Automated Interface Refinement for Compositional Verification

Haiqiong Yao, *Student Member, IEEE,* and Hao Zheng, *Member, IEEE*

***Abstract*—Compositional verification is essential for verifying large systems. However, approximate environments are needed when verifying the constituent modules in a system. Effective compositional verification requires finding a simple but accurate over-approximate environment for each module. Otherwise, many verification failures may be produced, therefore incuring high computational penalty for distinguishing the false failures from the real ones. This paper presents an automated method to refine the state space of each module within an over-approximate environment. This method is sound as long as an over-approximate environment is found for each module at the beginning of the verification process, and it has less restrictions on system partitioning. It is also coupled with several state space reduction techniques for better results. Experiments of this method on several large asynchronous designs show promising results.**

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

I. Introduction

Model checking has become a very important alternative to simulation for verifying complex concurrent systems. However, the state explosion problem limits it to small designs, a serious barrier which prevents its widespread acceptance. Although a number of techniques, such as symbolic model checking [7], [33], SAT based bounded model checking [5], [6], [4], and abstraction [14], [13], have been developed to alleviate the state space explosion problem, model checking still does not scale well as the system complexity increases. Compositional verification is viewed as one of the most promising approaches to attack state explosion by divide-and-conquer. It decomposes a large complex verification problem into simpler subtasks of lower complexity, each of which verifies a module in a system. The conclusion for the entire problem is drawn by combining the results from verifying the subtasks following certain compositional reasoning rules without actually verifying the entire system.

In general, properties of a module are satisfied only in a specific environment, which allows a module to be verified effectively in isolation. However, finding such an environment in traditional methods requires user guidance, which suffers from two severe weaknesses. First, it impairs the "push-button" characteristic of model checking. Second, assumptions provided by users are often error-prone and insufficient to model the concrete environment. Furthermore, if an environment is too coarse, the extra behaviors increase the chance of producing false counter-examples, which may incur a high computation penalty to distinguish them from the real ones.

Haiqiong Yao and Hao Zheng are with the CSE dept. of the Univ. of South Florida, Tampa, FL 33620. This research is supported by the CAREER Award contract# CCF-0546492 and an award CNS-0551621 from the National Science Foundation.

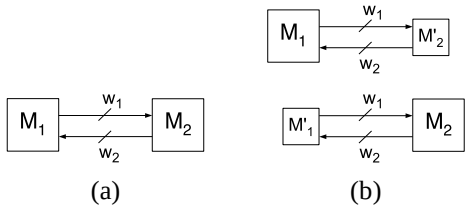

Fig. 1. Interface refinement for compositional verification. Block $1'$ and $2'$, abstractions of 1 and 2, are the environments for 2 and 1, respectively.

To address these problems, this paper presents a novel framework for compositional verification of concurrent systems. This framework combines an automated interface refinement method and several state space reduction techniques, and enables large systems to be verified. The automated interface refinement makes the state space of a module obtained from an over-approximate environment more accurate, thus alleviating the burden of finding an exact environment for a module at the beginning. It is based on the observation that modules are monotonic in that restricting the input behavior of a module does not increase its output behavior. A module can be refined by the behavior of its neighbor. And its restricted behavior can then be used to refine its neighbors.

To introduce the idea, refer to Fig.1. The inputs and outputs of M_1, w_2 and w_1, are the outputs and inputs of M_2, respectively. During verification, the inputs of M_1 and M_2 are driven by some environment abstraction M_2' and M_1', instead of M_2 and M_1 themselves. After finding the state space of M_1 and M_2, constraints on their outputs, w_1 and w_2, are derived. Since w_1 and w_2 are the inputs to M_2 and M_1, respectively, these derived constraints are used to restrict the behavior on w_1 and w_2 defined by M_1' and M_2', respectively. Since these modules are monotonic, more restricted constraints on the outputs may be derived from these modules after their input behavior is constrained. If so, the newly restricted output constraints are used in the next iteration to restrict the input behavior of the neighboring modules again. This process repeats until the output constraints from both modules can no longer be strengthened. Although the idea is illustrated using an example with two modules, it naturally applies to systems with an arbitrary number of modules. In a later section, this method is proved to be sound by showing that the refined state space of each module is still an abstraction of the exact one.

This framework also includes several state space reduction techniques that may help to extract stronger interface con-

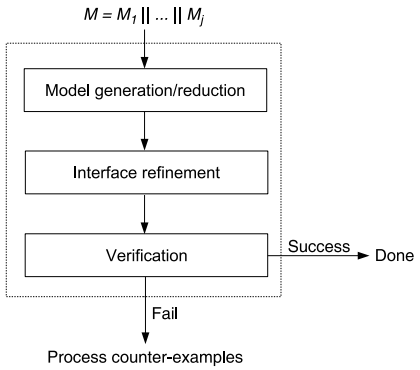

Fig. 2. The flow of Model checking with interface refinement.

straints, thus enabling the refinement to be more effective. They may also reduce the intermediate state space significantly to allow more flexible system partitioning by lowering the peak space requirement of the largest module in a system. In addition, they do not produce extra behavior compared to traditional abstraction approaches. Therefore, the only source of false failures is the over-approximate environment used for verifying each module. This is highly desirable because it requires less computation to confirm the found failures.

This method is not complete in that false counter-examples may still exist if the over-approximate environment is not completely refined. However, the chances of finding false counter-examples are significantly reduced when interfaces of the modules are refined to be more accurate. Even though false counter-examples may still show up after refinement, it would be much easier for users to refine the derived interface constraints further by hand because a substantial amount of unnecessary information has been removed. Efficiently distinguishing the false counter-examples itself is very important, and deserves extensive discussion in a separate presentation. It is not covered in this paper. Fig. 2 shows an overview of the verification flow in our method.

This paper is organized as follows: Section II gives an overview of the previous work on compositional reasoning and assumption learning methods. Section III gives a brief background review of the modeling and verification of concurrent systems. Section IV describes our compositional verification method. Section V addresses the automated interface constraint extraction and refinement method. Section VI presents the state space reduction techniques to make the interface refinement more effective. Section VII demonstrates our method on several large examples. The last section concludes the paper, and points out some future research directions.

II. Related Work

Compositional verification is essential to verifying large systems. It can be roughly classified as *compositional minimization* and *compositional reasoning*. Compositional minimization [8], [26], [30] in general constructs the local model for each module in a system, minimizes it, and composes it with the minimized models of other modules to form a reduced global model for the entire system, on which verification is performed. On the other hand, compositional verification based on *assume-guarantee* style reasoning [16], [27], [3], [29], [34] does not construct the global model. Instead, verification of a system is broken into separate analyses for each module of the system. The result for the entire system is derived from the results of the verified individual modules. When verifying each module, abstractions or assumptions about the environments with which the modules interact are needed for sound verification, and must be discharged later. The success of compositional reasoning relies on discovery of appropriate environment assumptions for every module. This is typically done by hand. If the modules have complex interactions with their environments, generating accurate environment assumptions can be challenging. Therefore, the requirement of manually finding assumptions has been a factor limiting the practical use of compositional reasoning.

In recent years, various approaches to automated assumption generation for compositional reasoning have been proposed. In the *learning-based* approaches, assumptions represented by deterministic finite automata are generated with the L^* learning algorithm and analysis of local counter-examples [37], [2], [19], [25], [10]. The learned assumptions can result in orders of magnitude reduction in verification complexity. However, these approaches may generate assumptions with too many states and fail verification in some cases [37], [2].

Comparatively, this method has several significant differences from the learning based ones. First, interface behavior of a module is encoded implicitly in Boolean formulas for interface signals instead of finite automata. Second, the interface behavior of a module is refined by iteratively examining the interactions between the module and its neighbors, rather than relying on local counter-example analysis. However, there is nothing to prevent counter-examples from being used to further refine module interfaces. Not using counter-examples for refinement allows more freedom in system partitioning, while existing learning based methods seem more suitable only for two-module partitions as discussed in [37], [18]. Third and probably more importantly, the interface constraints generated by this method are not assumptions. Therefore, there is no need to discharge the interface constraints later on. Despite the differences, this method and the learning-based methods can be combined to achieve better results.

In addition, this method facilitates verification reuse. If a module in a system is modified and its interface becomes more restricted, then refinement can be applied again on top of the results from the previous iteration. On the other hand, refinement is not needed if the interface of the modified modules become more liberal. In both cases, verification can be limited to the modified modules.

In the *interface constraint-based* approaches, restrictions from environment are imposed on the modules of a system to remove the behavior that should not take place. Generation of interface constraints based on the analysis of synchronization between modules is proposed by Cheung and Kramer [11]. However, it cannot capture effective interface constraints due to deficiencies in analysis of synchronization between distant modules. Alfaro and Henzinger provide interface automata to represent a module and its environment [20], [21], [22].

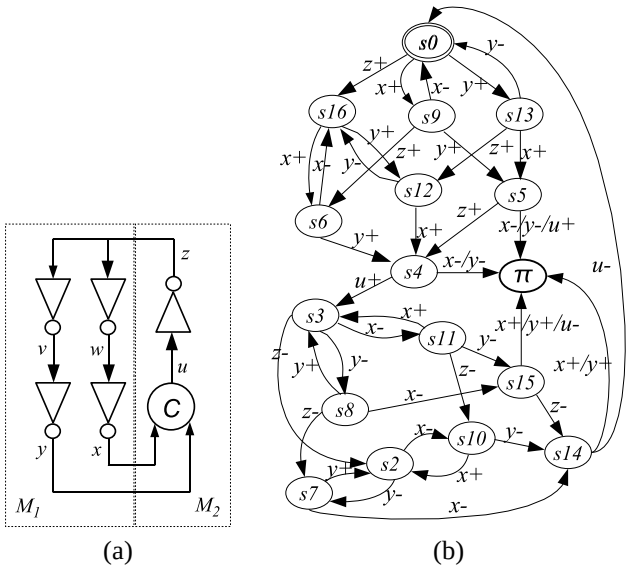

Fig. 3. (a) A simple asynchronous circuit. (b) The SG for module M_2 where both inputs are set to be completely free.

