to be blown up first, and then reduced gradually. The state space of a component generated with different contexts is illustrated in Fig. 1. The different circles characterize the component state space obtained using different contexts. Circle *U* refers to the state space resulting from an under-approximate context where the input behavior is more restricted than that of the exact context, while circle *O* refers to the state space resulting from an over-approximate context that includes extra input behavior. Between these two, circle *E* refers to the state space of a component resulting from the exact context as if it is embedded in the whole system. Obviously, the state space outside circle *E* is unreachable. In the existing modular verification approaches, the initial state space of each component is constructed as indicated by circle *O*. The goal of abstraction refinement is to shrink circle *O* to be as close to circle *E* as possible by reducing the unreachable state space. If the unreachable state space is very large initially, which can be the case in many situations, the process of this reduction can take a lot of time. In addition, the complexity of each component needs to be controlled during partitioning because the size of the single largest component dictates if the whole system can be verified. To accommodate this requirement, fine-grained partitioning is desired or required in the existing approaches. However, this may result in functionally unnatural partitioning that may cause some negative effects, such as more false counter-examples. In addition to the excessive peak size problem, verification is delayed in the above approach because refinement continues even though failures are found hoping that these failures may be removed later by refinement. This paper refers to this kind of state space construction as *state space contraction*.

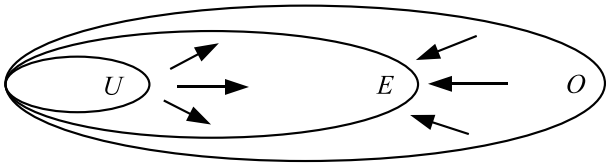

Fig. 1. U, E, and O represent the state space generated using under-approximate, exact, and over-approximate contexts, respectively.

To address the above problems, this paper presents a different approach, *state space expansion*. The basic idea is that the state space of a component is constructed using an under-approximate context where input behavior for each component is restricted. Starting from the initial state space denoted as U, components iteratively exchange information on their

Hao Zheng is with the CSE dept. of the Univ. of South Florida, Tampa, FL 33620. This material is based upon work supported by the National Science Foundation under Grant No. 0546492 and 0930510. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

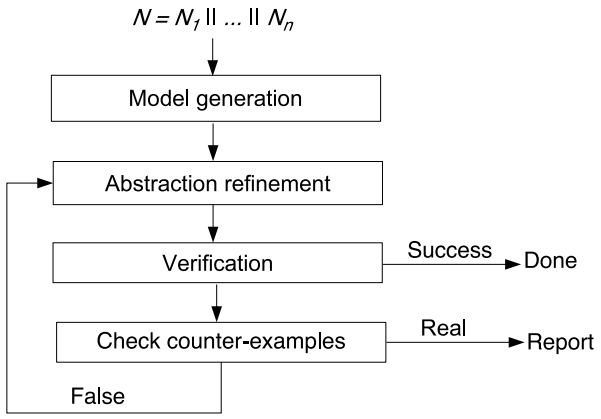

Fig. 2. A compositional verification flow.

interfaces to loosen their input behavior, and this allows them to gradually expand their state spaces. This process iterates until the state space of each component reaches a fixpoint or a counter-example is found. At the end, the component state space is still an abstraction of the concrete one if none of them contains counter-examples. However, the experimental results show that the resulting component state space is much closer to the concrete one leading to shorter verification time.

Fig. 2 shows a compositional verification flow. It takes as input a parallel composition of a set of components described in some high level modeling formalism. In general, an abstract model needs to be generated for each component for sound verification. However, an abstract model typically includes impossible behavior that may not occur when the component is embedded in the whole system. Abstraction refinement is then applied before verification to eliminate the impossible behavior as much as possible to avoid large number of false counter-examples, which can be very expensive to check if they are real. After verification, if all components are correct, the entire system is claimed to be correct. On the other hand, if any counter-example is found to be real, it is reported. Otherwise, the false counter-examples are used in the next iteration of model refinement and verification.

The method presented in this paper focuses on model generation step in Fig. 2. There are rich literatures on abstraction refinement, and discussing and comparing this work with all of them in detail is out of scope of this paper. Since other compositional reasoning approaches, to our best knowledge, either do not consider model generation or do not support the similar method to the one presented in this paper, this paper compares this work with one previous abstraction refinement method for modular verification to show its effectiveness.

This paper is organized as follows: section II gives an overview of the previous work on compositional verification and abstraction refinement. Section III gives a brief background review. The next three sections describe the new method proposed in this paper. Section IV describes our modular verification framework. Section V introduces the concepts of interface constraints, and describes the compositional reachability method using the constraints. Section VI presents experimental results on several large asynchronous designs, and the last section concludes the paper, and points out some future improvements.

II. Related Work

Compositional reasoning and *abstraction* are essential to verifying large systems. Compositional reasoning, broadly referring to compositional verification or compositional minimization, takes advantage of the given design hierarchy. A general compositional verification method is based on *assume-guarantee* style reasoning, and verifies global properties by verifying local properties of each component in a system [28], [22], [17], [18], [26]. It has been applied to the verification of timed circuits [35]. In a compositional verification framework, each component of a system is considered separately. During verification, assumptions about the environment with which the component interacts are made; then these assumptions need to be discharged later. Assumptions are typically generated by hand. If the component has complex interactions with its environment, it can be difficult to make accurate assumptions. Recently, there is some work on deriving assumptions automatically. In [21], an automated approach is described to generate the assumptions for compositional verification. This approach starts with a set of the weakest assumptions for a component, and iteratively refines these assumptions. Although the approach guarantees that the iteration terminates, it is not clear how efficient the approach would be in terms of iterations necessary to generate a set of assumptions to prove the properties. Also, this approach can only handle safety properties. In addition, global specification needs to be broken down to local properties defined on the interfaces of the components, which can be very difficult. Similar work is also described in [3], [1], [2], [32].

Abstraction produces the reduced model of a system by abstracting away certain details that are unnecessary when reasoning about the system [6], [11]. In [20], a hierarchical approach similar to that in [12] 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 [24], 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 [19], a software model checking method utilizing *lazy abstraction* is presented to improve performance by adding information during abstraction refinement only when necessary. This method and [8], [5] fall into a category called “Counter-Example Guided Abstraction Refinement” (CEGAR). In general, these methods build an abstract model where verification is applied. If an abstract counter-example is found, it is checked on the concrete model. If there is a corresponding concrete counter-example, then a true violation is found. Otherwise, the abstract counter-example is false due to information loss in the abstract model. And the abstract model is refined using the false abstract counter-example, and then verification repeats. The method presented in this paper is orthogonal to those CEGAR

approach in that this method builds abstract models from under-approximations, while the CEGAR approaches refines over-approximate models with the false counter-examples. In addition, this method is proposed by verifying individual components in a design, while the CEGAR approaches are applied to verifying the entire designs. On the other hand, CEGAR approaches can be used following this method to check if counter-examples in any component is real.

In [16], an approach is presented to construct a model from under-approximation similar to our method. It gradually adds more execution traces into an under-approximated model after it is checked correctly. However, that approach is for bounded model checking to find counter-examples while ours is for proving correctness. Again, that approach considers entire designs, while ours belongs to compositional verification.

Several tools have been developed for asynchronous circuit verification [13], [33], [29]. [13] uses a hierarchical verification approach similar to [12]. It checks safety as well as liveness properties. In [33], asynchronous circuits and the specification modeled in Petri-nets are represented by BDDs, and verification is performed by symbolic traversal. Compared to this method, both approaches are inherently non-compositional. In [38], [39], a modular approach is presented to verify timed asynchronous designs using abstraction methods 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. Although these methods are very effective for a particular kind of Petri-nets, they are not sufficient for the Petri net models used in our method.

III. Preliminaries

This section introduces basic notations and definitions for state graphs and their relative operators. It also presents how the correctness of safety properties is formulated and checked in this framework.

