

Modular Model Checking of Large Asynchronous Designs with Efficient Abstraction Refinement

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

Abstract—Divide-and-conquer is essential to address state explosion in model checking. Verifying each individual component in a system in isolation efficiently requires an appropriate context, which traditionally is obtained by hand. This paper presents an efficient modular model checking approach for asynchronous design verification. It is equipped with a novel abstraction refinement method that can refine a component abstraction to be accurate enough for successful verification. It is fully automated, and eliminates the need of finding an accurate context when verifying each individual component, although such a context is still highly desirable. This method is also enhanced with additional state space reduction techniques. The experiments on several non-trivial asynchronous designs show that this method efficiently removes impossible behaviors from each component including ones violating correctness requirements.

Index Terms—formal methods, model checking, modular verification, logic verification, circuit verification, abstraction, refinement

1 INTRODUCTION

Compositional methods are essential to address the state-explosion problem [10] by verifying the individual components without considering the whole system. Effectiveness of these methods depends on if an over-approximated and yet accurate context can be found for each component such that all the essential behavior of that component can be checked. If such a context is not sufficiently accurate, a large number of false counter-examples may be produced during verification, which may lead to high computation penalty to distinguish them from the real ones. Traditionally, such contexts are obtained by hand, which is very error prone and tedious. Furthermore, accurate contexts are very difficult to obtain if the component interfaces are complex.

To address this problem, this paper proposes a novel method where such a context does not need to be found explicitly. Instead, component abstractions obtained with over-approximate environments are refined iteratively such that behaviors not allowed on the interface among components are removed. This abstraction refinement method is based on the observation that the interface behaviors are allowed if they are synchronized among the components connected by that interface. Consider a system $M = M_0 \parallel \dots \parallel M_n$. When composing two components $M_i \parallel M_j$, only the synchronized behaviors in M_i and M_j appear in the composition, while the unsynchronized ones are removed. This method borrows the idea of parallel composition and refines M_i and M_j by checking their synchronized interface behaviors with respect to those allowed by $M_i \parallel M_j$. The key to this

method is to identify and remove the unsynchronized behaviors in each component without actually constructing the composition.

Traditional parallel composition is defined on two components, therefore synchronization-based refinement can only refine two components at a time. This is not sufficient in practice since a component may interact with multiple neighbors, which themselves may have interactions with each other. The interface interactions among the neighbors may have an effect on how the component interacts with its neighbors. If these interactions are not considered, extra behaviors in the component may not be removed causing a large number of false counter-examples. Therefore, the above refinement approach is enhanced by extending synchronization to more than two components at a time to take inter-dependencies among components into account for stronger refinement.

Ideally, the best result can be obtained if synchronization is applied to all components in M in one step. However, the complexity of this approach may be as high as that of $M_0 \parallel \dots \parallel M_n$. Therefore, it can only be applied locally to a subset of components at a time where they share some common interfaces. In each iteration of the refinement process, components in each subset are refined together. It is possible that components M_i or M_j exists in different subsets, and they can be further refined with other components after they are refined with each other. Therefore, the above step needs to be repeated to achieve stronger refinement results. This method guarantees to terminate when the violating behaviors in each component are eliminated by refinement, or the components cannot be refined further.

Since this method starts with component abstractions which may contain a large amount of impossible behavior which may be removed later on, the complexity at the beginning of the verification process could be very

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

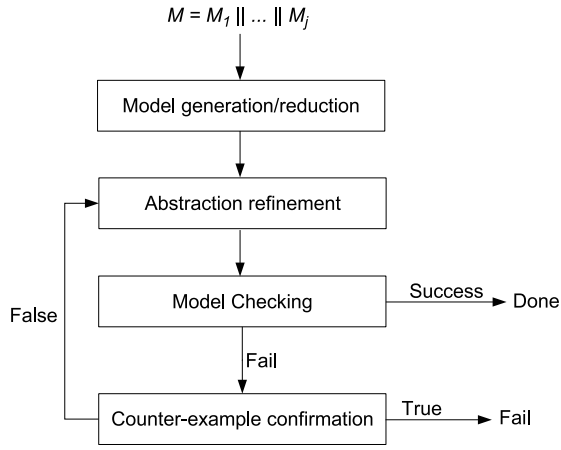


Fig. 1. The flow of modular model checking with abstraction refinement.

high. To address this problem, two additional reduction techniques are used to control the complexity of the initial component abstractions and that of the intermediate results by removing irrelevant behavior from the beginning and whenever possible during the entire refinement process. By removing the irrelevant behavior, these techniques help refinement to eliminate more false counter-examples found during verification as well as speed up the whole process.

Fig.1 shows the verification flow used in our method. In general, designs are described in some high level specification formalism. Given a design, it is first translated such that a finite state model is generated for each component in a design with an over-approximate environment. Next, these models are refined and verified using the method presented in this paper. If counter-examples exist in any component, it is necessary to determine if they are real. This step is performed on the entire design by a guided-simulation assisted with partial-order reduction as in [38]. If a counter-example is real, users are notified. Otherwise, the cause of such a false counter-example is extracted and used to refine all components further. If all components are verified without any counter-example, the entire design is claimed to be correct.

The main contributions of this paper are a local synchronization detection method for component refinement based on parallel composition, and an abstraction refinement method for efficient modular model checking where each component is refined iteratively and incrementally by checking synchronization among components sharing common interfaces. To our best knowledge, we cannot find any other compositional methods similar to the one presented in this paper. Although this method is presented in the context of asynchronous designs, we believe the concepts behind this method are general, and can be used in other application domains.

This paper is organized as follows. The next section reviews previous related work. Section 3 presents some

preliminary background necessary to understand the topics in the later sections. Section 4 is the core of this paper, and presents the abstraction refinement framework with a novel interface synchronization method. Section 5 describes the reduction techniques used in this method, and Section 6 shows the experimental results of the presented method on several large asynchronous designs. The last section concludes the paper, and points out some possible research directions in the future.

2 RELATED WORK

Compositional verification is essential to verifying large systems. It can be roughly classified as *compositional minimization* and *compositional reasoning*. Compositional minimization [5], [22], [27] 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. In [32], a modular verification approach with failure-directed abstraction is presented. However, abstraction in that approach is done by Petri-net reductions on a certain type of Petri-nets. If a system is represented by a different kind of Petri-nets or other formalisms, verification complexity may not be reduced significantly. More importantly, it does not support abstraction refinement such as the one described in this paper.

On the other hand, compositional verification based on *assume-guarantee* style reasoning [34], [12], [23], [3], [25], [30] does not construct the global model. Instead, verification of a system is broken into separate analyses for each component of the system. The result for the entire system is derived from the results of verifying individual components. When verifying each component, abstractions or assumptions about the environments with which the components 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 component. This is typically done by hand. If the components 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 [33], [2], [15], [21], [7]. 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 [33], [2].

Comparatively, this method has several significant differences from the learning based ones. First, the interface

behavior of a component is refined by iteratively examining the interactions between the component 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 component interfaces. Not using counter-examples for refinement allows more freedom in system partitioning, while existing learning based methods seem more suitable only for two-way partitions as discussed in [33], [14]. Second, modular verification based on our method does not require complex reasoning rules, and circular structures in a design can be handled without difficulty. Third and probably more importantly, there are no assumptions generated in this method, therefore there is no need for the assumption discharging step. Despite the differences, this method and the learning-based methods can be combined to achieve better results.