The module and the environment are refined in an alternating fashion so that the module accepts only input actions generated by the environment, and issues output actions corresponding to these input actions. Refinement of interface automata in the component-based design is similar to refinement of environment assumptions in compositional verification [1] [24], [31]. A similar approach, *thread-modular reasoning*, is proposed in [28] for multithreaded program verification.

Counterexample guided abstraction refinement (CEGAR) [17], [15], [13] uses a set of abstraction predicates to build a reduced finite state model for a system. If such a model passes verification, the concrete system is concluded to be correct. Otherwise, the abstract model is iteratively refined by adding more relevant variables based on the analysis of the spurious counterexamples until the model passes verification, or a counterexample is confirmed to be genuine. Therefore, the concept of CEGAR is similar to that of learning-based compositional verification approaches. However, the learning-based approaches are a step forward such that verification is applied to an abstract model of each module in a system, instead of a global model of the entire system. CEGAR is first coupled with compositional verification in [9].

III. Preliminaries

State graphs are used to model the behavior of concurrent systems. This section introduces basic notations and definitions for state graphs and their relative operators. It presents how the correctness of safety properties is formulated and checked in this framework.

A. State Graphs

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 or firings cause the movement from one state to another. The definition of state graphs is given as follows.

Definition 3.1 (State Graphs): A state graph (SG) G 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 G, and represents violations of some prescribed 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 visible or invisible behavior of systems. For a SG, $\mathcal{A} = \mathcal{A}^I \cup \mathcal{A}^O \cup \{\zeta\}$. $\mathcal{A}^I$ is the set of actions generated by an environment of a system such that the system can only observe and react. $\mathcal{A}^O$ is the set of actions generated by a system responding to its environment. ζ represents the internal behavior of a system invisible on the interface. Consequently, for a state transition $(s_1, \zeta, s_2) \in R$, the environment cannot distinguish between s_1 and s_2 due to $L(s_1) = L(s_2)$. In other words, execution of invisible actions does not affect the state labellings. 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 G. We assume that the state transition set R is total such that every state has some successor. A SG G is deterministic if for all states $s, s', s'' \in S$ and all actions $a \in \mathcal{A}$, $R(s, a, s')$ and $R(s, a, s'')$ hold, then $s' = s''$. Otherwise, G is non-deterministic. Our method allows non-deterministic SGs.

Fig.3(a) shows a simple asynchronous circuit. 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 two modules, M_1 and M_2. Fig.3(b) shows the corresponding SG for M_2 where both of its inputs are set to be totally free, meaning they can change to high or low in any state. In asynchronous circuits, each wire w has two actions, $w+$ and $w-$. For M_2, $P = \{x, y, z, \neg x, \neg y, \neg z\}$, its input actions $\mathcal{A}^I = \{x+, x-, y+, y-\}$, and its output actions $\mathcal{A}^O = \{z+, z-\}$, and its invisible actions are $\{u+, u-\}$. To make this and following figures of SGs readable, state labellings are not shown. As an example, the labeling of s_0 of M_2 is $\{\neg x, \neg y, \neg z\}$.

A *path* of G is an infinite sequence of alternating states and actions $\rho = (s_0, a_0, s_1, a_1, s_2, \cdots)$ such that $s_0 = init$, $s_i \in S$, $a_i \in \mathcal{A}$, and $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$. A path is *autonomous* if all actions on that path are in $\mathcal{A}^O \cup \{\zeta\}$. An autonomous path executes irrespective of input actions. A path is *visible* if it does not contain any ζ action. Given a SG G, the set of all paths starting from the initial state $init$ is the language of G, denoted as $\mathcal{L}(G)$. A subpath is defined as a fragment of a path such that $\hat{\rho} = (s_i, a_i, s_{i+1}, a_{i+1}, \cdots, s_{i+j})$ for $i, j \geq 0$. A state $s' \in S$ is *reachable from* a state $s \in S$ if there exists a subpath $\hat{\rho} = (s_0, a_0, s_1, a_1, s_2, \cdots, s_n)$ such

that $s_0 = s$ and $s_n = s'$. A state s is reachable in G if s is reachable from the initial state $init$.

Given a path, its *projection* onto a set of visible actions is defined as follows.

Definition 3.2 (Path Projection): Let $\rho = (s_0,a_0,s_1,a_1,\cdots)$ be a path. Its projection over a set of visible actions $\mathcal{A}' \subseteq \mathcal{A}^I \cup \mathcal{A}^O$, denoted by $\rho[\mathcal{A}']$, is an sequence of alternating states and actions

$$\rho[\mathcal{A}'] = (s'_0, a'_0, s'_1, a'_1, \cdots)$$

such that

$$\rho[\mathcal{A}'] \quad = \quad \begin{cases} \rho' & \text{if } a_0 \notin \mathcal{A}' \text{ or } a_0 = \zeta, \\ (s_0, a_0) \circ \rho'[\mathcal{A}'] & \text{otherwise.} \end{cases}$$

where $\rho' = (s_1, a_1, s_2, ...)$, and $(s_0, a_0) \circ \rho'[\mathcal{A}']$ is the concatenation of (s_0, a_0) and $\rho'[\mathcal{A}']$.

Definition 3.3 (Observable Equivalence): Let ρ and ρ' be two paths, $\mathcal{A}' \subseteq \mathcal{A}^I \cup \mathcal{A}^O$, and their projections be

$$\rho[\mathcal{A}'] = (s_0, a_0, s_1, a_1, \cdots) \text{ and } \rho'[\mathcal{A}'] = (s'_0, a'_0, s'_1, a'_1, \cdots).$$

ρ and ρ' are observably equivalent, denoted as $\rho \sim \rho'$, iff

$$\forall i \geq 0.\ L(s_i) = L(s'_i) \wedge a_i = a'_i.$$

The observable equivalence is used in Section III-B to define a relation between SGs.

Given a system with multiple modules, its SG can be constructed by composing the module SGs in parallel. Two SGs can be composed if their output action sets are disjoint.

Definition 3.4 (Parallel Composition of SG): Let

$$G_1 = (P_1, \mathcal{A}_1, S_1, init_1, R_1, L_1)$$
$$G_2 = (P_2, \mathcal{A}_2, S_2, init_2, R_2, L_2)$$

be two SGs. If $\mathcal{A}_1^O \cap \mathcal{A}_2^O = \emptyset$, the parallel composition of G_1 and G_2 is defined as

$$G_1 \| G_2 = (P_1 \cup P_2, \mathcal{A}_1 \cup \mathcal{A}_2, S, (init_1, init_2), R, L)$$

where

1) $S = \{(s_1, s_2) \mid s_1 \in S_1 \wedge s_2 \in S_2\}$, and
   a) $(s_1 = \pi \Rightarrow s_2 = \pi) \wedge (s_2 = \pi \Rightarrow s_1 = \pi)$,
   b) $L(s_1) \cap P_2 = L(s_2) \cap P_1$.
2) $\forall (s_1, s_2) \in S.\ L(s_1, s_2) = L_1(s_1) \cup L_2(s_2)$.
3) $R \subseteq S \times \mathcal{A} \times S$ such that $\forall s_1 \in S_1 . \forall s_2 \in S_2.\ (s_1, s_2) \in S$, $s_1 \neq \pi$, $s_2 \neq \pi$, and
   a) $\forall a \in \mathcal{A}_1 - \mathcal{A}_2.\ R_1(s_1, a, s'_1)$ and

$$\begin{cases} s'_1 \neq \pi & \Rightarrow \quad R((s_1, s_2), a, (s'_1, s_2)) \\ s'_1 = \pi & \Rightarrow \quad R((s_1, s_2), a, (\pi, \pi)) \end{cases}$$

   b) $\forall a \in \mathcal{A}_2 - \mathcal{A}_1.\ R_2(s_2, a, s'_2)$ and

$$\begin{cases} s'_2 \neq \pi & \Rightarrow \quad R((s_1, s_2), a, (s_1, s'_2)) \\ s'_2 = \pi & \Rightarrow \quad R((s_1, s_2), a, (\pi, \pi)) \end{cases}$$

   c) $\forall a \in \mathcal{A}_1 \cap \mathcal{A}_2.\ R_1(s_1, a, s'_1) \wedge R_2(s_2, a, s'_2)$ and

$$\begin{cases} s'_1 \neq \pi \wedge s'_2 \neq \pi & \Rightarrow \quad R((s_1, s_2), a, (s'_1, s'_2)) \\ s'_1 = \pi \vee s'_2 = \pi & \Rightarrow \quad R((s_1, s_2), a, (\pi, \pi)) \end{cases}$$

   Similarly, R also includes $((\pi, \pi), a, (\pi, \pi))$ for all $a \in \mathcal{A}_1 \cup \mathcal{A}_2$.

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 the two individual SGs. It has been shown that parallel composition of SGs is commutative and associative in [12].

B. Correctness and Conformance Relation

Failure state π can be used to represent various undesirable behavior that a system is not expected to produce. A system is regarded as being correct if π is not reachable in its SG. A path is referred to as a *failure trace* if a SG contains the failure state π reachable via such path. The set of all failure traces of a SG G is denoted as $\mathcal{F}(G)$. Obviously, $\mathcal{F}(G) \subseteq \mathcal{L}(G)$. A system is correct if $\mathcal{F}(G) = \emptyset$.