A. State Graphs

State graphs are used to model the behavior of concurrent systems. A state graph is a vertex-labeled and edge-labeled digraph. Vertices represent states, labeled with propositions that hold. Edges represent state transitions, labeled with actions whose executions cause the movement from one state to another. More formally, a state graph (SG) M is a 6-tuple $(P, \mathcal{A}, S, init, R, L)$ where

1) P is a finite set of atomic state propositions,
2) $\mathcal{A}$ is a finite set of actions,
3) S is a finite set of states,
4) $init \in S$ is the initial state,
5) $R \subseteq S \times \mathcal{A} \times S$ is the set of state transitions, and
6) $L : S \rightarrow 2^P$ is a state-labeling function.

In the above definition, S includes a special state π which denotes the *failure state* of a SG M, and represents violations of various safety properties. How a system behaves does not matter after it enters the failure state. Therefore, for every $a \in \mathcal{A}$, there is a $(\pi, a, \pi) \in R$. Each non-failure state is labeled with a non-empty set of propositions. For π, $L(\pi) = \emptyset$. Actions are used to model dynamic behavior of systems. For a SG, $\mathcal{A} = \mathcal{A}^I \cup \mathcal{A}^O \cup \mathcal{A}^X$ where $\mathcal{A}^I$ is the set of actions controlled by an environment of a system such that the system can only observe and react, $\mathcal{A}^O$ is the set of actions controlled by a system responding to its environment, and $\mathcal{A}^X$ is the set of actions controlled by a system internally. Each action a is associated with two sets of propositions, denoted as $\bullet a$ and $a\bullet$, respectively. For example, in asynchronous circuits, each wire w has two actions, $w+$ and $w-$, while $\bullet w+ = \{\neg w\}$ and $w+\bullet = \{w\}$, and $\bullet w- = \{w\}$ and $w-\bullet = \{\neg w\}$. Execution of an action a results in a new state by removing $\bullet a$ from and adding $a\bullet$ into the labellings of the existing state. Given $(s, a, s') \in R$,

$$L(s') = (L(s) - \bullet a) \cup a \bullet .$$

This paper uses $(s_1, a, s_2) \in R$ and $R(s_1, a, s_2)$ to denote that (s_1, a, s_2) is a state transition of a SG M. We assume that the state transition set R is total such that every state has some successor.

Fig.3(a) shows a simple asynchronous circuit as the running example to illustrate the ideas presented in this paper. The component labeled with "C" is a C-element whose output is high when both inputs are high, low when both inputs are low, or remains unchanged otherwise. This circuit is partitioned into three components, M_1, M_2 and M_3. Fig.3(b), (c) and (d) show the corresponding SGs for the components M_1, M_2, and M_3 where their inputs are set to be totally free, meaning they can change to high or low in any state. For clarity, only the labellings of the initial states are shown. In the figure, labellings of multiple actions on a single arc indicate multiple state transitions with the same start and end states but on different actions. For example, in Fig. 3(d), the arc from s_{14} to π denotes two different state transitions: $(s_{14}, x+, \pi)$ and $(s_{14}, y+, \pi)$. In particular, for M_3, its input actions $\mathcal{A}^I = \{x+, x-, y+, y-\}$, and its output actions $\mathcal{A}^O = \{z+, z-, u+, u-\}$.

A path of M is a sequence ρ of alternating states and actions of M, $\rho = (s_0, a_0, s_1, a_1, s_2, \cdots)$ such that $s_0 = init$, $s_i \in S$, $a_i \in \mathcal{A}$, and $\forall i \geq 0 : (s_i, a_i, s_{i+1}) \in R$. A state $s' \in S$ is *reachable from* a state $s \in S$ if there exists a path $\rho = (s_0, a_0, \ldots, s_n)$ such that $s = s_0$ and $s' = s_n$. A state s is reachable in M if s is reachable from the initial state $init$. The trace of path ρ, denoted by $\sigma(\rho)$, is the sequence of actions $(a_0, a_1, \cdots)$. Two traces $\sigma = (a_0, a_1, \cdots)$ and $\sigma' = (a'_0, a'_1, \cdots)$ are *equivalent*, denoted by $\sigma = \sigma'$, iff $\forall i \geq 0 : a_i = a'_i$ The set of all paths of M forms the language of M, denoted by $\mathcal{L}(M)$.

In some cases, not all actions of a component are used in a larger design. These unused actions are converted to invisible actions. Since only the interface behavior is of interest to verification, the information on states and state transitions related to invisible actions are abstracted away with a special action ζ. For ζ, $\bullet\zeta = \zeta\bullet = \emptyset$. The projection of a SG M by hiding a subset of $\mathcal{A}_1 \subset \mathcal{A}$ is defined as follows.

Definition 3.1: Let M be a SG, and $\mathcal{A}_1 \subseteq \mathcal{A}$. The projection of M onto $\mathcal{A}_1$, denoted by $M' = M[\mathcal{A}_1]$, is a SG such that

1) $P' = P - \bigcup_{\forall a \in \mathcal{A} - \mathcal{A}_1} (\bullet a \cup a\bullet)$.

2) $\mathcal{A}' = \mathcal{A}_1$.
3) $S' = \{s' \mid \forall s \in S \exists s' \in S' : L'(s') = L(s) \cap P'\}$.
4) $L'(init') = L(init) \cap P'$.
5) For each $(s, a, s') \in R$, there is a $(s, \zeta, s') \in R'$ if $a \notin \mathcal{A}'$, or $(s, a, s') \in R'$, otherwise.
6) $\forall s \in S : L'(s) = L(s) \cap P'$.

Similarly, given a trace $\sigma = (a_0, a_1, \ldots)$, its projection onto a subset of visible actions $\mathcal{A}' \subseteq \mathcal{A}$, denoted by $\sigma[\mathcal{A}']$, is obtained by removing from σ all the actions $a \notin \mathcal{A}'$. $\sigma[\mathcal{A}']$ is defined recursively as follows.

$$\sigma[\mathcal{A}'] \quad = \quad \begin{cases} \sigma' & \text{if } a_0 \notin \mathcal{A}' \text{ or } a_0 = \zeta, \\ (a_0) \circ \sigma' & \text{otherwise.} \end{cases}$$

where $\sigma' = (a_1, ...)[\mathcal{A}']$, and $\circ$ is the concatenation operator. Given two paths $\rho = (s_0, a_0, \ldots)$ and $\rho' = (s'_0, a'_0, \ldots)$ of M, ρ and ρ' are equivalent, denoted as $\rho \sim \rho'$, iff $\sigma(\rho) = \sigma(\rho')$.

The SG of a system is obtained by composing the component SGs. Parallel composition is defined as follows. This definition is very similar to the traditional definition in [2] except that more rules are included for cases involving π. Given $M_1 = (P_1, \mathcal{A}_1, S_1, R_1, init_1, L_1)$ and $M_2 = (P_2, \mathcal{A}_2, S_2, R_2, init_2, L_2)$, if $\mathcal{A}_1^O \cap \mathcal{A}_2^O = \emptyset$, the parallel composition of M_1 and M_2, $M_1 \| M_2 = (P, \mathcal{A}, S, R, init, L)$, is defined as follows.

1) $P = P_1 \cup P_2$,
2) $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$,
3) $S \subseteq S_1 \times S_2$ such that for each $(s_1, s_2) \in S$, the following conditions hold
   a) $L_1(s_1) \cap P_2 = L_2(s_2) \cap P_1$.
   b) $(s_1 = \pi \Rightarrow s_2 = \pi) \wedge (s_2 = \pi \Rightarrow s_1 = \pi)$.
4) $R \subseteq S \times \mathcal{A} \times S$ such that for each $((s_1, s_2), a, (s'_1, s'_2)) \in R$, if $s_1 \neq \pi$ and $s_2 \neq \pi$, then the following conditions hold.
   a) $(s'_1 = \pi \Rightarrow s'_2 = \pi) \wedge (s'_2 = \pi \Rightarrow s'_1 = \pi)$
   b) $\forall a \in \mathcal{A}_1 - \mathcal{A}_2 : R_1(s_1, a, s'_1) \wedge (s_2 = s'_2)$
   c) $\forall a \in \mathcal{A}_2 - \mathcal{A}_1 : R_2(s_2, a, s'_2) \wedge (s_1 = s'_1)$,
   d) $\forall a \in \mathcal{A}_1 \cap \mathcal{A}_2 : R_1(s_1, a, s'_1) \wedge R_2(s_2, a, s'_2)$.