In the *interface constraint-based* approaches, restrictions from the 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 [8]. 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 [16], [17], [18]. 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] [20], [28]. A similar approach, *thread-modular reasoning*, is proposed in [24] for multi-threaded program verification.

Counterexample guided abstraction refinement (CEGAR) [13], [11], [9] 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 component in a system, instead of a global model of the entire system. CEGAR is first coupled with compositional verification in [6].

[36] presents an alternative abstraction refinement approach. However, this work has several important differences. First, the approach in [36] derives interface invariants from component models which are used to reduce other components subsequently. These invariants represent necessary conditions for input behavior that must be satisfied in all states. These invariants are often inad-

equate if the interface interactions among components are complex. On the other hand, the synchronization-based refinement in this paper does not use interface invariants for refinement, and can remove invalid interface behaviors more effectively. Second, *autofailure reduction*, a technique to be described in a later section, is combined with reachability analysis in this paper to contain the sizes of component models generated at the beginning of the verification flow, while in [36] it is only applied after the models are initially generated. This on-the-fly reduction avoids generating a large number of redundant behaviors in the first place, therefore leading to lower model complexity and higher verification efficiency. Third, partial order reduction is supported in this work to reduce the complexity of the intermediate results, while it is not available in [36].

3 PRELIMINARIES

This section describes basic notations and definitions necessary for later sections. Readers are referred to [10], [26] for more detail.

3.1 State Graphs

This paper uses *state graphs* (SGs) to model asynchronous systems. The definition of state graphs is given as follows.

Definition 3.1 (State Graphs): A state graph M is a tuple $(\mathcal{A}, S, R, init)$ where

- 1) $\mathcal{A}$ is a finite set of actions,
- 2) S is a finite non-empty set of states,
- 3) $R \subseteq S \times \mathcal{A} \cup \{\zeta\} \times S$ is the set of state transitions,
- 4) $init \in S$ is the initial state.

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 is a symbol to represent any actions of a component not visible to other components. Since the invisible actions are irrelevant on the interface, ζ is used to represent all invisible actions to make the presentation clear. In the above definition, S includes a special state π which denotes the *failure state* of a SG M , and represents violations of some prescribed properties. Since it does not matter how a system behaves after it enters the failure state, there is a $(\pi, a, \pi) \in R$ for every $a \in \mathcal{A}$. The failure state can broadly represent conditions causing hazards in asynchronous circuits and other safety errors. In the sequel, we also use $R(s, a, s')$ to denote $(s, a, s') \in R$.

Fig.2(a) shows a simple asynchronous circuit for illustration. 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.2(b), (c) and (d) show the corresponding SGs for the components M_1 , M_2 , and M_3

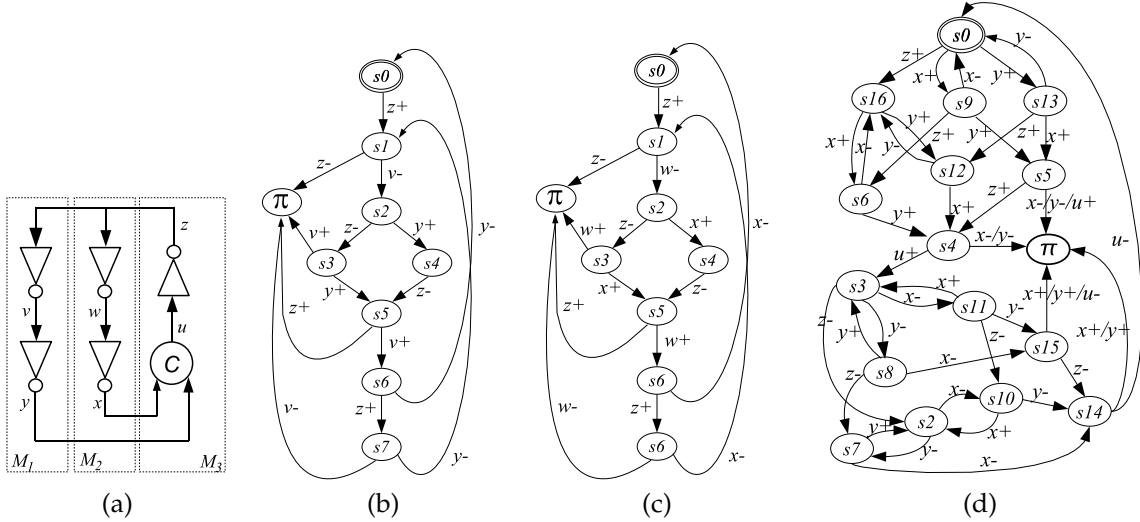


Fig. 2. (a) A simple asynchronous circuit. (b) - (d) The SGs for component M_1 , M_2 , and M_3 where the inputs of the components are set to be completely free.

where their 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_3 , its input actions $\mathcal{A}^I = \{x+, x-, y+, y-\}$, its output actions $\mathcal{A}^O = \{z+, z-\}$, and its invisible action ζ represents $u+$ and $u-$.

A path ρ of M is a sequence of alternating states and actions of M , $\rho = (s_0, a_0, s_1, a_1, s_2, \dots)$ such that $s_i \in S$, $a_i \in \mathcal{A}$, 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 $\mathcal{A}^O \cup \{\zeta\}$. A state $s' \in S$ is *reachable from* a state $s \in S$ if there exists a path $\rho = (s_0, a_0, s_1, a_1, s_2, \dots, s_n)$ 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 $(a_0, a_1, \dots)$. Two traces $\sigma = (a_0, a_1, \dots)$ and $\sigma' = (a'_0, a'_1, \dots)$ 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 $\mathcal{L}(M)$.

Given a SG M , its projection onto $\mathcal{A}' \subseteq \mathcal{A}$ denoted by $M' = M[\mathcal{A}']$, is a SG such that 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. Given a trace $\sigma = (a_0, a_1, \dots)$, 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 $a \notin \mathcal{A}'$ as shown below.

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

where $\sigma' = (a_1, \dots)[\mathcal{A}']$, and $\circ$ is the concatenation operator.

Given two paths, their equivalence is defined as follows.

Definition 3.2: Let $\rho = (s_0, a_0, s_1, a_1, \dots)$ and $\rho' = (s'_0, a'_0, s'_1, a'_1, \dots)$ be two paths 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 similar to that in [4] except that more rules are created for situations involving π .

Definition 3.3 (Parallel Composition of SG): Let

$$M_1 = (\mathcal{A}_1, S_1, R_1, init_1) \text{ and } M_2 = (\mathcal{A}_2, S_2, R_2, init_2)$$

be two SGs. If $\mathcal{A}_1^O \cap \mathcal{A}_2^O = \emptyset$, the parallel composition of M_1 and M_2 is defined as

$$M_1 \parallel M_2 = (\mathcal{A}_1 \cup \mathcal{A}_2, S_1 \times S_2, R, (init_1, init_2))$$

where

- 1) For each $(s_1, s_2) \in S$, the following condition holds

$$(s_1 = \pi \Rightarrow s_2 = \pi) \wedge (s_2 = \pi \Rightarrow s_1 = \pi)$$

- 2) $R \subseteq S \times \mathcal{A} \times S$ 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

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

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

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

- 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

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

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 component state is the failure state. When several components 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.