Given a failure trace $\rho = (s_0, a_0, \cdots, s_i, a_i, \pi, \cdots)$, 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 trace. In such case, ρ and ρ' are called *failure equivalent*.

Definition 3.5 (Failure Equivalence): Given two path $\rho = (s_0, \cdots, s_i, a_i, \pi, \cdots)$ and $\rho' = (s'_0, \cdots, s'_i, a'_i, \cdots)$, ρ and ρ' are failure equivalent, denoted as $\rho \sim_F \rho'$ iff

$$\forall (0 \leq h \leq i).\ L(s_h) = L(s'_h) \wedge a_h = a'_h$$

The definition of conformance relation between two SGs is given as follows.

Definition 3.6 (Conformance): Given SGs G and G', G conforms to G', denoted as $G \preceq G'$, iff the following conditions hold:

1) $P' = P$.
2) $\mathcal{A}' = \mathcal{A}$.
3) $\forall s \in S$, $\exists s' \in S'$ such that $L(s) = L'(s')$
4) For every path $\rho \in \mathcal{L}(G)$, there exists a path $\rho' \in \mathcal{L}(G')$ such that $\rho \sim \rho'$ or $\rho \sim_F \rho'$.

Intuitively, the conformance relation states that any visible path of G is also a visible path of G'. Also, for any failure trace in G, there exists an equivalent failure trace in G'. In other words, the language accepted by G is also accepted by G'. Therefore, given G and G', they satisfy the following property:

$$G \preceq G' \wedge \mathcal{F}(G') = \emptyset \quad \Rightarrow \quad \mathcal{F}(G) = \emptyset.$$

This property states that G is correct if G' is correct.

IV. Compositional Verification

This section describes our compositional verification method. This method assumes that a system is described in some high level modeling language, and it is constructed by parallel composition of simpler modules, $M = M_1 \| \cdots \| M_n$, without giving the definition of $\|$ for such a language. By virtue of the complexity of the entire system M, our goal is to check the correctness of M by verifying each M_i without actually composing them. If each individual M_i is verified correctly, then the entire system is correct.

When a module M_i is considered, the rest of the system is regarded as the environment of M_i, denoted as $\mathcal{E}_i$. The task of verifying M can be decomposed into n sub-problems of verifying $M_i \| \mathcal{E}_i$ for $1 \leq i \leq n$ where only failures in a module are checked in each sub-problem. However, simply composing the modules other than M_i in the system into $\mathcal{E}_i$ would make the complexity of verifying $M_i \| \mathcal{E}_i$ be very close to that of verifying the entire system M. To reduce the complexity, it needs to find an approximation of $\mathcal{E}_i$, $\mathcal{E}_i'$, such that

1) $\mathcal{E}_i'$ is much simpler than $\mathcal{E}_i$ in terms of the number of states in the resultant SG, and
2) the same conclusion for verifying $M_i \| \mathcal{E}_i$ can be drawn from verifying $M_i \| \mathcal{E}_i'$.

In practice, finding such an ideal approximate environment for a module to satisfy both requirements is extremely difficult, if not impossible. Therefore, our method loosens the requirements of an approximate environment to be much simpler but preserve all the interface behavior of $\mathcal{E}_i$. This ensures that M_i is failure free in the entire system if it is failure free in $\mathcal{E}_i'$. If this is true for every module, the entire system is guaranteed to be failure free. The above discussion is formalized in the the following compositional verification rule.

$$\frac{\begin{array}{rll} 1: & \mathcal{F}(G_i') = \emptyset & \text{for } 1 \leq i \leq n \\ 2: & G_i \preceq G_i' & \text{for } 1 \leq i \leq n \end{array}}{\mathcal{F}(G_1 \| \cdots \| G_n) = \emptyset}$$

where G_i and G_i' are the SGs generated from $M_i \| \mathcal{E}_i$ and $M_i \| \mathcal{E}_i'$, respectively, using some state space exploration algorithm. This rule is sound but incomplete. The proof for soundness is straightforward, and is not be given. On the other hand, the approximate environment may result in behavior impossible in the entire system that causes a module to fail, resulting in false failures.

Algorithm 1 shows our verification framework based on the above compositional rule. In this algorithm, a SG G_i containing all reachable states and the corresponding state transitions is generated first for each $M_i \| \mathcal{E}_i'$. This step can be accomplished using some existing depth first search algorithm such as the one in [35]. Thereafter, a series of reductions is applied to G_i to control the size of these SGs in terms of the number of states and state transitions. These reductions, performed by procedures `autofailure`, `abstract` and `rmRed`, are described in detail in Section VI.

The challenges to the efficiency of algorithm `verify` include finding a much simpler yet accurate environment for each module, therefore reducing the extra behaviors in G_i introduced by an over-approximate environment. To meet this challenge, an interface refinement approach, implemented by procedure `refine` on line 6 in the algorithm, is developed and described in detail in the next section. It takes the SGs G_i and their respective initial constraints $\mathcal{C}_i$ for $1 \leq i \leq n$, and reduces iteratively state transitions from G_i invalidated by the constraints. After all SGs cannot be refined further, the algorithm checks each G_i and reports failures found in any of them. Concepts of constraints and how constraints are found and used to reduce SGs are introduced in the next section.

Algorithm 1: `verify`$(M = M_1 \| \cdots \| M_n)$

```
1  foreach i, 1 ≤ i ≤ n do
2      find SG G_i for M_i ‖ E_i';
3      autofailure(G_i);
4      abstract(G_i);
5      rmRed(G_i);
6  refine({G_1, ···, G_n}, {C_1, ···, C_n});
7  foreach i, 1 ≤ i ≤ n do
8      if F(G_i) ≠ ∅ then
9          return "M has a failure";
10 return "M is failure free";
```

V. Interface Refinement

This section describes an interface refinement algorithm that makes the SGs obtained from the system modules with approximate environment more accurate. This algorithm is fully automated, and iteratively generates more accurate yet conservative interface constraints to refine the SGs as long as the initially selected environments for the modules are abstractions of the exact ones.

A. Definition and Properties 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*, which are defined as follows.

Definition 5.1 (Constraints): Let $G = (P, \mathcal{A}, S, init, R, L)$ be a SG. Let $f : 2^P \rightarrow \{\text{FALSE}, \text{TRUE}\}$ be a Boolean function defined over P. A constraint $\mathcal{C} = \{(a, f) | a \in \mathcal{A}\}$ of G is a set of pairs of actions of G 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 $\forall a \in \mathcal{A}. \mathcal{C}_1(a) \Rightarrow \mathcal{C}_2(a)$.

This section assumes that constraints are defined for all actions of SGs to simplify presentation. A constraint for actions may be provided by users, or derived automatically as shown in the following sections. When a constraint is imposed on actions, it may restrict how actions are enabled, therefore causing some state transitions to become invalid.

Definition 5.2 (Valid State Transitions): 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 G 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.

$$\forall a \in \mathcal{A}. \left((\mathcal{C}_1(a) \Rightarrow \mathcal{C}_2(a)) \Leftrightarrow (R_{\mathcal{C}_1(a)} \subseteq R_{\mathcal{C}_2(a)})\right) \quad (1)$$

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.

According to the above discussion, a reduced SG results from applying a stronger constraint.

Definition 5.3 (Applying Constraint): Let G be a SG such that $G = (P,\mathcal{A},S, init, R, L)$, and $\mathcal{C}$ be a constraint on $\mathcal{A}$. Applying $\mathcal{C}$ to G, denoted as $\langle\mathcal{C}\rangle G$, results in a new SG $G' = (P', \mathcal{A}', S', init', R', L')$ such that

1) $P' = P$, $\mathcal{A}' = \mathcal{A}$, $S' = S$, $init' = init$, $L' = L$, and
2) $R' = \bigcup_{\forall a \in \mathcal{A}} \left(R_{\mathcal{C}(a)} \cup \{(\pi, a, \pi)\}\right)$.

By the definition of constraints and conformance, a constraint $\mathcal{C}_1$ is stronger than another constraint $\mathcal{C}_2$ iff one SG imposed with $\mathcal{C}_1$ accepts a subset of language of a SG imposed with $\mathcal{C}_2$. This is formulated in the following lemma.

Lemma 5.1: Let $G = (P, \mathcal{A}, S, init, R, L)$ be a SG, $\mathcal{C}_1$ and $\mathcal{C}_2$ two constraints on $\mathcal{A}$. Then, the following property holds.

$$\left(\mathcal{C}_1 \Rightarrow \mathcal{C}_2\right) \quad \Leftrightarrow \quad \left(\langle\mathcal{C}_1\rangle G \preceq \langle\mathcal{C}_2\rangle G\right)$$

Proof: Let $G_1 = \langle\mathcal{C}_1\rangle G$, $G_2 = \langle\mathcal{C}_2\rangle G$. Therefore,

$$R_1 = \bigcup_{\forall a \in \mathcal{A}} \left(R_{\mathcal{C}_1(a)} \cup \{(\pi, a, \pi)\}\right)$$

$$R_2 = \bigcup_{\forall a \in \mathcal{A}} \left(R_{\mathcal{C}_2(a)} \cup \{(\pi, a, \pi)\}\right)$$

First, according to (1), $\forall a \in \mathcal{A}.\ R_{\mathcal{C}_1}(a) \subseteq R_{\mathcal{C}_2}(a)$ holds on account of $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$. Hence, $G_1 \preceq G_2$ holds.