   Otherwise, $s_1 = s'_1 = s_2 = s'_2 = \pi$ for every $a \in \mathcal{A}_1 \cup \mathcal{A}_2$.
5) $\forall (s_1, s_2) \in S : L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$.

In the above definition, the composite state is the failure state if either module state is the failure state. When several modules execute concurrently, they synchronize on the shared actions, and proceed independently on their invisible actions. If either individual SG makes a state transition to the failure state, there is a corresponding state transition to the failure state in the composite SG. The behavior of the composite SG captures the interaction between two individual SGs.

B. Correctness Definition

The failure state π is used to represent various safety violations that a system is not expected to produce. Liveness properties are not considered in this paper. A system is regarded as being correct if π is not reachable in its SG. A path is referred to as a *failure* if a SG contains the failure state π reachable via such path. The set of the failures in M is denoted as $\mathcal{F}(M)$ such that $\mathcal{F}(M) \subseteq \mathcal{L}(M)$ holds. A system is correct if $\mathcal{F}(M) = \emptyset$.

According to the definition of SGs, $(\pi, a, \pi) \in R$ for every $a \in \mathcal{A}$. Therefore, a failure $\rho_1 = (s_0, a_0, \cdots, s_i, a_i, \pi, \cdots)$ corresponds to a set of traces, denoted as $\Sigma(\rho_1)$. Given a failure $\rho = (s_0, a_0, \cdots, s_i, a_i, \pi, \cdots)$, the non-failure prefix of ρ is $(s_0, a_0, \cdots, s_i, a_i)$. If another trace ρ' has the same non-failure prefix of ρ, ρ' is also regarded as a failure. In such case, ρ and ρ' are called *failure equivalent*.

Definition 3.2: Given two path $\rho = (s_0, a_0, \ldots)$ and $\rho' = (s'_0, a'_0, \ldots)$, and $\exists j > 0 : s'_j = \pi$, ρ and ρ' are failure equivalent, denoted as $\rho \sim_F \rho'$, iff $\forall 0 \leq i \leq j.\ a_i = a'_i$.

With the equivalence between paths being defined, the abstraction relation between two SGs is defined as follows.

Definition 3.3: Given SGs M and M', M' is an abstraction of M, denoted as $M \preceq M'$, iff the following conditions hold:

1) $\mathcal{A} = \mathcal{A}'$.
2) For every path $\rho \in \mathcal{L}(M)$, there exists a path $\rho' \in \mathcal{L}(M')$ such that $\rho \sim \rho'$ or $\rho \sim_F \rho'$.

Intuitively, the abstraction relation defines that any path of M is also a path of M'. For any failure in M, there exists an equivalent failure in M'. In other words, the language accepted by M is also accepted by M'. Hence, $\mathcal{F}(M) = \emptyset$ if $\mathcal{F}(M') = \emptyset$. Therefore, the following property holds.

$$M \preceq M' \text{ and } \mathcal{F}(M') = \emptyset \quad \Rightarrow \quad \mathcal{F}(M) = \emptyset. \qquad (1)$$

Intuitively, the above property states that the concrete model M is correct if the abstract M' is correct.

IV. Modular Verification

In general, a system description is typically given in some high level modeling formalism. A finite state model is extracted from such a description for verification. This paper assumes that a system is described in a high level modeling formalism as $N = N_1 \| \ldots \| N_n$ where the system is the parallel composition of components $N_i (1 \leq i \leq n)$, and the parallel operator $\|$ is well defined for such a formalism. Flat verification approaches find the SG M for N where verification is applied. Due to state explosion, it is often impossible to verify N as a whole.

To deal with the high complexity, modular verification considers the components $N_i (1 \leq i \leq n)$ separately. First, each component N_i is composed with a context $\mathcal{E}_i$ defining actions in $\mathcal{A}_i^I$, and a typical reachability algorithm based on depth first search is applied to find the reachable state space M_i such that $M_i = \texttt{Reach}(N_i \| \mathcal{E}_i)$. Function `Reach` shown in Algorithm 1 is a simplified version of the one in [30].

When considering a component N_i, its context is the composition of all components in N except N_i. The SG of N_i embedded in such a context is referred to as M_i^C. It is straightforward to see that $\forall 0 \leq i \leq n : \mathcal{F}(M_i^C) = \emptyset \Rightarrow \mathcal{F}(M) = \emptyset$. However, the complexity of M_i^C may be as high as that of the SG of N. Therefore, it is necessary to find a M_i^A for component N_i such that $M_i^C \preceq M_i^A$ and the complexity of M_i^A should be much lower than that of M_i^C. By the definition of the abstraction relation and property (1), $\forall 0 \leq i \leq n : \mathcal{F}(M_i^A) = \emptyset \Rightarrow \mathcal{F}(M) = \emptyset$.

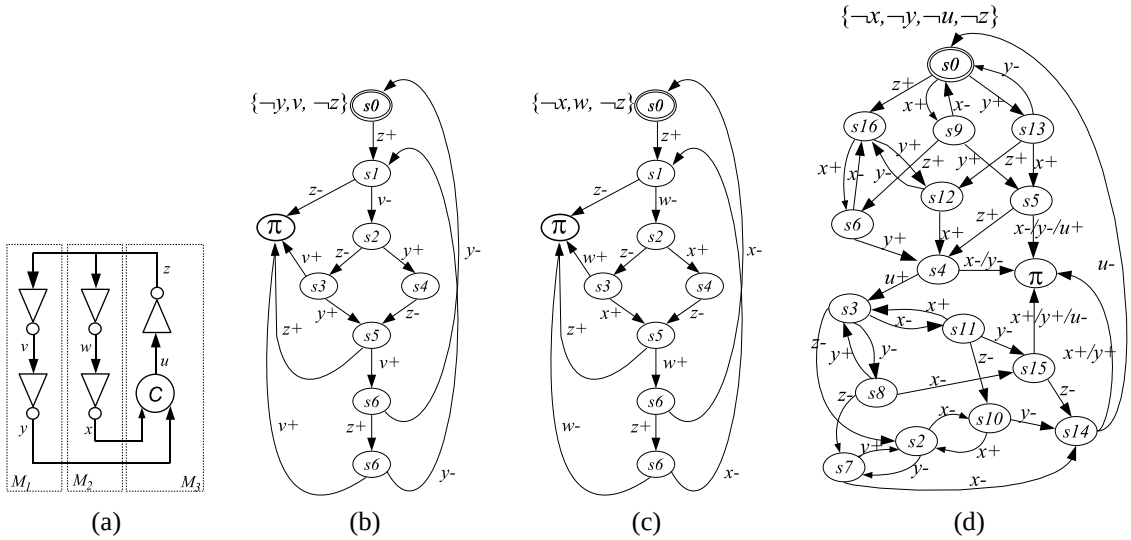

Fig. 3. (a) Block digram of a simple asynchronous circuit. (b) - (d) The SGs for module M_1, M_2, and M_3 where the inputs of the components are set to be completely free.