3.2 Correctness Definition

The failure state π can be used to represent various safety violations. 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 $\rho_1 = (s_0, a_0, \dots, s_i, a_i, \pi, \dots)$ corresponds to a set of traces. Given a failure $\rho = (s_0, a_0, \dots, s_i, a_i, \pi, \dots)$, the non-failure prefix of ρ is $(s_0, a_0, \dots, s_i, a_i)$. 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 3.4: Given a path $\rho = (s_0, a_0, \dots)$, if there exists another path $\rho' = (s'_0, a'_0, \dots)$ such that $\exists j > 0. s'_j = \pi$ and $\forall 0 \leq i \leq j. a_i = a'_i$, then ρ is failure equivalent to ρ' , denoted as $\rho \sim_F \rho'$.

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

Definition 3.5: 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 $\rho \sim_F \rho'$.

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.

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

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

The abstraction relation also satisfies the following property. Let M and M' be two SGs, and $\mathcal{A}$ be the action set of M . Then, the following equation holds.

$$(M \parallel M')[\mathcal{A}] \preceq M \quad (2)$$

Basically, this property states that a component's behaviors are restricted if it is composed with another component. It is useful when proving a theorem in a later section.

4 ABSTRACTION REFINEMENT FOR MODULAR VERIFICATION

This section describes our modular verification method with an abstraction refinement approach. Given a system $M = M_0 \parallel \dots \parallel M_n$, we wish to verify each M_i for $0 \leq i \leq n$ individually due to the obvious difficulty in verifying M . When verifying M_i , its context is the composition of all components in M except M_i projected to $\mathcal{A}_i$. M_i

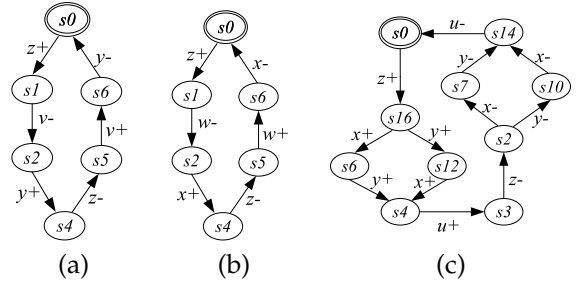


Fig. 3. SGs from Fig. 2(b)-(d) after refinement using interface synchronization. Note that all transitions to the failure state and the failure state itself are removed.

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 M . Therefore, it is necessary to find a M_i^A such that

$$M_i^C \preceq M_i^A \preceq M_i$$

and the complexity of M_i^A is much lower than that of M_i^C . By the definition of the abstraction relation and property (1),

$$\forall 0 \leq i \leq n. \mathcal{F}(M_i^A) = \emptyset \Rightarrow \mathcal{F}(M) = \emptyset$$

This implies that the whole design is correct if an abstraction of every individual component is verified to be correct. In the remainder of this section, a fully automated method is presented to find such M_i^A for $0 \leq i \leq n$.

4.1 Local Synchronization Detection

In a concurrent system, communicating components are synchronized on shared actions. The parallel composition $M = M_1 \parallel M_2$ captures concurrent executions of M_1 and M_2 with synchronization on the alphabet $\mathcal{A}_1 \cap \mathcal{A}_2$. Unsynchronized behavior, which M_2 does not allow M_1 to take place, or vice versa, can be eliminated from the composite SG M . Using the notion of languages of state graphs, the parallel composition $\parallel$ satisfies

$$\mathcal{L}(M)[\mathcal{A}_1] \subseteq \mathcal{L}(M_1) \text{ and } \mathcal{L}(M)[\mathcal{A}_2] \subseteq \mathcal{L}(M_2)$$

The illegal paths such as $\mathcal{L}(M_1) - \mathcal{L}(M)[\mathcal{A}_1]$ and $\mathcal{L}(M_2) - \mathcal{L}(M)[\mathcal{A}_2]$ in M_1 and M_2 , respectively, are removed during parallel composition. In this method, the state transitions that appear on paths in $\mathcal{L}(M)[\mathcal{A}_i]$ where $i = 1, 2$ are referred to as *synchronized* transitions.

Definition 4.1: Given two SGs M_1 and M_2 , a transition (s_i, a, s'_i) in M_i where $i = 1, 2$ is synchronized if there exists a corresponding transition $((s_1, s_2), a, (s'_1, s'_2))$ in $M_1 \parallel M_2$. Otherwise, the transition is not synchronized.

According to the above definition and parallel composition, a transition on an invisible action is automatically

synchronized. A transition on a visible action is synchronized if there is a corresponding transition in another component on the same visible action.

The goal of refinement is to remove from M_i the unsynchronized transitions. Based on the above discussion, the synchronized transitions can be identified during parallel composition. Given M_1 and M_2 , the idea of local synchronization detection is to refine M_1 and M_2 by removing the unsynchronized transitions identified during parallel composition. The above procedure is denoted by $\text{sync}(M_1, M_2)$, which returns two SGs M'_1 and M'_2 such that the following conditions hold for M'_1 .

- 1) $S'_1 = \{s_1 \in S_1 \mid (s_1, s_2) \in S\}$
- 2) $R'_1 = \{(s_1, a, s'_1) \in R_1 \mid ((s_1, s_2), a, (s'_2, s'_2)) \in R\}$

and the similar condition holds for M'_2 too. In the above definition, S and R are the sets of states and state transitions of $M_1 \parallel M_2$. It is clear that $M'_i \preceq M_i$ since M'_i includes states reachable only in the composition and the unsynchronized state transitions in M_i are removed from the resultant SG M'_i , i.e. $\mathcal{L}(M'_i) \subseteq \mathcal{L}(M_i), i = 1, 2$. This indicates that function $\text{sync}()$ can be viewed as a monotonic function in that applying it each time makes the SGs more restricted.

The concept of parallel composition is not new. However, using it as an approach to refinement is novel in this work. Although the composite SG is used in the above description to make presentation clear, that full composite SG is never generated during synchronization to save time and memory. This point becomes clear in Section 4.3.

4.2 Iterative Abstraction Refinement

In the following discussion, the parallel composition $M = M_0 \parallel \dots \parallel M_n$ is also denoted as $M = \{M_0, \dots, M_n\}$. When applying $\text{sync}()$ to every pair of components $\{M_i, M_j\} \subseteq M$ and $\mathcal{A}_{ij} = \mathcal{A}_i \cap \mathcal{A}_j \neq \emptyset$, it reduces M_i and M_j with respect to $\mathcal{A}_{ij}$. Additionally, this reduction can probably render some other transitions in M_i on $\mathcal{A}_i - \mathcal{A}_{ij}$ to become unreachable, causing further reduction on behaviors on the interfaces of M_i with components other than M_j . The same argument also applies to M_j . Since each component may share interfaces with multiple other components, to maximally refine each component, $\text{sync}()$ needs to be applied iteratively until all components cannot be reduced further. The above discussion is formulated as shown in the following framework.

- 1 : $\forall 0 \leq i, j \leq n \wedge i \neq j. (M_i^1, M_j^1) = \text{sync}(M_i, M_j)$
- 2 : $\forall 0 \leq i, j \leq n \wedge i \neq j. (M_i^2, M_j^2) = \text{sync}(M_i^1, M_j^1)$
- ...
- l : $\forall 0 \leq i, j \leq n \wedge i \neq j. (M_i^l, M_j^l) = \text{sync}(M_i^{l-1}, M_j^{l-1})$

In each iteration, function $\text{sync}()$ refines M_i^k and M_j^k such that $(\mathcal{A}_i^O \cap \mathcal{A}_j^I) \cup (\mathcal{A}_i^I \cap \mathcal{A}_j^O) \neq \emptyset$ to be M_i^{k+1} and M_j^{k+1} , respectively. Since function $\text{sync}()$ is monotonic,

$$\forall 0 \leq i, j \leq n \wedge i \neq j. M_i^{k+1} \preceq M_i^k \text{ and } M_j^{k+1} \preceq M_j^k.$$

If M_i also has an interface with another component M_h and $\text{sync}(M_i, M_h)$ is completed before $\text{sync}(M_i, M_j)$, M_i may become more restricted after $\text{sync}(M_i, M_j)$, but this change is not available to M_h until the next iteration. This indicates that through iterations the effect of refinement of one component propagates to all other components gradually. From the above discussion, the above framework does not require complex reasoning rules as in assume-guarantee reasoning approaches.