Next, for every path $\rho_1 \in \mathcal{L}(G_1)$, there exists a path $\rho_2 \in \mathcal{L}(G_2)$ such that $\rho_1 \sim \rho_2$. ρ_1 consists of the state transitions from R_1 and ρ_2 from R_2. This implies $R_1 \subseteq R_2$. Thus, for $\forall a \in \mathcal{A}. R_{\mathcal{C}_1(a)} \subseteq R_{\mathcal{C}_2(a)}$, which leads to $\forall a \in \mathcal{A}. \mathcal{C}_1(a) \Rightarrow \mathcal{C}_2(a)$ by (1). Hence, $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ holds. ∎

The following lemma states that the conformance relation between two SGs is preserved when the same constraint is applied to both of them.

Lemma 5.2: Let G_1 and G_2 be two SGs with the same $\mathcal{A}$, and $\mathcal{C}$ a constraint on $\mathcal{A}$. The following property holds

$$\left(G_1 \preceq G_2\right) \quad \Rightarrow \quad \left(\langle\mathcal{C}\rangle G_1 \preceq \langle\mathcal{C}\rangle G_2\right)$$

Proof: Since $G_1 \preceq G_2$, for every path $\rho_1 = (s_0, a_0, s_1, \cdots)$ in $\mathcal{L}(G_1)$, there exists a path $\rho_2 \in \mathcal{L}(G_2)$ such that $\rho_1 \sim \rho_2$. If all state transitions (s_i, a_i, s_{i+1}) for $0 \leq i$ on ρ_1 are valid with respect to $\mathcal{C}$, they are also valid in G_2 with respect to $\mathcal{C}$. In other words, a path that is valid in $\langle\mathcal{C}\rangle G_1$ is also valid in $\langle\mathcal{C}\rangle G_2$. ∎

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. This is defined as follows.

Definition 5.4 (Extraction of Constraint): Let G be a SG such that $G = (P, \mathcal{A}, S, init, R, L)$. The constraint $\mathcal{C}$ extracted from G, denoted by $G\langle\mathcal{C}\rangle$, 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 G_1 and G_2 be two SGs such that $G_1 \preceq G_2$. According to the definition of the conformance relation, the behavior of G_1 is more restricted than that of G_2. This implies that the enabling condition of an action is more restricted in G_1 than in G_2. This indicates that a stronger constraint may be derived from the refined SG.

Lemma 5.3: Let G_1 and G_2 be two SGs, and $\mathcal{C}_1$ and $\mathcal{C}_2$ two constraints derived by $G_1\langle\mathcal{C}_1\rangle$ and $G_2\langle\mathcal{C}_2\rangle$, respectively. Then the following property holds.

$$\left(G_1 \preceq G_2\right) \quad \Rightarrow \quad \left(\mathcal{C}_1 \Rightarrow \mathcal{C}_2\right)$$

Proof: Since $G_1 \preceq G_2$, for every path $\rho_1 \in \mathcal{L}(G_1)$, there exists a path $\rho_2 \in \mathcal{L}(G_2)$ such that $\rho_1 \sim \rho_2$. By the definition of observable equivalence and constraints, for every $a \in \mathcal{A}$, if it is enabled on path ρ_1, it is also enabled on path ρ_2. It is possible that G_2 may have some path that does not exist in G_1. This implies that an action may be enabled on some path in G_2 but not enabled in G_1. To summarize, any action, if enabled in G_1, is also enabled in G_2, but this is not true in the other direction. This is equivalent to $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$. ∎

B. Interface Refinement for Compositional Verification

The previous section shows that accurate constraints help refine SGs by removing invalid state transitions. However, manually generating such constraints may be too expensive. This section proposes an algorithm to automatically derive constraints from and subsequently apply them to the abstract SGs. This algorithm iterates until the constraints for all SGs cannot be strengthened, and all SGs cannot be reduced.

To simplify the discussion, consider a system of two modules, $G = G_1 \| G_2$, such that $\mathcal{A}_1^I = \mathcal{A}_2^O$ and $\mathcal{A}_1^O = \mathcal{A}_2^I$. In the sequel, the input and output constraints refer to those on input and output actions of a module, respectively. A shared action between G_1 and G_2 is in $\mathcal{A}_1 \cap \mathcal{A}_2$. If a shared action a is in $\mathcal{A}_1^O \cap \mathcal{A}_2^I$, then the output constraints on a derived from G_1 can be used as input constraints to reduce G_2 by pruning the invalid state transitions on a. The case where a is in $\mathcal{A}_2^O \cap \mathcal{A}_1^I$ is handled similarly.

The essence of interface refinement lies in the alternating refinement on G_1 and G_2 with the interface constraints. When refining a SG, the output constraints derived from other SGs are applied to the considered SG where the invalid state transitions on input actions are removed. The output constraints are extracted from the reduced SGs, and then serve as input constraints for other SGs in the next iteration. Let $\mathcal{C}_1^i$ and $\mathcal{C}_2^i$ be the output constraints extracted from G_1^i and G_2^i at the ith iteration, respectively. The iterative process of interface refinement is illustrated as follows.

$$\begin{array}{rcl} \text{iteration } 0 & : & \langle\mathcal{C}_2^0\rangle G_1^0\langle\mathcal{C}_1^1\rangle, \quad \langle\mathcal{C}_1^0\rangle G_2^0\langle\mathcal{C}_2^1\rangle \\ \text{iteration } 1 & : & \langle\mathcal{C}_2^1\rangle G_1^1\langle\mathcal{C}_1^2\rangle, \quad \langle\mathcal{C}_1^1\rangle G_2^1\langle\mathcal{C}_2^2\rangle \\ & & \ldots \\ \text{iteration } l & : & \langle\mathcal{C}_2^l\rangle G_1^l\langle\mathcal{C}_1^{l+1}\rangle, \quad \langle\mathcal{C}_1^l\rangle G_2^l\langle\mathcal{C}_2^{l+1}\rangle \end{array}$$

where $\langle\mathcal{C}_2^i\rangle G_1^i\langle\mathcal{C}_1^{i+1}\rangle$ specifies that input constraint $\mathcal{C}_2^i$ is applied on G_1^i, and output constraint $\mathcal{C}_1^{i+1}$ is derived from $\langle\mathcal{C}_2^i\rangle G_1^i$. Let $G_1^{i+1} = \langle\mathcal{C}_2^i\rangle G_1^i$ and $\mathcal{C}_1^{TRUE} = \{(a, \text{TRUE}) \mid \forall a.\ a \in \mathcal{A}_1\}$. Since $G_1^i = \langle\mathcal{C}_1^{TRUE}\rangle G_1^i$ and $\mathcal{C}_2^i \Rightarrow \mathcal{C}_1^{TRUE}$, we have $G_1^{i+1} \preceq G_i^i$ by Lemma 5.1. The enabling condition of the output actions of G_1^{i+1} may become more restricted after applying $\mathcal{C}_2^i$, therefore $\mathcal{C}_1^{i+1} \Rightarrow \mathcal{C}_1^i$ by Lemma 5.3. The stronger constraint $\mathcal{C}_1^{i+1}$ extracted from the reduced G_1^{i+1} is used as the input constraint for G_2^{i+1}. The same reasoning applies to G_2^i. The above process terminates in the lth iteration when the extracted output constraints of all modules are stable, e.g. $\mathcal{C}_1^l = \mathcal{C}_1^{l+1}$ and $\mathcal{C}_2^l = \mathcal{C}_2^{l+1}$. This implies that G_1^l and G_2^l cannot be reduced anymore.

Theorem 5.1 below proves the soundness of the interface refinement process. It shows that our compositional verification method combined with the described refinement process is still sound in that the refined SGs are still abstractions of the exact SGs after refinement.

To prove Theorem 5.1, exact SGs need to be defined. Intuitively, the SG of a module is exact if its behavior is exactly the same when it is embedded in a larger system. The formal definition of exact SGs is shown as follows.

Definition 5.5 (Exact SGs): Let G_1 and G_2 be two SGs. G_1 is exact within $G = G_1 \| G_2$ if the following hold.

$$\forall(s_1, a, s_1') \in R_1, \text{ there exists a } ((s_1, s_2), a, (s_1', s_2')) \in R.$$

From the definition, the following property holds for the exact G_1

$$G_1 = \langle\mathcal{C}_2\rangle G_1 \tag{2}$$

where $\mathcal{C}_2$ is obtained by $G_2\langle\mathcal{C}_2\rangle$.

Theorem 5.1: Let G_1 and G_2 be exact within $G_1 \| G_2$. If G_1', and G_2' are SGs such that

$$G_1 \preceq G_1' \text{ and } G_2 \preceq G_2'$$

the following property holds.

$$G_1 \preceq \langle\mathcal{C}_2'\rangle G_1' \text{ and } G_2 \preceq \langle\mathcal{C}_1'\rangle G_2'$$

where $\mathcal{C}_i'$ is obtained by $G_i'\langle\mathcal{C}_i'\rangle$ for $i = 1, 2$.