Algorithm 1: Reach $(N_i \| \mathcal{E}_i)$

```
1  S = ∅, R = ∅;
2  Select an action a from enable(init);
3  Push (init, enable(init) − {a}, a) onto stack;
4  S = {init};
5  while stack is not empty do
6      Execute action a, and find a new state s′;
7      R = R ∪ {(s, a, s′)};
8      if s′ ∈ S then
9          Select another action a from enable(s);
10     else
11         S = S ∪ {s′};
12         else if enable(s) on top of stack is empty then
13             Pop stack;
14         else
15             Select an action a from enable(s′);
16             Push (s′, enable(s′) − {a}, a);
```

Traditionally, an over-approximate context $\mathcal{E}_i'$ needs to be found for N_i such that the SG M_i' for N_i includes all essential behavior in N_i to avoid false positive results. However, M_i' may include extra behavior that is not supposed to happen in real operation, and may lead to false counter-examples. To reduce false counter-examples, abstraction refinement is used to identify and remove extra behavior from M_i', and refines it to be M_i^A such that $M_i^A \preceq M_i'$. There are several serious issues in this approach as pointed out in the introduction. In the remainder of this paper, a different method is presented that works in the opposite direction and find $M_i^{A'}$ for N_i from M_i'' such that $M_i'' \preceq M_i^{A'}$ by expanding it with more behavior, and $M_i^{A'} \preceq M_i^A$.

V. Compositional Reachability Analysis

This section first shows the basic concepts of constraints which can be used to exchange interface information among components. Then, it presents a compositional reachability analysis method where components coordinate with each other to expand their SGs gradually within under-approximate contexts.

A. Concepts of Constraints

An action a is enabled in a state s if there is a state s' such that $R(s, a, s')$ holds. Recall that each state is labeled with a set of propositions. An action is also regarded to be enabled in a state only when all the labeled propositions hold. Let $conj : S \rightarrow 2^P$ be a function that maps a non-failure state to a Boolean conjunction on P, and it is defined as follows.

$$conj(s) = \bigwedge L(s) \text{ for } s \neq \pi.$$

Specifically, function $conj(s)$ returns a Boolean conjunction over the propositions labeled in state s if it is not the failure state. An action is enabled in s if $conj(s)$ evaluates to true. This definition relates each enabled action with a Boolean formula. Therefore, we can characterize the enabling conditions of actions with Boolean formulas, denoted as *constraints*. Given a SG $M = (P, \mathcal{A}, S, init, R, L)$, let $f : 2^P \rightarrow \{false, true\}$ be a Boolean function defined over P. A constraint $\mathcal{C} = \{(a, f) | a \in \mathcal{A}\}$ of M is a set of pairs of actions of M and their assigned Boolean functions. The rest of the paper uses $\mathcal{C}(a)$ to denote the reference to f corresponding to a such that $(a, f) \in \mathcal{C}$. Additionally, if $\mathcal{C}_1$ and $\mathcal{C}_2$ are defined on the same set of $\mathcal{A}$, $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ is used to denote $\forall a \in \mathcal{A} : \mathcal{C}_1(a) \Rightarrow \mathcal{C}_2(a)$. Constraints can also be regarded as the characteristic function of the excitation region for an action as in [10]

This section assumes that constraints are defined for all actions of SGs to simplify presentation. When a constraint is imposed on actions, it may restrict how actions are enabled, therefore causing some state transitions to become invalid. A state transition $(s, a, s') \in R$ such that $s \neq \pi$ is valid with respect to a constraint $\mathcal{C}$ iff $conj(s) \Rightarrow \mathcal{C}(a)$ holds.

By the above definition, a constraint $\mathcal{C}$ of a SG M on an action a corresponds to a set of valid state transitions defined as follows.

$$R_{\mathcal{C}(a)} = \{(s, a, s') \in R \mid conj(s) \Rightarrow \mathcal{C}(a) \wedge s \neq \pi\}$$

It can be seen that $R_{\mathcal{C}(a)}$ becomes smaller if a stronger constraint $\mathcal{C}$ on a is imposed. Intuitively, a stronger constraint implies that the enabling conditions for actions become more restricted, and more state transitions may not be valid anymore. This observation is reflected in the following property.

$$\forall a \in \mathcal{A} : \big((\mathcal{C}_1(a) \Rightarrow \mathcal{C}_2(a)) \Leftrightarrow (R_{\mathcal{C}_1(a)} \subseteq R_{\mathcal{C}_2(a)})\big) \quad (2)$$

where $\mathcal{C}_1$ and $\mathcal{C}_2$ are two different constraints. This property states that the behavior in a SG regarding an action a is reduced when a stronger constraint is imposed on a, and vice versa. For example, $R_{\mathcal{C}_2(a)}$ includes all state transitions $(s, a, s') \in R$ in a SG if $\mathcal{C}_2(a) = true$, and $R_{\mathcal{C}_1(a)} \subseteq R_{\mathcal{C}_2(a)}$ for all other $\mathcal{C}_1(a)$. This example illustrates that $true$ is the weakest constraint for any action of a SG, and the SG remains the same with such a constraint.

As seen above, a constraint corresponds to a set of state transitions of a SG. Therefore, the constraint of a given SG can also be extracted. Let M be a SG such that $M = (P, \mathcal{A}, S, init, R, L)$. The constraint $\mathcal{C}$ extracted from M satisfies

$$\forall a \in \mathcal{A} : \left(\mathcal{C}(a) = \bigvee_{R(s,a,s') \wedge s \neq \pi} conj(s)\right)$$

where $\bigvee_{R(s,a,s') \wedge s \neq \pi} conj(s)$ is the disjunction of $conj(s)$ for all state transitions $(s, a, s') \in R$ such that s is not the failure state.

Let M_1 and M_2 be two SGs such that $M_1 \preceq M_2$, and $\mathcal{C}_1$ and $\mathcal{C}_2$ two constraints derived by $M_1\langle\mathcal{C}_1\rangle$ and $M_2\langle\mathcal{C}_2\rangle$, respectively. According to the definition of the abstraction relation, the behavior of M_1 is more restricted than that of M_2. This implies that the enabling condition of an action is more restricted in M_1 than in M_2. Consequently, this indicates that a stronger constraint may be derived from the refined SG as shown by the following property

$$(M_1 \preceq M_2) \quad \Rightarrow \quad (\mathcal{C}_1 \Rightarrow \mathcal{C}_2) \quad (3)$$

B. Model Generation

This section presents a compositional method that constructs the state space of each component using an under-approximate environment, and expands it to include all states and state transitions allowed by its neighboring components with constraints introduced in the last subsection. To simplify the presentation, N_i denotes a component where all its inputs are completely free.

The expansion-based method is described in Algorithm 2. Intuitively, constraints determine which state transitions are allowed in a state. As shown in the algorithm, the initial constraints for the inputs of each component are set to $false$, which indicates that the inputs remain stable, and no state transitions on inputs are allowed. With stable inputs, some component M_i may produce some state transitions on its outputs. Then, the output constraints of M_i are found by function `Extract`. Since the outputs of M_i may be the inputs of another component M_j, the output constraints from M_i become the input constraints for M_j. If the new input constraints are weaker than they were before, M_j may produce some more state transitions on its outputs, resulting in new input constraints for M_i. If the new constraints are weaker than before, new states may be found for some components. In other words, this process alternates between two phases: expanding the component state spaces and exchanging constraints. It iterates until the output constraints produced by each component do not change anymore, or failures are found in a component SG.

Algorithm 2: `Expand`$(N = N_1 \| \ldots \| N_n)$

```
1  Let A be all actions in N;
2  foreach a ∈ A do
3      C' = C' ∪ {(a, false)};
4  foreach 1 ≤ i ≤ n do
5      Let M_i be an empty SG for N_i;
6  C = ∅;
7  while C ≠ C' do
8      C = C';
9      for 1 ≤ i ≤ n do
10         C_i = findConstraint(C, M_i) ;
11         M_i = Reach(N_i, M_i, C_i) ;
12         if F(M_i) ≠ ∅ then
13             return F(M_i);
14         C_i^O = Extract(M_i);
15         C' = C' ∪ C_i^O;
```

Next, the functions used in Algorithm 2 are explained with more detail. Function `findConstraint` takes the union of the output constraints from all components, finds a subset of these constraints for the input actions $\mathcal{A}_i^I$ of component M_i, and project these constraints onto the interface of M_i. More specifically, `findConstraint`$(\mathcal{C}, M_i)$ returns $\mathcal{C}_i^I$ such that

$$\mathcal{C}_i^I = \{(a, f') \mid \forall a \in \mathcal{A}_i^I : f' = \mathcal{C}(a)[P_i]\}$$

where $\mathcal{C}(a)[P_i]$ denotes the projection of $\mathcal{C}(a)$ onto P_i.