The process performed by the above iterative refinement framework terminates when either of the following two conditions is satisfied:

- $\forall 0 \leq i \leq n. \mathcal{F}(M_i) = \emptyset.$
- $\forall 0 \leq i \leq n. M_i^l \equiv M_i^{l-1}.$

The first condition says that it is unnecessary to continue refinement if all components are failure free because this implies that the complete system is failure free too. The second condition indicates that continuing the process does not result in any further refinement if all components remain the same after being refined. Since function $\text{sync}()$ monotonically refines components, it is guaranteed that the iterative refinement process eventually terminates. At that point, any counter-examples found in any module are returned to determine if they are real. Identifying real counter-examples itself is very critical because it may become the bottleneck of the whole verification flow. In this method, counter-examples are handled by an approach similar to the one in [38].

The following theorem shows that given $M = \{M_0, \dots, M_n\}$, no valid behaviors in M_i and M_j for $0 \leq i, j \leq n$ are removed by $\text{sync}(M_i, M_j)$. The valid behaviors in M_i^C are also in M_i after refinement. This ensures soundness of the verification results when function $\text{sync}()$ is used.

Theorem 4.1: Let M_i for $0 \leq i \leq n$ be SGs, and $M = \{M_0, \dots, M_n\}$. Also let M'_i and M'_j be two SGs such that $(M'_i, M'_j) = \text{sync}(M_i, M_j)$ for all $0 \leq i, j \leq n$ and $i \neq j$. The following condition holds for M'_i and M'_j .

$$M[\mathcal{A}_i] \preceq M'_i \text{ and } M[\mathcal{A}_j] \preceq M'_j$$

Proof: According to property (2), the following condition holds.

$$M[\mathcal{A}_i] = \{M_0, \dots, M_n\}[\mathcal{A}_i] \preceq \{M_i, M_j\}[\mathcal{A}_i] \quad (3)$$

Note that the principle of function $\text{sync}()$ follows parallel composition. According to the definition of parallel composition, for every state transition $((s_i, s_j), a, (s'_i, s'_j))$ in $\{M_i, M_j\}$, there is a (s_i, a, s'_i) in M'_i . This implies that for every path ρ_{ij} in $\{M_i, M_j\}$, there exists a path ρ_i in M'_i such that $\rho_{ij}[\mathcal{A}_i] = \rho_i$. This means that the following condition holds

$$\{M_i, M_j\}[\mathcal{A}_i] \preceq M'_i \quad (4)$$

Therefore, combining (3) and (4) leads to

$$M[\mathcal{A}_i] \preceq M'_i$$

The same argument applies to M'_j , too. ■

As an example, consider refining SGs in Fig.2(c) and (d) using synchronization described above. From the initial states, $(s_0, z+, s_1)$ in Fig.2(c) and $(s_0, z+, s_{16})$ in Fig.2(d) are synchronized, while $(s_0, x+, s_9)$ in Fig.2(d) is not synchronized in s_0 with the SG in Fig.2(c). In fact, this transition is not synchronized with any transition in SG in Fig.2(c), and is removed eventually. This causes s_9 in Fig.2(d) to become unreachable. After refinement is done, SGs in Fig.2(b), (c), and (d) are reduced to the ones shown in Fig.3(a), (b), and (c), respectively. Note that transitions to the failure state and the failure state itself disappear in all refined SGs.

4.3 Simultaneous Multi-Synchronization

Function $sync(M_i, M_j)$ works fine if the components of a system have simple interactions. However, this is not always true in practice, and the function may not result in good refinement if there exist inter-dependencies among the components. Consider the example shown in Fig. 4, which shows three components in a system and their partial SGs. Among them, M_1 shares actions a and d with M_2 and M_3 , respectively, and M_2 and M_3 also share an action e . According to M_2 and M_3 in the figure, action d occurs before action a , therefore path (s_0, a, s_1, d, s_2) in M_1 is invalid. However, applying $sync(M_i, M_j)$ for $i, j = 1, 2, 3$ and $i \neq j$ cannot remove such a path from M_1 . This is because the ordering of actions a and d is determined by action e , which is invisible to M_1 when applying $sync(M_1, M_2)$ and $sync(M_1, M_3)$, respectively, therefore the correlation between a and d due to e is lost. This example illustrates the impact of the inter-dependencies among components on the refinement results.

To address this problem and take the component inter-dependencies into account, we extend function $sync()$ to synchronize multiple components simultaneously. For convenience, the function that synchronizes two components is denoted as $sync_2()$, while the function that synchronizes multiple components is denoted as $sync_n()$, which is also referred to as multi-synchronization. Function $sync_n()$ works similarly to $sync_2()$, but the key is that more actions become visible on the interfaces and participate in the synchronization process at the same time. In the following, we first illustrate the idea of this function by applying it to M_1 , M_2 , and M_3 in Fig. 4. Starting from $t_0 = (p_0, q_0, s_0)$, transitions (q_0, d, q_1) in M_3 and (s_0, d, s_3) in M_1 are synchronized, which result in a new composite state $t_1 = (p_0, q_1, s_3)$. In this new state, (p_0, e, p_1) in M_2 and (q_1, e, q_2) in M_3 are synchronized, thus resulting another new composite state $t_2 = (p_1, q_2, s_3)$. From this state, (p_1, a, p_2) in M_2 and (s_3, a, s_4) in M_1 are synchronized resulting in $t_3 = (p_2, q_2, s_4)$. In the above synchronization process, only the path $(t_0, d, t_1, e, t_2, a, t_3)$ is explored. After the synchronization is done, transition (s_0, a, s_1) and (s_1, d, s_2) and states s_1 and s_2 in M_1 are removed, and the reduced M_1 is shown in Fig. 4(d). This example shows that by making

more actions visible for synchronization leads to better refinement results.

From the above description, $sync_2()$ is a special case of $sync_n()$, both of which are denoted as $sync()$ in the rest of this paper. Algorithm 1 shows a pseudo procedure for the multi-synchronization function $sync$. This procedure takes a set of k component SGs. First, the shared actions between each pair of components are found, and together they are grouped into a set $\mathcal{A}$, which is the set of all visible actions in this set of components. Next, a tuple of component initial states are pushed onto a stack. Then, the procedure iterates until the stack is empty. In each iteration, the component states in the tuple on the top of stack is popped. For each pair of popped component states, all their outgoing transitions in these states are considered. If the actions a_h and a_j of transitions (s_h, a_h, s'_h) and (s_j, a_j, s'_j) in different components matches and they are in $\mathcal{A}$, these transitions are synchronized, and added into R'_h and R'_j , respectively. If actions are not the same but not in $\mathcal{A}$, they are automatically synchronized and also added into R'_h and R'_j , respectively. In each case, a new tuple of component states is created, and pushed onto stack for future synchronization. At the end of synchronization, R'_h for $0 \leq h \leq k$ stores all synchronized transitions of component M_h , and it replaces the state transition set in the original SG. And finally, there may be some states to become unreachable with respect to the new but smaller transition set, and they need to be removed.