Proof: According to Lemma 5.2,

$$\langle\mathcal{C}_2'\rangle G_1 \preceq \langle\mathcal{C}_2'\rangle G_1' \text{ and } \langle\mathcal{C}_1'\rangle G_2 \preceq \langle\mathcal{C}_1'\rangle G_2'$$

Let $\mathcal{C}_i$ be the constraints obtained by $G_i\langle\mathcal{C}_i\rangle$ for $i = 1, 2$. According to Lemma 5.3, $\forall a \in \mathcal{A}_i^O.\ \mathcal{C}_i \Rightarrow \mathcal{C}_i'$ for $i = 1, 2$. Again, according to Lemma 5.1,

$$\langle\mathcal{C}_2\rangle G_1 \preceq \langle\mathcal{C}_2'\rangle G_1 \text{ and } \langle\mathcal{C}_1\rangle G_2 \preceq \langle\mathcal{C}_1'\rangle G_2$$

Combining the results in the above steps, we have

$$\langle\mathcal{C}_2\rangle G_1 \preceq \langle\mathcal{C}_2'\rangle G_1' \text{ and } \langle\mathcal{C}_1\rangle G_2 \preceq \langle\mathcal{C}_1'\rangle G_2'$$

According to (2), $\langle\mathcal{C}_2\rangle G_1 = G_1$ and $\langle\mathcal{C}_1\rangle G_2 = G_2$. Therefore, $G_1 \preceq \langle\mathcal{C}_2'\rangle G_1'$ and $G_2 \preceq \langle\mathcal{C}_1'\rangle G_2'$. This completes the proof. ∎

Function `refine` shown in Algorithm 2 implements the interface refinement process presented above. It takes as arguments a set of SGs G_i, each of which is generated from a module in a system with an over-approximate environment, and a set of initial constraints $\mathcal{C}_i$ on the outputs of each module. The algorithm first merges these constraints into a single set, and then iteratively applies the constraint to reduce each SG and extracts new output constraints from the reduced SGs until the constraint does not change anymore. At this point, all state transitions in every SG are valid with respect to the constraints extracted from their neighbors, therefore no further reduction is possible. The initial constraints may be provided by users or obtained from high level representations. These constraints may be very abstract at the beginning, and may possibly be set to TRUE for all actions by default if nothing is known about the input interface of a module. However, more restricted initial constraints help reduce the number of iterations. Functions `apply` and `extract` follow the Definitions 5.3 and 5.4, and are described in more detail in the next section.

Algorithm 2: `refine`$(\{G_1, \cdots, G_n\}, \{\mathcal{C}_1, \cdots, \mathcal{C}_n\})$

```
1 C' = C_1 ∪ ... ∪ C_n;
2 C = ∅;
3 while C ≠ C' do
4     C = C';
5     C' = ∅;
6     foreach G_i, 0 ≤ i ≤ n do
7         apply(G_i, C);
8         C_i = extract(G_i);
9         C' = C' ∪ C_i;
```

Next, the complexity of the above algorithm in terms of the number of iterations needed to find the stable constraints is considered. Assume that the size of a SG G_i, $|G_i|$, is measured by the number of state transitions in R_i of G_i. Suppose the number of modules in a system is n and $|G_i| \leq m$ for all $1 \leq i \leq n$. In theory, the number of iterations needed to find the stable constraints is $O(mn)$. This complexity can be understood as follows. Consider the extreme case where exactly one state transition of exactly one SG is removed in each iteration. And suppose that all state transitions in G_i can be removed. Obviously, the process stops when the state transition set R_i of every G_i is reduced to be empty. Therefore, the maximal number of iterations necessary for termination is $O(mn)$. Although this complexity seems very high, in practice

the total number of iterations is not that large because many state transitions can be eliminated from multiple modules in a single iteration as shown by the experimental results.

C. Application and Extraction of Constraints

In the above discussion, the application of a constraint to and extraction of a constraint from a SG are represented as $\langle\mathcal{C}\rangle G\langle\mathcal{C}'\rangle$. This section shows how to reduce SGs by applying a constraint on a SG, i.e. $\langle\mathcal{C}\rangle G$, and extract a constraint from a SG, i.e. $G\langle\mathcal{C}'\rangle$.

Given a SG G and a constraint $\mathcal{C}$, the objective is to apply $\mathcal{C}$ on G to remove the invalid state transitions in G. A state transition $(s, a, s') \in R$ of G such that $s \neq \pi$ is invalid if $conj(s) \not\Rightarrow \mathcal{C}(a)$. The removal of the state transitions may render some states unreachable in G when all of their incoming state transitions are eliminated. In the last step, all unreachable states and their outgoing state transitions are also removed. Algorithm 3 shows the procedure to reduce G with $\mathcal{C}$.

Algorithm 3: `apply`$(G_i, \mathcal{C})$

```
1 foreach (s, a, s') ∈ R_i ∧ s ≠ π ∧ a ∈ A_i^I do
2     if conj(s) ⇒ C(a) does not hold then
3         Delete (s, a, s') from R_i;
4 Remove unreachable states and transitions from G_i;
```

Notice that constraint $\mathcal{C}$ is applied only on the input actions in Algorithm 3. In general, the constraint provided to function `apply` can be on either input or output actions. For example, when one describes a system, a constraint may be used to elaborate the system description additionally. This constraint can be created for any actions. However, when function `apply` is used for a SG in the above interface refinement framework, only the part of the constraint extracted from other SGs for the input actions of the SG under consideration is necessary. The part of the constraint for the output actions of this SG would not reduce this SG because it is extracted from itself. Therefore, only the state transitions labeled with input actions of a SG may be removed with respect to constraint $\mathcal{C}$ when `apply` is invoked. As a side effect, some other state transitions, when become unreachable due to the removed state transitions on input actions, may also be removed.

Each module updates its behavior on its output actions, while its input actions are defined by the environment. Therefore, given a SG of a module, only the constraint for non-input actions are extracted. However, the behavior on internal action ζ of a SG is invisible to other SGs, and the constraint for the internal actions is meaningless to other modules. Therefore, the constraint is extracted only for the output actions as shown in Algorithm 4.

D. Example

For the modules M_1 and M_2 in the circuit shown in Fig.3(a), their SGs after all reductions, which are described in the next section, are shown in Fig.6(a) and (b), respectively. How the presented refinement method removes the invalid state transitions is illustrated as follows.

Algorithm 4: `extract`(G_i)

```
1 foreach a ∈ A_i^O do
2     Add (a, FALSE) into C_i;
3 foreach (s, a, s') ∈ R_i and s ≠ π and a ∈ A_i^O do
4     Replace (a, f) ∈ C_i with (a, f ∨ conj(s));
5 return C_i;
```

First, $conj(s)$ for each non-failure state of the SG in Fig.6(a) on wires x, y, and z are listed as follows.

$$s_0 : \neg x \wedge \neg y \wedge \neg z \qquad s_2 : x \wedge y \wedge z \qquad s_3 : \neg x \wedge y \wedge z$$
$$s_4 : x \wedge \neg y \wedge z \qquad s_5 : \neg x \wedge \neg y \wedge z \qquad s_6 : x \wedge y \wedge \neg z$$
$$s_7 : x \wedge \neg y \wedge \neg z \qquad s_8 : \neg x \wedge y \wedge \neg z$$

Since the outputs of M_1 are x and y, which are the inputs of M_2, the constraint for actions on x and y is found by disjoining $conj(s)$ for all $(s, a, s') \in R$ such that a is on x or y and $s \neq \pi$. $x+$ is enabled in state s_3 and s_5. After disjoining $conj(s_3)$ and $conj(s_5)$, the constraint for $x+$ is $\neg x \wedge z$. For this example, the constraint C_1^0 for all actions is shown as follows.

$$x+ : \neg x \wedge z, \qquad x- : x \wedge \neg z$$
$$y+ : \neg y \wedge z, \qquad y- : y \wedge \neg z$$

The following shows $conj(s)$ for each state of the SG in Fig.6(b) also on wires x, y, and z.

$$s_0 : \neg x \wedge \neg y \wedge \neg z, \quad s_2 : x \wedge y \wedge \neg z, \quad s_4 : x \wedge y \wedge z$$
$$s_6 : x \wedge \neg y \wedge z, \quad s_7 : x \wedge \neg y \wedge \neg z, \quad s_9 : x \wedge \neg y \wedge \neg z$$
$$s_{10} : \neg x \wedge y \wedge \neg z, \quad s_{12} : \neg x \wedge y \wedge z, \quad s_{13} : \neg x \wedge y \wedge \neg z$$
$$s_{14} : \neg x \wedge \neg y \wedge \neg z, \quad s_{16} : \neg x \wedge \neg y \wedge z,$$

The output of M_2 is z which is the input of M_1. Similarly, the constraint C_2^0 for $z+$ and $z-$ can be derived, e.g. $\mathcal{C}_2^0(z+) = conj(s_0) \vee conj(s_9) \vee conj(s_{13}) \vee conj(s_{14}) = (\neg x \vee \neg y) \wedge \neg z$, $\mathcal{C}_2^0(z-) = conj(s_4) = x \wedge y \wedge z$.

According to C_2^0, $z+$ is enabled when either x or y is low, and $z-$ is enabled when both x and y are high. Applying C_2^0 to the SG in Fig.6(a) removes the following state transitions.

$$(s_6, z+, \pi) \quad (s_3, z-, \pi) \quad (s_4, z-, \pi) \quad (s_5, z-, \pi)$$

According to C_1^0, $x+$ and $y+$ are enabled only in states where z is high. Applying C_1^0 to the SG in Fig.6(b) makes the following transitions among others in the SG in Fig.6(b) invalid:

$$(s_0, x+, s_9) \quad (s_0, y+, s_{13})$$

Removing these transitions makes states s_9 and s_{13} unreachable. After removing the unreachable states and their outgoing state transitions, the SG in Fig.6(b) is reduced, and a stronger constraint can be derived in the next iteration. After refinement is done, the SGs in Fig.6(a) and (b) are reduced to the ones in Fig.4(a) and (b), respectively, which are failure free.