Function `Reach`$(N, M, \mathcal{C})$ used in Algorithm 2 is modified from `Reach` in Algorithm 1, and it is shown in Algorithm 3 where $\mathcal{C}$ is a constraint defined for input actions in N. This constraint specifies the conditions that input actions need to satisfy to become enabled. Additionally, partial SGs M_i generated during the expansion process are also used by this function to avoid redundant work, and only new states and state transitions found under constraint $\mathcal{C}$ are added into M. In Algorithm 3, new actions enabled in a state s under constraints $\mathcal{C}$ are defined by two functions $enable'(N, s, \mathcal{C})$ and $enable(N, s, \mathcal{C})$. $enable'(N, s, \mathcal{C})$ is used only once at the beginning every time when `Reach`$(N, M, \mathcal{C})$ is called, and it only includes input actions actions enabled in state s under $\mathcal{C}$. It is defined as follows.

$$enable'(N, s, \mathcal{C}) = \{a \mid a \in \mathcal{A}^I \wedge conj(s) \models \mathcal{C}(a)\}$$

Algorithm 3: Reach $(N, M, \mathcal{C})$

```
1  foreach s ∈ S do
2      E = enable'(N, s, C);
3      Select an action a from E;
4      Push (s, E − {a}, a) onto stack;
5  while stack is not empty do
6      Execute action a, and find a new state s';
7      R = R ∪ {(s, a, s')};
8      if s' ∈ S then
9          if E on top of stack is empty then
10             Pop stack;
11         else
12             Select another action a, and remove a from
               E;
13     else
14         S = S ∪ {s'};
15         E = enable(N, s', C);
16         Select an action a from E;
17         Push (s', E − {a}, a);
```

Algorithm 4: Extract(M)

```
1  P = ∅;
2  foreach a ∈ A^I ∪ A^O do
3      P = P ∪ •a;
4      P = P ∪ a•;
5  foreach (s, a, s') ∈ R_i and s ≠ π and a ∈ A_i^O do
6      Let c be conj(s) projected onto P;
7      Replace (a, f) ∈ C_i with (a, f ∨ c);
8  return C_i;
```

The reason why this function is necessary at the beginning of Reach$(N, M, \mathcal{C})$ is to avoid redundant work. Notice that no actions in $\mathcal{A}^O \cup \mathcal{A}^X$ in any state in M can be enabled under the previous constraints. When Reach$(N, M, \mathcal{C})$ is called, the new constraint may be weaker, and only new input actions may become enabled under the new constraint. If non-input actions are also considered, the enabled action set may include a large number of non-input actions that have been considered previously, and time would be spent without finding new states or state transitions.

On the other hand, $enable(N, s, \mathcal{C})$ is used in the rest of the algorithm, and it is defined as follows.

$$enable(N, s, \mathcal{C}) = enable(N, s) \cup enable'(N, s, \mathcal{C})$$

where function $enable(N, s)$ returns actions in $\mathcal{A}^O \cup \mathcal{A}^X$ enabled in s. Obviously, $enable'(N, s, \mathcal{C}) \subseteq enable(N, s, \mathcal{C})$. This function is defined as such because new states may be found by executing the input actions in $enable'(N, s, \mathcal{C})$, and actions include input and non-input actions may be enabled in these new states. From the above description, input actions are enabled subject to constraint $\mathcal{C}$ while non-input actions are enabled subject to the behavioral description of N.

Function Extract derives constraints for outputs of a component from its SG. Each component updates its behavior on its output actions, while its input actions are defined by the environment. Therefore, given a SG of a component, only the constraint for non-input actions are extracted. However, the behavior on internal action $\mathcal{A}^X$ of a SG is invisible to other SGs, and the constraints for the internal actions are meaningless to other modules. Therefore, the constraints are extracted only for the output actions as shown in Algorithm 4.

Theorem 5.1 below proves the soundness of the compositional reachability method described above. It shows that each component SG generated at the end of expansion is an abstraction of the SG of the entire system projected to the component. To prove the theorem, we show that every path of the complete SG projected to a component has a corresponding path in the component SG. To prove the above claim, we show that every action enabled in a path of the complete SG projected to the component is also enabled in the corresponding path of that component SG.

Theorem 5.1: Let M be the SG for $N_1 \| \ldots \| N_n$. Also let M_i be component SGs corresponding to N_i for all $1 \leq i \leq n$ after calling Expand$(N_1 \| \ldots \| N_n)$. The following property holds.

$$\forall 1 \leq i \leq n : M[\mathcal{A}_i] \preceq M_i.$$

Proof: To prove $M[\mathcal{A}_i] \preceq M_i$, it is necessary to show that for every $\rho \in \mathcal{L}(M[\mathcal{A}_i])$, there exists $\rho_i \in \mathcal{L}(M_i)$ such that $\rho \sim \rho_i$ or $\rho \sim_F \rho_i$.

Let q, s, and p denote states in $M[\mathcal{A}_i]$, M_i, M_j, respectively. Also let $\rho = (q_0, a_0, \ldots) \in \mathcal{L}(M[\mathcal{A}_i])$ where $L(q_0) = L(init) \cap P_i$.

First, we partition each path in $M[\mathcal{A}_i]$ according to actions in $\mathcal{A}_i$. Notice that for every (q_i, a_i, q_{i+1}) on ρ, $L(q_i) = L(q_{i+1})$ if $a_i \notin \mathcal{A}_i$. Therefore, ρ can be partitioned by $a'_0, a'_1, \ldots \in \mathcal{A}_i$ into $\varphi_0, \varphi_1, \ldots$ such that

$$\rho = \varphi_0 \circ a'_0 \circ \varphi_1 \circ a'_1 \circ \ldots$$

where $\circ$ denotes the concatenation operator, and $a'_l = a_k$ for some $a_k \in \mathcal{A}_i$ on ρ, and

$$\varphi_l = (q_{l,0}, \zeta, q_{l,1}, \zeta, \ldots, q_{l,m}).$$

Where $L(q_{l,h}) = L(q_{l,j})$ for $0 \leq h, j \leq m$. In particular, for all $q_{0,h}$ in φ_0, $L(q_{0,h}) = L_i(init_i) = L(M) \cap P_i$. Note that φ_l may be a single state instead of a path segment.

Next, we show that every action in $\mathcal{A}_i$ enabled in ρ is also enabled in a path in M_i. Consider action a'_0 first. It is enabled in $M[\mathcal{A}_i]$ after φ_0. To prove that a'_0 is also enabled in $init_i$, two cases need to be handled.

- Case 1: $a'_0 \in \mathcal{A}_i^O$. This means that action a'_0 is controlled by M_i. As shown in Algorithm 3, actions in $\mathcal{A}_i^O$ are enabled independent of any external constraints. Therefore, a'_0 is enabled in $init_i$.
- Case 2: $a'_0 \in \mathcal{A}_i^I$. This means that action a'_0 is controlled by another SG M_j. Similar to Case 1, a'_0 is enabled in $init_j$ of M_j. Next, we need to show that a'_0 is also enabled in $init_i$. According to Algorithm 2, a constraint c for a'_0 is extracted from $init_j$, which is projected to $P_i \cap P_j$ for it to be applied to M_i. Since the entire design has a single initial state, $L_i(init_i) \cap P_j = L_j(init_j) \cap P_i$, indicating that the

labellings of the initial states of M_i and M_j agree on the shared propositions. Therefore, the projected constraint c of a'_0 extracted from $init_j$ holds in $init_i$, and consequently it implies that a'_0 is also enabled in $init_i$.