Algorithm 1 is similar to the definition of parallel composition. However, the ultimate goal of this algorithm is to identify the synchronized transitions for each component efficiently, therefore the complete composite SG is not generated during synchronization as in parallel composition. In the algorithm, function $sync()$ does not store the reachable state transitions of the composite SG. Instead, the transitions in the individual components are marked synchronized corresponding to the reachable transitions found during synchronization. On the other hand, the reachable composite states (state tuples in the algorithm) found during synchronization are stored to guarantee the termination of the function. The time complexity of function $sync()$ is close to that of parallel composition since conceptually the function needs to explore all composite states reachable in the composite SG. The space complexity of that function can be much lower than that of parallel composition since the complete composite SG is not constructed and the memory usage is reduced by not storing the potentially very large number of composite state transitions.

4.4 Determining k and Partitioning

The abstraction refinement framework described in the previous section can be more effective when multi-synchronization function $sync()$ is used. Ideally, if function $sync()$ is applied to all components in a system simultaneously, then each component can be maximally

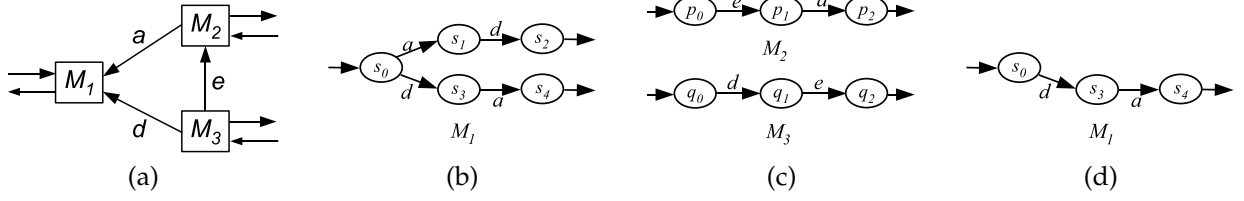


Fig. 4. (a) Components of a system with inter-dependency on e between M_2 and M_3 . (b) and (c) The portions of SGs for M_1 , M_2 , and M_3 . (d) SG of M_1 after synchronizing with M_2 and M_3 simultaneously.

Algorithm 1: $\text{sync}(\{M_i, M_{i+1}, \dots, M_{i+k}\})$

```

1  $\mathcal{A}_{hj} = \mathcal{A}_h^I \cap \mathcal{A}_j^O$  for  $0 \leq h, j \leq k$ , and  $i \neq j$ ;
2  $\mathcal{A} = \bigcup_{h,j=0}^k \mathcal{A}_{hj}$  for  $h \neq j$ ;
3 Push  $(init_i, init_{i+1}, \dots, init_{i+k})$  onto stack;
4 while stack is not empty do
5   Let  $(s_i, s_{i+1}, \dots, s_{i+k})$  be the top of stack;
6   Pop stack;
7   foreach pair of  $s_h$  and  $s_j$  such that  $h \neq j$  do
8      $q = (s_i, s_{i+1}, \dots, s_{i+k})$ ;
9     foreach  $(s_h, a_h, s'_h) \in \text{outgoing}(s_h)$  do
10      foreach  $(s_j, a_j, s'_j) \in \text{outgoing}(s_j)$  do
11        if  $a_h = a_j \wedge a_h \in \mathcal{A} \wedge a_j \in \mathcal{A}$  then
12          Add  $(s_h, a_h, a'_h)$  into  $R'_h$ ;
13          Add  $(s_j, a_j, a'_j)$  into  $R'_j$ ;
14          Replace  $s_h$  in  $q$  with  $s'_h$ ;
15          Replace  $s_j$  in  $q$  with  $s'_j$ ;
16        else if  $a_h \notin \mathcal{A}$  then
17          Add  $(s_h, a_h, a'_h)$  into  $R'_h$ ;
18          Replace  $s_h$  in  $q$  with  $s'_h$ ;
19        else if  $a_j \notin \mathcal{A}$  then
20          Add  $(s_j, a_j, a'_j)$  into  $R'_j$ ;
21          Replace  $s_j$  in  $q$  with  $s'_j$ ;
22   Push  $q$  onto stack;
23 foreach  $0 \leq h \leq k$  do
24    $R_h = R'_h$ ;
25   Remove unreachable states from  $S_h$ ;
```

refined. Even though the space complexity of function $\text{sync}(\{M_0, \dots, M_n\})$ is lower than that of parallel composition of $\{M_0, \dots, M_n\}$, all reachable composite states still need to be generated, and this number can be very large in practice. Therefore, the memory requirement may be prohibitively high if the number n of components for $\text{sync}()$ is too large. This put a limit on the number of components and the size of each component that can be taken by $\text{sync}()$. To control the size when calling $\text{sync}()$, only a subset of all components in a system are handled at a time. Given a design $M = \{M_0, \dots, M_n\}$ and a number k , it is divided into subsets $Q_0, \dots$, and Q_l such that

- $\forall 0 \leq i \leq l. |Q_i| \leq k$.
- $Q_0 \cup \dots \cup Q_l = M$.
- The following condition holds for any Q_i .

$$\forall M_h \in Q_i. \exists M_j \in Q_i. \mathcal{A}_h^O \cap \mathcal{A}_j^I \neq \emptyset \vee \mathcal{A}_h^I \cap \mathcal{A}_j^O \neq \emptyset$$

Note that it is possible for a component to be in different subsets if it shares common interfaces with multiple components which cannot be grouped together. The last condition above requires that the system structures need to be examined for partitioning. Each component in a subset needs to share interfaces with at least another component. Otherwise, such a component is not grouped in that subset to avoid unnecessary work.

In our method, k is first set to 2, and all components are grouped into subsets, each of which contains two components. Then verification with the iterative refinement process presented in the previous section is applied. If one or more components have failures, k is increased to 3 and the refinement is repeated. This sequence of the alternating steps continues until either of the termination conditions of the abstraction refinement process is satisfied, or k becomes too large for $\text{sync}()$. However, the experimental results show that $k = 2$ or 3 is usually enough, and is no larger than 4 for very finely partitioned systems with complex interactions among components.

To control the complexity during synchronization, users can also set an upper bound on k such that refinement stops when this upper bound is reached. When verification on a refinement fails and k is smaller than the upper bound, it is incremented by 1 indicating that one more component can be added into each subset. To avoid redundant work, a component is added into a subset if one of the following cases exists. The first case is as shown in Fig.4(a) such that a component is added if it shares common interface with 2 or more other components in the same subset. This makes sure that the inter-dependencies among components are taken into consideration. When k becomes larger, this condition is strengthened by requiring such a component have common interfaces with more components in the same subset for it to be added. In the second case, given two subsets Q_1 and Q_2 such that a component $M_j \in Q_1 \cap Q_2$. After refinement on these subsets separately, if M_j still has failures, this implies that M_j is a control point in a design whose correctness requires coordination of other components in Q_1 and Q_2 . Therefore, a new subset Q_3 is created which is defined as follows.

$$Q_3 = \{M_i \mid M_i \in Q_1 \cup Q_2 \wedge \mathcal{A}_i^O \cap \mathcal{A}_j^I \neq \emptyset\} \cup \{M_j\}$$

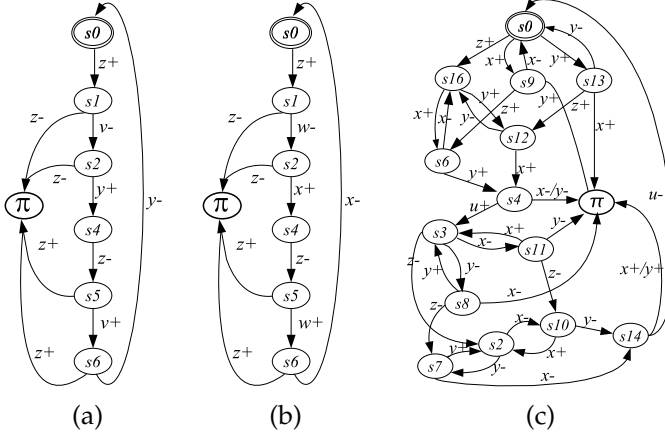


Fig. 5. (a) - (c) show the SGs from Fig. 2(b) - (d) after autofailure reduction, respectively.