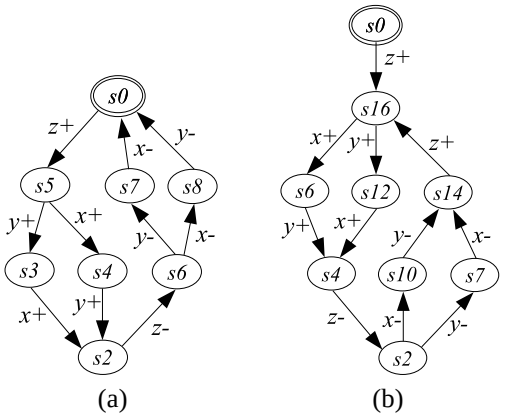

Fig. 4. (a) The SG in Fig.3(b) after refinement. (b) The SG in Fig.3(c) after refinement.

VI. State Space Reduction

This section introduces several techniques to reduce SGs without affecting verification results. Reducing the intermediate SGs during compositional verification controls the size of these SGs, thus allowing larger systems to be verified and more freedom in partitioning. They may also allow stronger interface constraints to be derived, which is desirable for interface refinement and verification. Finally, all the techniques, unlike other conservative approaches, do no introduce extra behavior including extra failures. This is also highly desirable in reducing computation cost needed to confirm the uncovered failures. These techniques are used in the compositional verification framework shown in Algorithm 1.

A. Autofailure Reduction

One technique, autofailure reduction, is based on the following observation. The failure state of a design may be entered by an action on an output or an internal action. However, the real cause of the failure can be traced back to an input action. This is because if an environment produces an input action that a system cannot handle, then the failure happens immediately or through a sequence of internal or output actions, and the environment cannot prevent it from eventually happening. This is referred to as *autofailure manifestation* in [23]. However, autofailure manifestation in [23] is only used to canonicalize trace structures for hierarchical verification. We adopt it in our method as a technique to reduce SGs.

Let $\rho = (s_0, t_0, s_1, t_1, s_2, \cdots, \pi)$ be a failure path in G. Recall that an autonomous path is independent of input actions. If a failure path of a system is autonomous, the failure is inherent in the system, and occurs no matter how the environment behaves. Autofailure reduction reduces a SG containing an autonomous failure path starting from the initial state $init$ to the one consisting of only a single failure state. If ρ is not autonomous, autofailure reduction searches for the largest index i such that action a_i is an input action, and $(s_{i+1}, a_{i+1}, s_{i+2}, \cdots, \pi)$ is an autonomous subpath of ρ. All state transitions on that autonomous subpath are removed, and s_{i+1} is converted to the failure state π. Notice that the removed state transitions on the autonomous subpath may be on the output actions. Therefore, autofailure reduction may strengthen the output constraints for a SG.

Let `autofailure`(G) be a procedure for autofailure reduction as shown in Algorithm 5. Lemma 6.1 shows that autofailure reduction preserves all possible traces of a SG.

Algorithm 5: `autofailure`(G)

```
1  s_2 = π;
2  foreach (s_1, a_1, s_2) ∈ R ∧ s_1 ≠ π do
3      if s_1 = init ∧ a_1 ∉ A^I then
4          return "G has a failure";
5      if a_1 ∉ A^I then
6          delete (s_1, a_1, s_2) ;
7          s_2 = s_1;
8      else
9          replace (s_1, a_1, s_2) with (s_1, a_1, π);
10         s_2 = π;
11 Remove unreachable states and transitions from G;
```

Lemma 6.1: Given a SG G, $G \preceq$ `autofailure`(G).

Proof: If no failure trace exists in G, the procedure of autofailure reduction does nothing. Therefore, $G \preceq$ `autofailure`(G).

Next, we consider G that contains failure paths. Let $\rho = (s_0, t_0, s_1, t_1, s_2, \cdots, \pi)$ be a failure path in G. Suppose ρ becomes $\rho' = (s_0, a_0, \cdots, s_i, a_i, \pi)$ after autofailure reduction. According to Definition 3.5 in section III-B, $\rho \sim_F \rho'$. The above discussion indicates that every failure trace in G is reduced to an equivalent failure in `autofailure`(G). For each non-failure trace in G, it either has a corresponding equivalent failure trace in `autofailure`(G), or simply exists in `autofailure`(G) if it does not have the prefix of any failure trace in `autofailure`(G). ∎

Refer to the SG in Fig.3(b). The state transition $t_1 = (s_{15}, u-, \pi)$ is on an invisible action $u-$. Both incoming state transtions $t_2 = (s_8, x-, s_{15})$ and $t_3 = (s_{11}, y-, s_{15})$ are on input actions $x-$ and $y-$, respectively. Autofailure reduction removes t_1, and changes t_2 and t_3 to $(s_8, x-, \pi)$ and $(s_{11}, y-, \pi)$, respectively. The operation is also applied to $(s_5, u+, \pi)$. After these operations, s_5 and s_{15} become unreachable, thus are removed. The reduced SG is shown in Fig.5(a).

B. Interface Abstraction

Given a module, some of its outputs may bcome invisible to its neighbors when it is plugged into a larger system. In this case, the corresponding state transitions on these outputs in its SG can be converted to invisible transitions. The traditional abstraction techniques collapse the invisible state transitions into single states [11]. This may cause extra behaviors and thus may introduce false failures. This section provides a different abstraction technique that compresses a sequence of invisible state transitions into a single visible state transition. This technique has certain desirable features over the previous approaches.

Let $(s_i, \zeta, s_{i+1}, \zeta, \cdots, s_{j-1}, \zeta, s_j, a_j, s_{j+1})$ be a subpath of a path in a SG G. After abstraction, the whole subpath is

replaced with state transition (s_i, a_j, s_{j+1}). This abstraction is different from the previous approaches in the following ways.

1) Since the sequence of invisible state transitions on a path is replaced by a visible state transition, the number of reachable states of `abstract`(G) may be reduced if some states have all their incoming state transitions on the invisible action. However, this may not always be the case, and the number of state transitions may be increased significantly.
2) This abstraction shortens the existing paths, but no new paths are created. Therefore, no new failure traces are introduced.
3) Nondeterminism may be introduced into a SG after abstraction. Consider two subpaths $(s_i,\zeta,\cdots, s_{j-1},\zeta,s_j,a_j,s_{j+1})$ and $(s_i,\zeta,\cdots,s_{k-1},\zeta,s_k,a_j,s_{k+1})$. They are reduced to (s_i, a_j, s_{j+1}) and (s_i, a_j, s_{k+1}), respectively. This causes nondeterminism even though the original SG is deterministic. However, the nondeterministic transitions do not affect the constraint extraction, and they may be eliminated if s_{j+1} or s_{k+1} is redundant as described in the next section.

Let `abstract`(G) be a procedure for the interface abstraction on a SG G as shown in Algorithm 6. The following lemma asserts that `abstract`(G) is an abstraction of G.

Lemma 6.2: Given a SG G, $G \preceq$ `abstract`(G).

Proof: It is straightforward to see that for every path ρ in G, there exists a path ρ' in `abstract`(G) such that $\rho \sim \rho'$. This satisfies the conditions of conformance relations, and completes the proof. ∎

The SG produced by `abstract`(G) in Algorithm 6 inherits every element of G except the updated R and S. In the algorithm, T and V store all visible state transitions and their states, respectively. The algorithm searches backwards from each visible state transition, and bypasses all the invisible state transitions along a path until another visible state transition is found or the initial state is reached. From these two state transitions, a new transition is created to replace the sequences of invisible state transitions and it is added into T. During the backward search, the invisible state transitions and states with both incoming and outgoing state transitions invisible are not added into T and V. After all state transitions have been handled, R and S of G are replaced with T and V, respectively.

Algorithm 6: `abstract` (G)

```
1  T = ∅, V = ∅;
2  foreach (s2, a2, s1) ∈ R ∧ (a2 ≠ ζ) ∧ s2 ≠ π do
3      foreach (s3, a3, s2) ∈ R ∧ s3 ≠ π do
4          if s2 = init ∨ a3 ≠ ζ then
5              T = T ∪ { (s2, a2, s1)};
6              V = V ∪ {s2, s1};
7          if  a3 = ζ then
8              s2 = s3;
9  replace R with T;
10 replace S with V;
11 Remove unreachable states and state transitions from G;
```

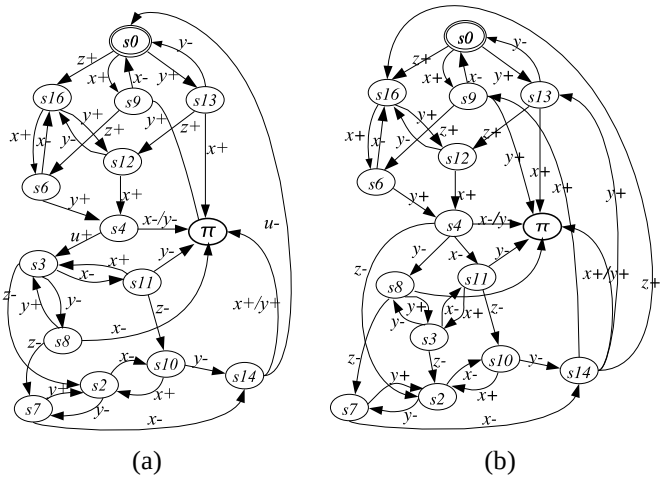

Fig. 5. (a) The SG from Fig.3(a) after the autofailure reduction . (b) The SG from (a) after the interface abstraction.

Refer to Fig. 5(a). State transition $(s_4, u+, s_3)$ is invisible while $(s_3, z-, s_2)$, $(s_3, x-, s_{11})$, and $(s_3, y-, s_8)$ are visible. The algorithm checks that a transition ending at s_4, $(s_6, y+, s4)$, is visible, and three new transitions $(s_4, z-, s_2)$, $(s_4, y-, s_8)$, and $(s_4, x-, s_{11})$ are added. Since the other incoming transition to s_4, $(s_{12}, x+, s_4)$, is visible too, the backward search stops, and the invisible transition $(s_4, u+, s_3)$ is removed. Now, nondetermination takes place at s_4. The same operation is similarly applied to $(s_{14}, u-, s_0)$. The abstracted SG is shown in Fig.5(b).