From both cases, it can be concluded that there exist $(init_i, a'_0, s_1)$ in M_i corresponding to $\varphi_0 \circ a'_0 \circ \varphi_1$ such that $L_i(s_1) = L(q) \cap P_i$ and for all q in φ_1. Since a'_0 is on the interface between M_i and M_j, there also exists $(init_j, a'_0, p_1)$ in M_j according to the definition of the SG parallel composition, and $L_i(s_1) \cap P_j = L_j(p_1) \cap P_i$. After executing a'_0, $L(q_{1,h}) = L_i(s_1)$ for all states $q_{1,h}$ in φ_1.

Similarly, the above argument can be applied to a'_1 from φ_1 in ρ and from s_1 in M_i, and the same conclusion can be drawn. By induction, it can be concluded that there exists (s_i, a'_i, s_{i+1}) in M_i corresponding to $\varphi_i \circ a'_i \circ \varphi_{i+1}$. This is equivalent to that there exists $\rho_i \in \mathcal{L}(M_i)$ for every $\rho \in \mathcal{L}(M[\mathcal{A}_i])$. Therefore, $M[\mathcal{A}_i] \preceq M_i$. ∎

On the other hand, this method is incomplete in that false counter-examples may exist in some component SGs. This is due to the limitation of the constraints, which do not give any information about the internal states of a component. This may cause extra input behavior introduced when the constraints are applied to expand component SGs. Therefore, refinement is needed after the model generation step to further remove extra behavior. This subject is out of scope of this paper.

C. Example

This section illustrates the idea of the compositional reachability method using the example shown in Fig.3. Initially, all signals are low. For SGs M_1 and M_2, no actions are enabled because none of these actions satisfies the initial constraint. For M_3, the initial constraint allows action $z+$ to be enabled. After executing this action, a new state is reached. The SGs after the first iteration is shown in Fig. 4(a).

Now, signal z has changed, and new constraint can be derived where z is high. This allows input action $z+$ in M_i and M_2 to be enabled. After executing this action, the invisible actions $v-$ and $w-$ also become enabled. Executing these actions lead to new states in M_1 and M_2. In these new states, output actions $y+$ and $x+$ become enabled. Again, executing these output actions result in new states where constraints for actions on x and y can be derived for M_3. Meanwhile, M_3 remains stable in this iteration since the constraints from M_1 and M_2 from the last iteration have not changed. The SGs after the second iteration is shown in Fig. 4(b).

Since the new constraints for actions on x and y allow actions $x+$ and $y+$ in M_3 to be enabled, M_3 is expanded with new states and state transitions after executing these actions. The updated M_3 is shown in Fig. 4(c) where M_1 and M_2 remain unchanged. Repeating this process eventually results in SGs for component M_1, M_2, and M_3 as shown in Fig. 4(d). Compared to SGs shown in Fig. 3(b)-(d) where they are constructed with over-approximate contexts, the SGs obtained by the compositional reachability method do not contain unreachable states and transitions including ones causing failures. The numbers of states and state transitions

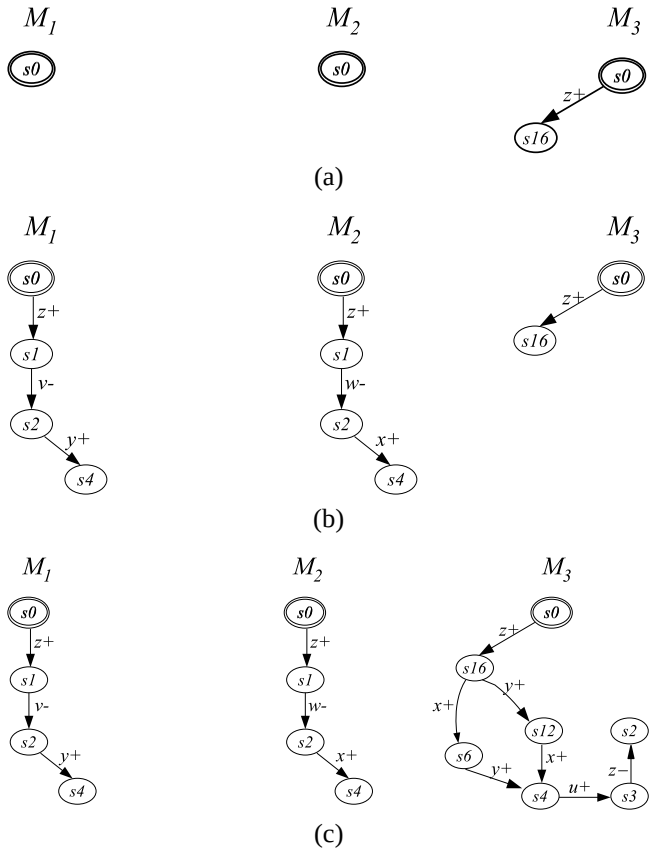

Fig. 4. (a)-(d) Snapshots of partial SGs generated during compositional reachability analysis.

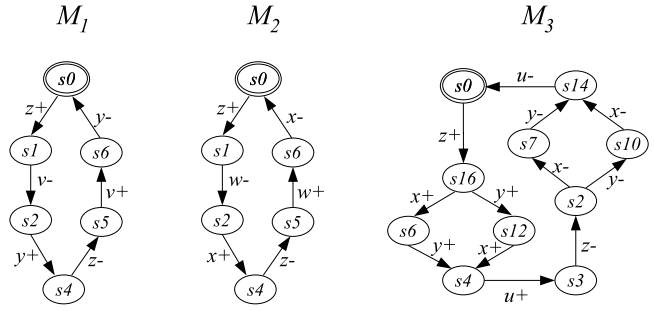

Fig. 5. Final SGs after compositional reachability analysis.

(states/transitions) in the SGs in Fig. 3(b)-(d) are 9/14, 9/14, and 17/38, respectively, while the numbers of states and state transitions in the SGs in Fig. 5 are 6/6, 6/6, and 10/12, respectively. For larger examples, the savings may be more significant as shown by the experimental results.

In [36], an abstraction refinement approach is presented where constraints are used to reduce state transitions in a component not allowed by its neighbors. In the above example, final SGs by the abstraction refinement and this method are the same. However, the next example in Fig. 6 shows that the abstraction refinement is incapable of reducing the extra state transitions introduced by over-approximate contexts, which may conceal the actual enabling conditions of actions.

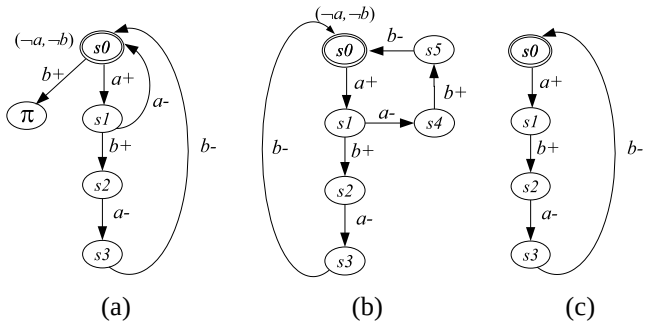

Fig. 6. SGs of two components communicating via a and b. (a) M_1 where a is output and b is input. (b) M_2 where a is input and b is output. (c) The SG of $M_1 \| M_2$.