5 REDUCTIONS

A potential issue in the presented method is the large size of the initial SGs and the complexity of synchronization. To address this issue, this section describes how two reduction techniques help to control the potential intermediate size blowup in this method.

5.1 On-the-Fly Autofailure Reduction

The first 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 [19]. However, autofailure manifestation in [19] is only used to canonicalize trace structures for hierarchical verification. It is adopted in our method as a technique to reduce SGs. More detail about this reduction technique and its soundness proof can be found in [36]. This reduction is applied after the SGs are generated in [36], while this section describes how it is combined with state space exploration so that the reduction is applied on-the-fly to control the size of the initial SGs, which is new in this method. First, a short overview of this reduction is given with a simple example. Next, we show algorithms how this reduction can be applied on-the-fly.

Let $\rho = (s_0, t_0, s_1, t_1, s_2, \dots, \pi)$ be a failure path in M . 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 $(s_{i+1}, a_{i+1}, s_{i+2}, \dots, \pi)$ is an autonomous subpath of ρ . All states on that autonomous subpath, referred to as *autofailure states*, 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.

Refer to the SG in Fig.2(d). The state transition $t_1 = (s_{15}, u-, \pi)$ is on an invisible action $u-$. Both incoming state transitions $t_2 = (s_8, x-, s_{15})$ 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 being removed. After autofailure reduction, the SGs shown in Fig. 2(b)-(d) are shown in Fig.5(a)-(c).

In this method, the SG of a component is found by reachability analysis from design descriptions in some high-level language. Since the environment for the component may be unknown at the beginning, it needs to be approximated very coarsely initially. This causes a large number of extra states and state transitions to be introduced during reachability analysis, thus resulting in significant size increase in the component SG. However, reachability analysis may find state transitions (s, a, π) such that a is a non-input action. In such a case, it is not necessary to explore other actions enabled in s because s is converted to π after autofailure reduction is invoked. Therefore, to remove these transitions as early as possible so as to better control the size of the initial SGs, autofailure reduction is combined with reachability analysis and executed on-the-fly to reduce the SG whenever a state transition (s, a, π) such that a is a non-input action is found.

Algorithm 2 shows a typical reachability analysis procedure based on depth-first search augmented with on-the-fly autofailure reduction. The basic reachability algorithm is a simplified version of the one in [31]. It takes a design description N , and starts from its initial state *init* by pushing *init*, a selected action a enabled in *init* for execution, and the remaining enabled actions in *init* onto stack. Then, the selected action is executed resulting in a different state s' . If s' is the failure state π or one of the autofailure states in F , and the executed action a is not an input action, then function *backtrack*(F, S, R), as to be described below, is called to pop some states out of stack and derive more autofailure states from s' . Then, the search continues by executing some action stored on the top of stack. Code from line 6 to line 8 in Algorithm 2 handles on-the-fly autofailure reduction. Otherwise, the search continues from s' if s' is new. If s' has been searched before ($s' \in S$), another action is selected from *enable*(s), and the search continues from s . The algorithm terminates when stack is empty. At the end of the algorithm, the autofailure states in F found by *backtrack*() are reduced to the failure state π , and S and R of the found SG also need to be reduced accordingly.

Function *backtrack*(F, S, R) is shown in Algorithm 3.

Algorithm 2: findSG (N)

```

1  $S = \emptyset, R = \emptyset, F = \{\pi\};$ 
2 Select an action  $a$  from  $enable(init);$ 
3 Push  $(init, enable(init) - \{a\}, a)$  onto stack;
4 while stack is not empty do
5   Execute action  $a$ , and find a new state  $s'$ ;
6   if  $s' = \pi \vee s' = F$  then
7      $F = backtrack(F, S, R);$ 
8     Select an action  $a$  from  $enable(s)$  on top of
      stack;
9   else
10    if  $enable(s)$  on top of stack is empty then
11      Pop stack;
12    else if  $s' \in S$  then
13      Select another action  $a$  from  $enable(s);$ 
14    else
15      Select an action  $a$  from  $enable(s');$ 
16      Push  $(s', enable(s') - \{a\}, a);$ 
17 Reduce states in  $F$  to the failure state;
18 Remove  $F$  from  $S;$ 
19 Adjust  $R$  accordingly;

```

It takes the set of autofailure states F and set of states S and transitions R found at that point, repetitively checks the state s and the action a on the top of stack, and pops stack if s is in F and a is a non-input action until the condition becomes false. All states popped out of stack are added into F since they are autofailure states too. After stack popping is done, the SG is searched backwards from F , and tries to discover more autofailure states. During backtracking, if there is a transition from the initial state to a state in F , the component fails no matter how the environment behaves, and verification terminates with a message indicating such a failure. Given $s_2 \in F$, s_1 of a transition (s_1, a_1, s_2) is also an autofailure state and such a transition is removed if a_1 is a non-input action. Otherwise (s_1, a_1, s_2) is replaced with (s_1, a_1, π) . The backward search repeats until F reaches a fixpoint. When the backward search is done, all autofailure states in F are returned. Experimental results show that applying on-the-fly autofailure reduction with reachability analysis can reduce the size of the initially generated SGs significantly.

5.2 Partial Order Reduction

It has long been recognized that state explosion in asynchronous systems is due to the excessively large number of interleavings of transitions in different components [26], [10]. When synchronizing components in this method, the number of composite states generated can also be very large due to the interleavings of invisible transitions in different components. Therefore, the complexity of synchronization can be lowered if the number of transitions in each component is reduced before they are synchronized.

Algorithm 3: backtrack (F, S, R)

```

1 while stack is not empty do
2   Let  $(s, enable(s), a)$  be the top of stack;
3   if  $a \notin A^I$  then
4     Add  $s$  into  $F;$ 
5     Pop stack;
6   else
7     break;
8  $Z = \emptyset;$ 
9 while  $Z \neq F$  do
10   $Z = F;$ 
11  foreach  $(s_1, a_1, s_2) \in R \wedge s_2 \in F$  do
12    if  $s_1 = init \wedge a_1 \notin A^I$  then
13      return “ $M$  has a failure”;
14    if  $a_1 \notin A^I$  then
15      delete  $(s_1, a_1, s_2);$ 
16       $F = F \cup \{s_1\};$ 
17    else
18      replace  $(s_1, a_1, s_2)$  with  $(s_1, a_1, \pi);$ 
19 return  $F;$ 

```

Partial order reduction can remove unnecessary interleaved transitions significantly. It is implemented in this method to reduce the size of SGs generated initially and during the refinement process. The implementation follows what is described in [26], [10], and the detail is not given. Care must be taken not to reduce any interface behavior when using partial order reduction. This is because it is unknown whether an interface behavior is valid or not until refinement is done.