C. Redundancy Removal

Recall that the procedure for the interface abstraction potentially introduces nondeterminism. A nondeterministic SG can be determinized with some well-known but very expensive algorithms [11]. However, nondeterminism does not affect the soundness of the verification results in our framework. Therefore, we propose a light-weight algorithm instead that targets on removing redundant state transitions and states.

Let $incoming(s)$ be the set of state transitions (s', a, s) such that $R(s', a, s)$ holds, and $outgoing(s)$ be the set of state transitions (s, a, s') such that $R(s, a, s')$ holds.

Definition 6.1 (Redundant States): Let G be a SG, and $s, s_1, s_1' \in S$ such that $s_1 \neq \pi$ and $s_1 \neq init$. We say that s_1 is redundant to s_1', denoted as s_1'/s_1, if there exists a $(s, a, s_1') \in incoming(s_1')$ for each $(s, a, s_1) \in incoming(s_1)$.

Redundant state s_1 and its incoming and outgoing transitions can be removed as follows.

- Remove all state transitions in $incoming(s_1)$ and $outgoing(s_1)$.
- For each $(s_1, a_1, s_2) \in outgoing(s_1)$, add (s_1', a_1, s_2) into R.

Therefore, removing redundant states always results in a smaller number of states and state transitions.

If the failure state is involved in nondeterminism, redundant state transitions are identified based on the following understanding: if an action in a state may or may not cause a failure nondeterministically, it is always regarded as causing a failure.

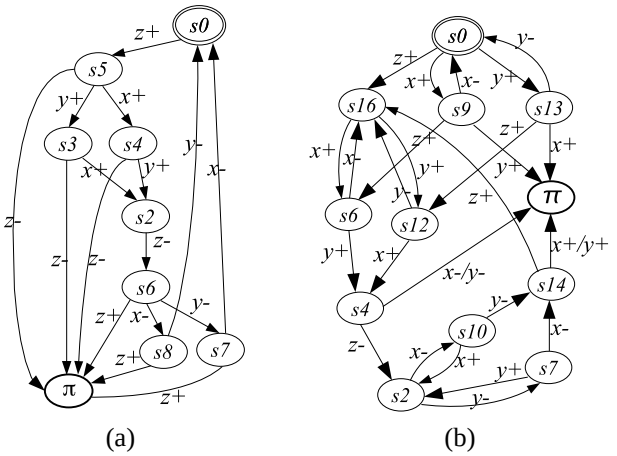

Fig. 6. (a) The SG of M_1 from in Fig.5(a) after and redundancy removal. (b) The SG from Fig.5(b) after the the redundancy removal.

It is formalized as failure equivalent state transitions in the following definition. The failure equivalent transitions do not have any impact on the behavior represented by a SG, and can simply be removed.

Definition 6.2 (Failure Equivalent Transitions): Given two state transitions (s, a, s_1) and (s, a, π) of a SG, (s, a, s_1) is failure equivalent to (s, a, π).

In general, removal of redundant states and state transitions does not help to generate stronger constraints for better refinement. However, if the failure state is involved in redundancy removal, it is possible that the removal of the failure equivalent transitions results in stronger constraints.

Let $\texttt{rmRed}(G)$ be a procedure to generate a new SG by removing the redundancies in G as shown in Algorithm 7. The following lemma states that the resultant SG conforms to the original SG.

Lemma 6.3: Given a SG G, $G \preceq \texttt{rmRed}(G)$.

Proof: We consider two cases.

- Case 1: s'/s where s' and s are not the failure state. For any path $\rho = (\cdots, s_i, a_i, s, a_{i+1}, \cdots)$ in G, there exists a path $\rho' = (\cdots, s_i, a_i, s', a_{i+1}, \cdots)$ in $\texttt{rmRed}(G)$ such that $\rho \sim \rho'$.
- Case 2: For any path $\rho = (\cdots, s_i, a_i, s', \cdots)$ in G and $(s_i, a_i, \pi) \in R$, there exists $\rho' = (\cdots, s_i, a_i, \pi, \cdots)$ such that $\rho \sim_F \rho'$. This indicates that ρ' is in $\texttt{rmRed}$ while ρ is not.

From case 1 and 2, it is concluded that $G \preceq \texttt{rmRed}(G)$. ∎

Refer to the SG in Fig. 5(b). $(s_4, y-, s_8)$ is failure equivalent to $(s_4, y-, \pi)$ and it is removed. Similarly, $(s_4, x-, s_{11}), (s_{14}, x+, s_9)$ and $(s_{14}, y+, s_{13})$ are removed as failure equivalent transitions. After removing unreachable states s_3, s_8, and s_{11}, the reduced SG is shown in Fig.6(b).

Algorithm 7: $\texttt{rmRed}\ (G)$

```
1  foreach (s, a, π) ∈ R do
2      if (s, a, s1) ∈ R then
3          R = R − {(s, a, s1)} ;
4  S' = ∅;
5  foreach s'1 ∈ S do
6      foreach s1 ∈ S do
7          if s'1/s1 then
8              R = R − {incoming(s1) ∪ outgoing(s1)};
9              S = S − {s1};
10             foreach (s1, a1, s2) ∈ outgoing(s1) do
11                 R = R ∪ {(s'1, a1, s2)};
12 Remove unreachable states and state transitions from G;
```

VII. Experimental Results

A prototype has been implemented for the method and algorithms presented in this paper in an asynchronous system verification tool FLARE, and experiments have been performed on several asynchronous circuits. These experiments aim to show the scalability of verification using our compositional method.

FLARE is an explicit model checker for asynchronous circuit and system verification. It can perform flat and compositional verification. Its closest relative is ATACS [36]. Although ATACS also supports compositional verification and failure-directed abstraction, the abstraction is mainly limited to a certain type of high-level modeling formalism. More significantly, the automated interface refinement supported in FLARE is not available in ATACS. There are other tools supporting compositional verification, but we could not find another one that supports asynchronous system verification in a way similar to our tool.

A. Examples and Environment Setup

The first three designs used in our experiments are a self-timed FIFO [32], a tree arbiter of N cells [23], and a distributed mutual exclusion element consisting of a ring of DME cells [23]. Although all these designs have a regular structure 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 from Intel's RAPPID asynchronous instruction length decoder [38]. This example is an unoptimized version of the actual circuit used in RAPPID with higher complexity, which is more interesting for experimenting with our methods. The last example is a pipeline controller for an asynchronous processor TITAC2 [39]. All five examples are failure free. Note that all the examples are too large to apply flat approaches.

In the experiments, DME, arbiter, and FIFO examples are partitioned according to their natural structures. In other words, each cell is a module. For the tag unit circuit, it is partitioned into three modules, where the middle five blocks form a module, and gates on the sides of the middle module form the other two modules. The pipeline controller is partitioned into 10 modules, each of which contains five gates. All results are obtained on a Linux workstation with an Intel Pentium-D dual-core CPU and 1 GB memory. In the results tables, time is in seconds, and memory is in MBs.

According to the compositional verification shown in section IV, it is necessary to generate an over-approximate

TABLE I
EXPERIMENTAL RESULTS (REFINEMENT + REDUCTION + COMPOSITION).

System	$\#Cells$	$\|W\|$	$Mem(MB)$	$Time(Sec.)$	$\|S\|^{peak}$	$\|R\|^{peak}$	$\|S\|$	$\|R\|$	$\#Iter$	$\#\pi$
FIFO	100	402	30	18	57	153	15	21	3	0
	200	802	80	41	57	153	15	21	3	0
	400	1602	237	102	57	153	15	21	3	0
	600	2402	471	184	57	153	15	21	3	0
	800	3202	781	290	57	153	15	21	3	0
DME	20	220	35	43	361	9497	33	61	4	0
	50	550	88	113	361	9497	33	61	4	0
	100	1100	191	249	361	9497	33	61	4	0
	200	2200	446	600	361	9497	33	61	4	0
	300	3300	771	1044	361	9497	33	61	4	0
ARB	7	66	3	2	290	758	38	68	3	0
	15	122	7	6	385	1407	104	340	5	0
	31	250	33	47	1737	9636	689	3833	7	0
	63	506	262	988	15684	134438	5018	48342	8	0
TAGUNIT	3	48	117	103	3697	280392	2112	122648	2	0
PIPECTRL	10	50	23	47	1409	12736	679	4882	6	4

environment for each module in a system to simulate the actual environment. In the experiments, *maximal environment* is used as the initial approximate environment. The concept of the maximal environment introduced in [27] is adapted for the modeling formalism used in our tool FLARE. The maximal environment defines all possible behaviors on all inputs driving a system. The behaviors of each input is completely independent to those of other inputs, and updating the values of inputs is interleaved in all possible orderings. The constraints extracted for the inputs of a module from the maximal environment are all TRUE, since the behaviors of inputs are completely unconstrained.

The maximal environment is used as the worst case scenario to demonstrate the effectiveness of the presented method to refine a very coarse approximation. If better knowledge of the internal details about a design is available, a more accurate initial environment approximation can be obtained and used instead of the maximal environment, thus speeding up the refinement process.