In Fig.6, M_1 in Fig. 6(a) has input actions $b+$ and $b-$, and output actions $a+$ and $a-$, while M_2 in Fig. 6(b) has input actions $a+$ and $a-$, and output actions $b+$ and $b-$, respectively. Fig 6(c) shows the SG of $M_1 \| M_2$. According to $M_1 \| M_2$, transitions $(s0, b-, \pi)$ and $(s1, a-, s0)$ in M_1, and $(s1, a-, s4)$, $(s4, b+, s5)$, and $(s5, b-, s0)$ in M_2 are extra since they do not exist in $M_1 \| M_2$. The constraints for $a+$ and $a-$ from M_1 are $\mathcal{C}(a+) = \neg a \wedge \neg b$ and $\mathcal{C}(a-) = a$, and constraints for $b+$ and $b-$ from M_2 are $\mathcal{C}(b+) = \neg b$ and $\mathcal{C}(b-) = \neg a \wedge b$, respectively. Using these constraints cannot remove any of these extra state transitions. However, using the state space expansion method described in the paper avoids generating these extra state transitions in the first place. This example demonstrates an important advantage of the expansion-based method over abstraction refinement.

VI. Experimental Results

A prototype of the compositional reachability method described in this paper is incorporated into an asynchronous system verification tool `Plato`, an explicit model checker, which can perform non-compositional and compositional verification. The asynchronous designs are described using an variant of Petri-nets (PN) which are augmented with Boolean guards for the PN transitions [27]. The tool also supports abstraction refinement for SGs constructed using over-approximate environment. Experiments have been performed on several large asynchronous circuit designs, and results are compared with those obtained by using abstraction refinement.

A. Examples

In our method, asynchronous systems are specified in a high level description. To verify a design, all components in that high level description are converted to SGs first. The first three designs are a self-timed FIFO [25], a tree arbiter of multiple cells [12], and a distributed mutual exclusion element consisting of a ring of DME cells [12]. Despite all these designs having regular structures to be scaled easily, 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 [34]. This example is an unoptimized version of the actual circuit used in RAPPID with higher complexity, which is more interesting for experimenting our methods. The last example is a pipeline controller for an asynchronous processor TITAC2 [37]. All these five examples are failure free, and all of them are too large for the non-compositional approaches.

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

B. Results and Analysis

The experimental results are shown in Table I. To show the effectiveness of this compositional reachability method, it is compared with an abstraction refinement method as described in [36]. This abstraction refine method also utilizes the constraints. However, the initial component state graphs are constructed using over-approximate contexts, and constraints are derived and applied to reduce states and transitions in each component SG not allowed by its neighboring components iteratively. The results obtained by state space contraction with abstraction are shown in columns in Table I under *Over-Approximate*, while the results by state space expansion described in this paper are shown in columns in the table under *Under-Approximate*.

All experiments are performed on a Linux workstation with a Intel Pentium Dual-Core CPU and 1 GB memory. In the table, column $\#Cells$ shows the number of components in a design after partitioning, column $|\mathcal{A}|$ shows the number of actions in a design. Column Mem and $Time$ are the maximal memory and the total time taken for verifying each design, respectively. The last column $\#\pi$ shows the number of components containing failures at the end of each verification run. The memory is in MBs and the time is in seconds.

First thing to notice from the table is that the memory and runtime usage required by the method based on state space expansion are much less than what are required by the state space contraction-based one for all designs. The savings are results of not generating unreachable state space for each components in the first place and therefore avoiding time for abstraction refinement. Next, all designs except PC are free of failures after using method *Under-Approximate*. Even for PC, the number of components containing failures is less by using the state space expansion-based method. It is more interesting when ARB is examined more closely. Although the results in the table shows all components in ARB 15, 31, and 63 free of failures under *Over-Approximate*, they are obtained by composing several smaller components together to form larger ones so that more state space reduction can be applied to lead to stronger constraints and consequently stronger refinement. Otherwise, more than half of all components in ARB 15, 31 and 63 would contain failures. This indicates that more accurate constraints that can be derived because a lot of unreachable state space is not generated in the first place in the state space expansion-based method,

TABLE I
EXPERIMENTAL RESULTS AND COMPARISON WITH THE CONTRACTION-BASED METHOD.

Design	# Cells	$\|\mathcal{A}\|$	*Over-Approximate*			*Under-Approximate*		
			Mem	Time	$\#\pi$	Mem	Time	$\#\pi$
FIFO	100	804	30	18	0	16	15	0
	200	1604	80	41	0	36	34	0
	400	3204	237	102	0	74	78	0
	600	4804	471	184	0	124	126	0
	800	6404	781	290	0	183	177	0
FIFO$*$	800	6404	772	273	1	28	31	1
DME	20	440	35	43	0	6	10	0
	50	1100	88	113	0	18	30	0
	100	2200	191	249	0	41	83	0
	200	4400	446	600	0	92	199	0
	300	6600	771	1044	0	147	383	0
DME$*$	300	6600	748	990	1	29	41	1
ARB	15	244	7	6	0	2	2.1	0
	31	500	33	47	0	6	5	0
	63	1012	262	988	0	16	12	0
ARB$*$	63	1012	255	912	1	11	9	1
TU	3	96	117	103	0	12	7.7	0
PC	10	100	23	47	4	1	1.5	1

$*$ — one of cells is injected with failures.

therefore these constraints characterize the enabling conditions of actions more precisely. On the other hand, in the state space contraction-based method, constraints representing the true enabling conditions of actions may be concealed by the unreachable states caused by the over-approximate contexts as shown by the second example in the previous section. This consequently leads to the unreachable state space not being able to be identified and removed. Therefore, state space expansion brings double advantages of reducing runtime and memory usage as well as introducing less number of false failures, which contributes to further savings of avoiding the expensive counter-examples confirmation step.

For designs followed with $*$ in Table I, one of cells is intentionally injected with failures. As shown by the results in the table, this method is much more efficient compared with method *Over-Approximate*. As explained before, this method stops right away when a failure is found in any component in a design while method *Over-Approximate* has to keep refining component SGs containing failures in hope that eventually these failures may be removed after the extra behavior is refined away, which takes more time. Therefore, method *Under-Approximate* is also more efficient for designs containing failures.

TABLE II
EXPERIMENTAL RESULTS AND COMPARISON WITH THE ATACS.

Design	# Cells	ATACS			*Under-Approximate*		
		Mem	Time	$\#\pi$	Mem	Time	$\#\pi$
FIFO-s	800	75	1783	0	164	255	0
DME-s	300	61	678	0	123	320	0
ARB-s	63	11	104	0	15	12	0

The same experiments are also performed using ATACS [29], the closest relative to our method. ATACS supports a similar modular verification framework as in this paper. However, modular verification is made possible in ATACS by Petri-net reduction based abstraction, and the Petri-net reductions are effective only on a certain type of Petri-nets, and it does not support abstraction refinement described in this paper. Therefore, a lot of false counter-examples may be produced if the context for a component derived by these reductions is not accurate. Since these Petri-net reductions are not effective on the specification formalism used in this method, little or no reduction is achieved when deriving context for each component, and verification for each component is like verifying the entire design. 1 GB memory is exhausted when verifying the first component in all experiments, therefore the runtime and memory usage results obtained by using ATACS on these examples are not shown in Table I.

To compare the work in this paper and ATACS, the behavioral descriptions of FIFO, DME, and ARB are modeled in Petri-nets acceptable for ATACS and used for experiments. The results are shown in Table II. Notice that these new descriptions do not model the actual circuits, instead they describe the circuits' behavioral specification. It can be seen from the table that the memory usage by ATACS is far less than that by this method while the runtime is much longer. This is because ATACS produces a very small Petri-net description for each component, and the resulting SG is small too. Moreover, only the SG for a single component is generated at a time. However, reduction needs to be performed on the whole design descriptions for each component, therefore taking more time. Even though ATACS shows some advantage over this method, the effectiveness of ATACS depends on if the design descriptions are appropriate for the reductions available in ATACS. These experiments also show that this method is more general in terms of formalisms describing designs.

Table III shows the comparison of the largest SGs encountered during the verification process using method *Over-Approximate* and *Under-Approximate*. The largest SGs for the components produced by method *Over-Approximate* occur at the beginning of the verification process when the SGs for

TABLE III
LARGEST SGS FOUND BY *Over-Approximate* AND *Under-Approximate*.