Preserving all possible interface behavior, however, has a negative effect on how much reduction can be achieved by partial order reduction. Before refinement, many states of a component may have input actions enabled due to the over-approximate environment used initially, therefore it is possible that not much reduction can be obtained. Additionally, the size of each component cannot be too large to avoid the initial size blowup due to the over-approximate environment used. This limits the number of invisible actions in a component, thus further limiting the effectiveness of partial order reduction. Despite all the above issues, it still helps to remove at least some unnecessary transitions, therefore contributing to lowering the complexity of synchronization.

In this method, partial order reduction is used in Algorithm 2 to contain the sizes of component SGs initially generated. It is also used during synchronization to reduce the number of composite states to be explored. This helps to speed up the overall refinement process.

6 EXPERIMENTAL RESULTS

We have implemented a prototype of the automated compositional verification with the abstraction refinement method described in this paper in an asynchronous

TABLE 1
Experimental results and comparison with the method in [36].

Design	#C	A	Method 1				Method 2				
			Mem (MB)	Time (Sec.)	#Iter	# π	Mem (MB)	Time (Sec.)	#Iter	# π	k
FIFO	100	804	30	18	3	0	30	9.46	1	0	2
	200	1604	80	41	3	0	80	27	1	0	2
	400	3204	237	102	3	0	237	84	1	0	2
	600	4804	471	184	3	0	470	174	1	0	2
	800	6404	781	290	3	0	780	301	1	0	2
DME	20	440	35	43	4	0	14	14	1	0	2
	50	1100	88	113	4	0	40	39	1	0	2
	100	2200	191	249	4	0	97	96	1	0	2
	200	4400	446	600	4	0	264	257	1	0	2
	300	6600	771	1044	4	0	502	487	1	0	2
ARB	7	132	3	2	3	0	3	1.18	1	0	2
	15	244	7	6	5	0	6	2.99	1	0	2
	31	500	33	47	7	0	16	7.70	1	0	2
	63	1012	262	988	8	0	39	25	2	0	2
TAGUNIT	3	96	117	103	2	0	10	3.9	1	0	2
PIPECTRL	10	100	23	47	6	4	13	30	4	0	4
Average	181	2103	223	242	5	n/a	163	98	1	n/a	2

TABLE 2

Results from the abstraction refinement without any reduction, with only autofailure reduction (AFR), and with only partial order reduction (POR).

Design	#C	No Reduction		Static AFR		POR	
		Mem (MB)	Time (Sec.)	Mem (MB)	Time (Sec.)	Mem (MB)	Time (Sec.)
ARB	7	12	19	3	2.3	12	17
	15	19	62	7	6.9	19	60
	31	33	175	17	27	32	169
TAGUNIT	3	73	57	73	62	10	3.8
PIPECTRL	10	16	41	14	25	16	40
Average	13	31	71	23	25	18	58

system verification tool Plato, an explicit model checker, which can perform non-compositional and compositional verification. Experiments have been performed on several non-trivial asynchronous circuit designs to demonstrate the scalability of verification using the method presented in this paper.

6.1 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. To ensure soundness, an over-approximate environment for each component needs to be found to simulate the interface behaviors between the component and the rest of the design. The SGs generated this way are abstractions of the concrete ones. The *maximal environment* [23] is used for each component in all experiments. In the maximal environment for a component, all inputs of the component are totally free. Therefore, the maximal environment defines all possible behaviors on inputs of a component. The reason for choosing the maximal environment for each component is to generate the worse-case abstract SGs to experiment the effectiveness of the described refinement method. In practice, more restricted and accurate environment is highly desirable since it yields smaller and more concrete SG and thus making the described refinement method more efficient.

The first three designs are a self-timed FIFO [29], a tree arbiter of multiple cells [19], and a distributed mutual exclusion element consisting of a ring of DME cells [19]. 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 [35]. 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.

6.2 Analysis of Results

All experiments were performed on a Linux workstation with a Intel Pentium-D dual-core CPU and 1 GB

memory. The same experiments are also performed using ATACS [32], the closest relative to our method. As described before, although ATACS also supports modular verification, abstraction is done by Petri-net reductions, and they are limited to a particular type of Petri-nets, which is different from the formalism used in this method. In all experiments, little or no reduction is obtained when considering individual components. ATACS fails to verify all examples after exhausting 1 GB memory when generating the SG for the first component. The runtime and memory usage results from using ATACS are trivial, and they are not shown in the results table.

In Table 1, columns under *Method 1* show the results from using the approach in [36], while the results from using the method presented in this paper are shown in columns under *Method 2*. In both result tables, time is in seconds, and memory is in MBs. The last row of both tables shows the average results for all examples.

In Table 1, the first three columns, Design, $\#C$, and $|A|$, show the designs, the number of components after partitioning, and the size of A of each design, respectively. The four columns under *Method 1* show the peak memory, the total runtime, the total number of iterations needed to finish verifying each design, and the number of components containing the failures at the end, respectively. There are five columns under *Method 2*. The first four columns have the same meanings as those under *Method 1*. The last one shows the largest k to finish verifying each design successfully. The results shown under *Method 2* are obtained with on-the-fly autofailure and partial order reduction during refinement.

From the results under *Method 2* in Table 1, the following observations can be obtained. First of all, for scalable FIFO, ARB and DME examples, memory and runtime usages grow polynomially as the number of components in the systems increases as shown more clearly in Fig. 6. Second, all examples are verified failure free by this method, even though the results are still over-approximations of the concrete ones after refinement. Third, the peak memory is usually reached during synchronization. However, the memory usage is much smaller than that of parallel composition if the components were composed. As observed in the experiments, the memory usage required is highest at the beginning of the refinement process. This is because all component SGs are generated with the maximal environment, and these SGs contain a lot of extra states and state transitions, which make the complexity of synchronization higher. This indicates the negative effect of using the maximal environment for each component as the initial approximation. As indicated above, memory consumption may decrease if the initial approximate environment chosen for each component is more restricted than the maximal environment. On the other hand, these results show the effectiveness of this method to refine very coarse environment description to be accurate enough to finish verification successfully.

It is also interesting to notice that the number of

iterations for the refinement process to terminate is pretty small for all examples except PIPECTRL. In these experiments, k is first set to 2 for all examples. This is enough for all examples other than PIPECTRL to show failure-freedom. This is understandable in that all the designs other than PIPECTRL have well defined and relatively simple interface interactions. For PIPECTRL, k has to be increased to 4 to refine it to be failure-free. This is because the interactions among the components in PIPECTRL are much more complex, and there are rich inter-dependencies among the components.

Also note that with respect to compositional verification, only one component needs to stay in memory at a time. However, we keep SGs of all modules in memory for simplicity, and the memory numbers in columns under *Mem* show the total peak memory usage for all component during verification.

Compared with the results under *Method 1*, the presented method is much more effective. This method shows that PIPECTRL contains no failure while *Method 1* does not succeed. Additionally, for all examples, this method requires much less memory and runtime. It shows the effectiveness of the presented abstraction refinement method with synchronization to remove impossible behavior introduced by over-approximate environment for each component.