B. Results

In the first experiment, all examples are verified with the compositional method combined with the interface refinement and all state space reduction techniques presented in this paper. The results are shown in Table I. In the table, column $\#Cells$ is the number of cells in a system, and column $|W|$ is the total number of wires in each system, which is a rough estimate of the system complexity. Column $Mem(MB)$ and $Time(Sec.)$ are the peak memory used and the total runtime taken for verifying each system. The numbers of states and state transitions of the single largest SG for a module found during the procedure of performing compositional approach, which serve as a critical metric for the effectiveness, are shown in columns $|S|^{peak}$ and $|R|^{peak}$, respectively. The next two columns $|S|$ and $|R|$ show the numbers of states and state transitions in the largest SG after the refinement is done. Column $\#Iter$ shows the number of iterations required to complete the interface refinement, and the last column $\#\pi$ shows the number of modules in a system that have the failure state after the refinement.

In these experiments, selective composition is performed if SGs of some modules are found to have failures. In selective composition, some or all of the SGs with failures are composed as allowed by the available memory. This results in some state transitions becoming invisible and leads to reductions on the composed SGs. Then, the refinement is applied again. The reduced composite SGs in turn may result in reductions on other SGs. The number of iterations in Table I is the total number of iterations for the refinement before and after selective composition. In the experiments, selective composition is needed for ARB and PIPECTRL. For PIPECTRL, the original 10 modules all contain failures. Selective composition merges these modules into 5 larger ones, and one of them is failure-free.

From Table I, we obtain the following observations. First, for scalable FIFO, ARB, and DME examples, memory and runtime usages grow polynomially as the number of modules in the systems increases. Second, although the refined SG for each module is still an abstraction of the exact one after the interface refinement, all examples except PIPECTRL are shown to be failure-free. Even though interface refinement may not eliminate failures for each module in PIPECTRL completely, the number of modules containing failures and the number of failure traces in those modules are reduced significantly by the refinement and reductions, therefore making distinguishing false counterexamples easier. In the table, we only show the size of the largest SG encountered during verification. This is because it is the largest module of a system that determines the success or failure of verification. With respect to compositional verification, only one module needs to stay in memory at a time. However, our method keeps SGs of all modules in memory for simplicity, and the memory numbers in Column $Mem(MB)$ show the total peak memory usage for all modules during verification.

To fully appreciate the power of the introduced reduction techniques, the difference between the traditional abstraction and the one proposed in this paper, and the impact of selective composition on the verification results, we run three more experiments. All these experiments use the compositional verification approach presented in this paper without using

TABLE II
COMPARISON OF THREE DIFFERENT EXPERIMENTS.

System	$\#Cells$	$E1$			$E2$			$E3$		
		$Mem(MB)$	$Time(Sec.)$	$\#\pi$	$Mem(MB)$	$Time(Sec.)$	$\#\pi$	$Mem(MB)$	$Time(Sec.)$	$\#\pi$
FIFO	100	30	11	0	29	11	96	30	18	0
	200	80	28	0	78	26	196	80	41	0
	400	236	74	0	232	66	396	237	102	0
	600	470	140	0	470	140	596	471	184	0
	800	780	227	0	772	201	796	781	290	0
DME	20	14	13	0	13	9	0	35	43	0
	50	40	37	0	38	27	0	88	113	0
	100	97	90	0	94	69	0	191	249	0
	200	264	249	0	258	202	0	446	600	0
	300	502	474	0	492	402	0	771	1044	0
ARB	7	3	1	6	2	1	2	3	2	0
	15	8	3	15	6	4	3	7	5	7
	31	18	8	30	15	9	12	17	13	15
	63	42	22	59	36	23	30	40	33	31
TAGUNIT	3	26	28	0	11	10	0	117	103	0
PIPECTRL	5	13	10	5	8	10	4	19	50	5

selective SG composition as in the previous experiment. The results are shown in Table II. In the first experiment labeled as $E1$, all examples are verified with only the interface refinement. In the second experiment labeled as $E2$, all examples are verified with the interface refinement and all the reduction techniques, except that the traditional state space abstraction is used instead of the interface abstraction presented in this paper. In the third experiment labeled as $E3$, all reduction techniques including the interface abstraction plus the interface refinement are used for all examples. This experiment is similar to the one used for Table I except that selective composition is not applied. Comparing the results in these two tables, the runtime and memory usage for all examples as shown in Table II are generally less, much less in some cases, than those shown in Table I. The main reason is the SG composition, which causes the size blowup if used for some examples. However, without selective composition, the verification results become worse in terms of the number of modules with failures. For example, 31 out of 63 modules in the ARB have failures under $E3$ without using the SG composition, while none has failures if selective composition is applied as shown in Table I.

Comparing the results in columns $E1$ and $E3$, whether applying reductions or not does not make much difference in these experiments except for the TAGUNIT. In this case, memory blows up because interface abstraction creates a much large number of state transitions to preserve all possible behaviors of the modules in TAGUNIT and avoid introducing extra paths. The increased state transitions may be removed by performing redundancy removal or autofailure reduction for the examples exclusive of TAGUNIT. This example illustrates the negative effect of the interface abstraction technique. On the positive side, using the new abstraction causes less failures introduced. Comparing results in columns under $E2$ and $E3$, no module has failures when interface abstraction is used, while the number of modules with failures increases as the number of cells increases for the FIFO. However, for ARB and PIPECTRL, the number of modules with failures under $E2$ is actually smaller than that under $E3$. This is because of the aggressive reduction feature of the traditional abstraction where larger state space may be trimmed as long as the failure traces are preserved. Trimming a larger state space, including the valid portion, helps to produce stronger output constraints, which then cause the other modules to be more reduced. This is why the memory and runtime are also less under $E2$. Using traditional abstraction always preserves failures in at least one module, but its aggressiveness in reducing state space may cause some modules to lose their failure traces.

VIII. Conclusion

While compositional verification is an effective approach to attack state explosion in model checking, generating accurate yet simple environments for system modules poses a big challenge. This paper proposes an interface refinement algorithm for compositional verification where the module interfaces are monotonically refined. This method is fully automated and sound as long as the initial environment for each module in a system is over-approximated. This allows very coarse environments to be used when verification starts. In addition, several state space reduction techniques are introduced, and they may help remove irrelevant behavior, thus making interface refinement more effective. The initial experimental results are encouraging. This refinement algorithm is general as long as the designs can be modeled using the SG formalism presented in this paper, and it can be combined with other compositional verification approaches. In the future, it would be interesting to investigate other representations of constraints for more effective refinement, and efficient partitioning strategies for better verification.

References

[1] R. Alur, L. de Alfaro, T. Henzinger, and F. Mang. Automating modular verification. In *Proceedings of the 10th International Conference on Concurrency Theory*, LNCS, pages 82–97. Springer-Verlag, 1999.

[2] 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.

[3] S. Berezin, S. Campos, and E. Clarke. Compositional reasoning in model checking. In *COMPOS*, volume 1536 of *LNCS*, pages 81–102. Springer-Verlag, Sept. 1998.

[4] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking, 2003.
[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.
[6] A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
[7] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, 1994.
[8] D. Bustan and O. Grumberg. Modular minimization of deterministic finite-state machines. In *Proceedings the 6th International workshop on Formal Methods for Industrial Critical Systems (FMICS'01)*, July 2001.
[9] S. Chaki, E. Clarke, J. Ouaknine, and N. Sharygina. Automated, compositional and iterative deadlock detection. In *MEMOCODE 2004*, pages 201–210, 2004.
[10] 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.
[11] S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, 1996.
[12] S. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.
[13] 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.
[14] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
[15] 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.
[16] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.
[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Intl. Conf. on Computer Aided Verification*, pages 154–169, 2000.
[18] J. Cobleigh, G. Avrunin, and L. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 97–108, New York, NY, USA, 2006. ACM Press.
[19] 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.
[20] L. de Alfaro and T. Henzinger. Interface automata. *Foundations of Software Engineering*, pages 109–120, 2001.
[21] L. de Alfaro and T. henzinger. Interface theories for component-based design. In *Proceedings of the 1st International Workshop on Embedded Software*, pages 148–165, Oct 2001.
[22] L. de Alfaro and T. Henzinger. Interface-based design. *Engineering Theories of Software-intensive Systems*, 195:83–104, 2005.
[23] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.
[24] M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. Refining interface alphabets for compositional verification. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS. Springer-Verlag, 2007.
[25] D. Giannakopoulou, C.S.Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, pages 297–320, 2005.
[26] S. Graf, B. Steffen, and G. Luttgen. Compositional minimization of finite state systems using interface specifications. *Formal Aspects of Computation*, 8(5):607–616, 1996.
[27] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
[28] T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. Int. Conf. on Computer Aided Verification*, volume 2725 of *LNCS*, pages 262–274. Springer-Verlag, 2003.
[29] 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.
[30] 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.
[31] F. Lang. Refined interface for compositional verification. In *FORTE'06: Formal Techniques for Networked and Distributed Systems*, volume 4229 of *LNCS*. Springer Verlag, 2006.
[32] 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.
[33] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1993.
[34] K. L. Mcmillan. A methodology for hardware verification using compositional model checking. Technical report, Cadence Berkeley Labs, 1999.
[35] C. J. Myers. *Asynchronous Circuit Design*. Wiley Inter-Science, 2001.
[36] C. J. 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, 2001.
[37] W. Nam and R. Alur. Learningcbased symbolic assume-guarantee reasoning with automatic decomposition. In *Proc. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4218 of *LNCS*, 2006.
[38] 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.
[39] 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.

PLACE PHOTO HERE

Haiqiong Yao Haiqiong Yao received a B.S in computer science from Yunnan University, Kunming, China, in 1997, and a M.S degree in computer science from PLA University of Science and Technology, Nanjing, China, in 2004. She is currently pursuing a Ph.D. degree at the University of South Florida.

Her main research interest is formal methods for software and hardware verification, primarily in the area of compositional reasoning, abstraction and reduction techniques for model checking.

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.