Design		*Over-Approximate*		*Under-Approximate*	
	Cells	$\|S\|$	$\|R\|$	$\|S\|$	$\|R\|$
FIFO	All	57	188	20	28
DME	All	329	1100	152	272
ARB	All	673	3760	52	84
TU	Cell 1	181	474	101	149
	Cell 2	17481	108376	9410	43635
	Cell 3	1081	3624	236	447

some components are produced with maximal environment. For all examples, the SGs for all components in each example are refined to the ones whose numbers of states and transitions are the same as the corresponding entries under *Under-Approximate*. However, these entries show the size of the largest SGs produced by method *Under-Approximate* at the fixpoint of reachability analysis. These SGs also happen to be the SGs produced from the corresponding components embedded within the exact contexts. These results demonstrate the tightness of the SGs generated by this method.

The next set of experiments tries to show the impact of design partitioning on the performance of these two methods. In these experiments, FIFO with 100 cells (FIFO-100), ARB with 31 cells (ARB-31), and DME with 20 cells (DME-20) are selected. For FIFO-100, five cells are grouped into a single component while the other components still have a single cell. For ARB-31 and DME-20, one component contains two cells while the others have a single cell. The results from using both methods are shown in Table IV. Comparing the entries in this table and the corresponding ones in Table I shows that design partitioning impacts much more dramatically on method *Over-Approximate* where both memory usage and runtime increase significantly. While the memory usage and runtime increase too in method *Under-Approximate*, the magnitude of increase is much smaller and proportional to the size of SGs of the largest partition in the designs. Again, the largest SGs found in method *Over-Approximate* are much larger than those found in method *Under-Approximate*, which are the final results after refinement is done in method *Over-Approximate*. In this compositional method, the complexity of the largest partition determines if the whole design can be verified. Therefore, it is desirable that all partitions are created with about similar complexities, and smaller partitions are better in terms of higher verification performance and less memory requirement.

Failures found at the end of verification can be determined using the approach described in [39]. However, in the above experiments, such approach is not used to show the capability of this method to avoid the false counter-examples in the first place. Since the component SGs constructed using this method contain far less unreachable state space leading to less failures to consider, time needed to determine the truth of the failures in the state space expansion-based method can be much less than that in the state space contraction-based method.

VII. CONCLUSIONS

This paper describes a state space expansion method to construct component state space compositionally. It uses the constraints extracted from a component's neighbors to determine the enabling conditions of its inputs, and constructs the component state space by gradually loosening the enabling conditions for inputs allowed by its neighbors. Initial experiments show that this method is very effective to avoid generating large portion of unreachable state space in the first place, therefore leading to big savings in memory and runtime usage.

The method presented in this paper is based on an explicit representation. Such an explicit representation is more flexible for asynchronous designs, and can be easier to be adopted for hybrid system verification with appearance of continuous variables. Additionally, the performance of explicit model checking is more predictable. However, since implicit representations such as BDDs are widely used in many application domains, it would be interesting to investigate if the presented method can be modified for these implicit representations. Moreover, it is also necessary to find a better representation of constraints to characterize the enabling conditions of actions more accurately, therefore making the constructed state space to be as close the exact one as possible.

REFERENCES

[1] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. Int. Conf. on Computer Aided Verification*, volume 3576 of *LNCS*, pages 548 – 562. Springer-Verlag, 2005.

[2] M. Bobaru, C. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Proc. Int. Conf. on Computer Aided Verification*. LNCS, 2008.

[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*, LNCS, pages 534 – 547. Springer-Verlag, 2005.

[4] S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, 1996.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5):752–794, 2003.

[6] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Mass., 2001.

[8] E. Clarke, A. Gupta, J. Kukula, and O. Shrichman. Sat based abstraction-refinement using ilp and machine learning techniques. In *Proc. International Workshop on Computer Aided Verification*, pages 265–279, 2002.

[9] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, pages 331–346. Springer-Verlag, 2003.

[10] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer Series in Advanced Microelectronics, 2002.

[11] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[12] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.

[13] J. Ebergen and R. Berks. VERDICT: A verifier for asynchronous circuits, Aug. 1995.

[14] D. Giannakopoulou, C. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of ASE'02*, pages 3–12. IEEE Computer Society, 2002.

[15] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proc. Int. Conf. on Computer Aided Verification*, pages 186–196, 1990.

TABLE IV
IMPACT OF PARTITIONING ON *Over-Approximate* AND *Under-Approximate*.

Design	*Over-Approximate*				*Under-Approximate*			
	Mem	Time	$\|S\|$	$\|R\|$	Mem	Time	$\|S\|$	$\|R\|$
FIFO-100	115	183	48505	256348	53	46	20276	79644
DME-20	62	92	23671	112768	11	17	8768	27152
ARB-31	57	86	9837	31074	6	6	444	1054

[16] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. *SIGPLAN Not.*, 40(1):122–131, 2005.

[17] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[18] T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: methodology and case studies. In *Proc. Int. Conf. on Computer Aided Verification*, pages 440–451. Springer, 1998.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *The 29th Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[20] H. E. Jensen, K. G. Larsen, and A. Skou. Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT*, pages 19–30, 2000.

[21] J. M. Jensen, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *LNCS*, volume 2619, pages 331–346, 2003.

[22] C. Jones. Tentative steps toward a development for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[23] J. Krimm and L. Mounier. Compositional state space generation from lotos programs. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 239–258, London, UK, 1997. Springer-Verlag.

[24] K. Larsen, B. Steffen, and C. Weise. A constraint oriented proof methodology. In *Formal Systems Verification*, volume 1169 of *LNCS*, pages 405–435. Springer-Verlag, Nov. 1996.

[25] 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.

[26] K. L. Mcmillan. A methodology for hardware verification using compositional model checking. Technical report, Cadence Berkeley Labs, 1999.

[27] E. Mercer. *Correctness and Reduction in Timed Circuit Analysis*. PhD thesis, University of Utah, 2002.

[28] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Eng.*, SE-7(4):417–426, 1981.

[29] C. Myers, W. Belluomini, K. Killpack, E. Mercer, E. Peskin, and H. Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, Feb. 2001.

[30] C. J. Myers. *Asynchronous Circuit Design*. Wiley Inter-Science, 2001.

[31] A. Pnueli. In transition from global to modular temporal reasoning about programs. pages 123–144, 1985.

[32] C. S. Păsăreanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Form. Methods Syst. Des.*, 32(3):175–205, 2008.

[33] O. Roig, J. Cortadella, and E. Pastor. Hierarchical gate-level verification of speed-independent circuits. pages 129–137, May 1995.

[34] K. Stevens, R. Ginosar, and S. Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, 1999.

[35] S. Tasiran and R. K. Brayton. Stari: A case study in compositional and hierarchical timing verification. In *International Conference on Computer Aided Verification*, volume 1254 of *LNCS*. Springer-Verlag, 1997.

[36] H. Yao and H. Zheng. Automated interface refinement for compositional verification. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 27(3):433–446, 2009.

[37] 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, Mar. 1996.

[38] H. Zheng, E. Mercer, and C. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Transactions on Computer-Aided Design of Integrate Circuits and Systems*, 22(9):1138–1153, 2003.

[39] 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 of Integrate Circuits and Systems*, 25(3):403–412, 2006.

PLACE PHOTO HERE

Hao Zheng Hao Zheng received the M.S. and Ph.D degrees in Electrical Engineering from the University of Utah, Salt Lake City, UT, in 1998 and 2001, respectively. He worked as a research scientist for IBM Microelectronics Division from 2001 to 2004 to help make model checking a standard step in a ASIC design flow. Currently, he is an assistant professor of the Computer Science and Engineering department of the University of South Florida. His research interests include formal methods in computer system design and verification, parallel and distributed computing and its applications in design automation, and reconfigurable computing. His recent research includes development algorithms and methods that make model checking scalable to large systems. Zheng received an NSF CAREER award in 2006, and an USF Outstanding Research Achievement award in 2007.