In all experiments, the maximal environment is used to find the SG for each component because we assume no knowledge of its interface. However, it causes a large number of extra states and state transitions to be introduced, including ones leading to the failure state. As described before, the failure traces can be trimmed using autofailure reduction to reduce the size of SGs. To show the effectiveness of on-the-fly autofailure and partial order reduction, three more sets of experiments are performed where no reduction is used during the whole refinement process, only autofailure reduction is used after the initial SGs are generated with the maximal environment, and only partial order reduction is used. The results are shown in Table 2.

Comparing the results in Table 2 and those under *Method 2* in Table 1, the runtime results are much worse without any reduction at all during refinement. For ARB 31, it takes 175 seconds for the whole verification process to finish while it takes only less than 8 seconds for the refinement process in *Method 2* to succeed. This is because all possible behaviors within the context where all inputs are totally free are generated, and it takes much more time for function *sync* to determine and remove behaviors that are not allowed by the interface synchronization of a component and its neighbors. Applying autofailure reduction after SGs are generated is not as bad as shown in columns under Static AFR in Table 2, but the runtime is still higher across all examples because more states and state transitions are generated in the first place compared to using on-the-fly autofailure reduction and more time is needed to perform autofailure reduction afterward. These results

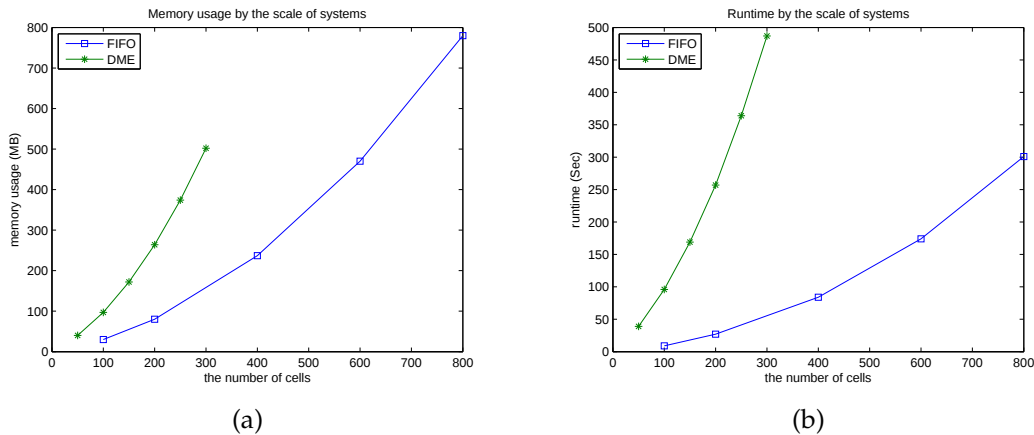


Fig. 6. Memory and runtime usage for FIFO and DME in the number of components.

show the effectiveness of on-the-fly autofailure reduction and the fundamental necessity of controlling the size of the initially generated SGs.

The results obtained with only partial order reduction used are shown in columns under POR in Table 2, and partial order reduction reduces runtime and memory significantly only for TAGUNIT, while it does not lead to much reduction for other examples. The reason might be that the largest component in TAGUNIT contains comparatively large number of invisible actions operating in parallel, which is more suitable for partial order reduction. However, neither reduction alone is sufficient in all cases, and better results can be obtained if they are used together along with the abstraction refinement method described in this paper.

Method 1 uses a different refinement approach with autofailure reduction applied after the initial component SGs generated. Therefore, the entries under “Static AFR” in Table 2 and those under *Method 1* in Table 1 show a better comparison between the refinement approaches in [36] and this paper. From these entries, it can be seen that the new refinement approach in this paper is better in terms of lower memory usage and runtime. With on-the-fly autofailure reduction combined with reachability analysis and partial order reduction, the effectiveness and performance of this work are much higher than that in [36] as shown by the results in Table 1.

If counter-examples still exist after refinement, they are checked on the whole design as in [38], which can be very expensive. From the above experimental results, it shows that the described abstraction refinement method can very effectively eliminate the extra behaviors, including those leading to counter-examples. This helps to avoid high computation penalty for confirming the false counter-examples.

7 CONCLUSION

While compositional verification is effective at attacking state explosion in model checking, generating accurate yet simple environments for system components poses

as a big challenge for compositional verification to be practical. This paper proposes an alternative method where generating such an environment is not important anymore. This is done by a novel abstraction refinement method where each component is refined monotonically and iteratively. The initial experimental results are very encouraging. This refinement algorithm is fully automated and general, and can be combined with other compositional verification approaches. In the future, we plan to investigate the abstraction refinement techniques at a higher abstraction level to generate more restrictive initial environment, and explore efficient partitioning methods for better performance.

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] M. Bobaru, C. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Proc. Int. Conf. on Computer Aided Verification*. LNCS, 2008.
- [5] 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.
- [6] S. Chaki, E. Clarke, J. Ouaknine, and N. Sharygina. Automated, compositional and iterative deadlock detection. In *MEMOCODE 2004*, pages 201–210, 2004.
- [7] 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.
- [8] S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, 1996.
- [9] 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.
- [10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Mass., 2001.

- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] L. de Alfaro and T. Henzinger. Interface automata. *Foundations of Software Engineering*, pages 109–120, 2001.
- [17] 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.
- [18] L. de Alfaro and T. Henzinger. Interface-based design. *Engineering Theories of Software-intensive Systems*, 195:83–104, 2005.
- [19] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.
- [20] 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.
- [21] D. Giannakopoulou, C.S.Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, pages 297–320, 2005.
- [22] 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.
- [23] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
- [24] 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.
- [25] 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.
- [26] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
- [27] 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.
- [28] F. Lang. Refined interface for compositional verification. In *FORTE’06: Formal Techniques for Networked and Distributed Systems*, volume 4229 of LNCS. Springer Verlag, 2006.
- [29] 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.
- [30] K. L. Mcmillan. A methodology for hardware verification using compositional model checking. Technical report, Cadence Berkeley Labs, 1999.
- [31] C. J. Myers. *Asynchronous Circuit Design*. Wiley Inter-Science, 2001.
- [32] 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.
- [33] W. Nam and R. Alur. Learningbased symbolic assume-guarantee reasoning with automatic decomposition. In *Proc. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4218 of LNCS, 2006.
- [34] A. Pnueli. In transition from global to modular temporal reasoning about programs. pages 123–144, 1985.
- [35] 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.
- [36] H. Yao and H. Zheng. Automated interface refinement for compositional verification. to appear in *IEEE Transactions on Computer-Aided Design of Integrate Circuits and Systems*, 2008.
- [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, C. Myers, D. Walter, S. Little, and T. Yoneda. Verification of timed circuits with failure directed abstractions. *IEEE Transactions on Computer-Aided Design*, 25(3):403–412, 2006.



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.



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 verification for software and hardware, primarily in the area of compositional reasoning, abstraction and reduction techniques for model checking.



Tomohiro Yoneda Tomohiro Yoneda received B.E., M.E., and Dr. Eng. degrees in Computer Science from the Tokyo Institute of Technology, Tokyo, Japan in 1980, 1982, and 1985, respectively. In 1985 he joined the staff of Tokyo Institute of Technology, and he moved to National Institute of Informatics in 2002, where he is currently a Professor. He was a visiting researcher of Carnegie Mellon University from 1990 to 1991. His research activities currently focus on formal verification of hardware and synthesis of asynchronous circuits. Dr. Yoneda is a member of IEEE, Institute of Electronics, Information, and Communication Engineers of Japan, and Information Processing Society of Japan.