# A Compositional Minimization Approach for Large Asynchronous Design Verification

Hao Zheng[1], Emmanuel Rodriguez[1], Yingying Zhang[1], and Chris Myers[2]

[1] University of South Florida, Tampa FL 33620, USA,
zheng@cse.usf.edu, yingyingz@mail.usf.edu>
erodrig9@gmail.com
[2] University of Utah, SLC UT 84112, USA
myers@ece.utah.edu

**Abstract.** This paper presents a compositional minimization approach with efficient state space reductions for verifying non-trivial asynchronous designs. These reductions can result in a reduced model that contains the exact same set of observably equivalent behavior in the original model, therefore no false counter-examples result from the verification of the reduced model. This approach allows designs that cannot be handled monolithically or with partial-order reduction to be verified without difficulty. The experimental results show significant scale-up of the compositional minimization approach using these reductions on a number of large asynchronous designs.

**Keywords:** model checking, compositional verification, minimization, abstraction

## 1 Introduction

*Compositional verification* is essential to address the state explosion problem in model checking large systems. The compositional methods can be roughly classified into *compositional reasoning* or *compositional minimization*. *Assume-guarantee* based compositional reasoning [2, 8, 13, 14, 18] does not construct the global state space. Instead, the 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, 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 the 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.


UU IR Author Manuscript

In the *learning-based* approaches, assumptions represented by deterministic finite automata are generated with the $L^*$ learning algorithm and analysis of local counter-examples [20, 1, 9, 11, 5]. 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 [20, 1]. An automated interface refinement method is presented in [23] where the models of the system modules are refined, and the extra behavior is removed by extracting the interface interactions among these modules. Although the capability of these methods has been demonstrated by verifying large examples, it is difficult for them to handle inherently global properties such as deadlock freedom.

Compositional minimization [4, 12, 16], on the other hand, iteratively constructs the local model for each component in a system, minimizes it, and composes it with the minimized models of other components. Eventually, a reduced global model is formed for the entire system where verification is performed. To contain the size of the intermediate results, user-provided context constraints are required. The need for the user-provided context constraints may also be a problem because the user-provided constraints may be overly restrictive, thus resulting in real design errors escaping detection. Similar work is also described in [6, 7].

The key to the success of compositional minimization is state space reduction. In the most existing work, reduction is conservative in that more behavior may be introduced, but no essential behavior may be removed during reduction. This is necessary since no real errors can be missed when verifying the reduced model. However, false errors may be introduced by reduction in the same time. When an error is found while verifying such a reduced model, it needs to be checked whether this error is real, typically done on the concrete model. This can be very time-consuming. If reduction is too conservative, the number of false errors may become too excessive, and checking these false errors can become the bottleneck.

In [22, 27, 28], methods are described for compositionally verifying asynchronous designs based on Petri-net reduction. These methods simplify Petri-net models of asynchronous designs either following the design partitions or directed by the properties to be verified, then verification is done on the reduced Petri-nets. However, these methods are limited to certain types of Petri-nets, and not easily extended to other formalisms.

This paper presents a number of state space reductions that can be used with compositional minimization. In this method, a design is modeled as a parallel composition of state graphs derived from the high-level descriptions of the components in a design. Before composing the component state graphs to form a global model for verification, these state graphs are reduced to lower the complexity. The reductions remove certain state transitions and states from a state graph in such a way that the observable behavior on the interface remains the same. At the end, a reduced state graph for the entire design, which is equivalent to the concrete model of the design in terms of observable behavior, is produced


UU IR Author Manuscript

for verification. This method is sound and complete in that the reduced model is verified to be correct if and only if the concrete model is correct.

The reduction method presented in this paper is similar, in some degree, to the partial order reduction method[15] as both try to identify and remove certain transitions to eliminate equivalent paths. Partial order reduction determines the independent transitions such that the order of executing these transitions does not affect the verification results, and it removes all but one independent transition in each state during the state space traversal to avoid generating states and transitions that correspond to some equivalent paths. However, determining which transitions are independent requires the information of the global state space, which is not available during the state space traversal, therefore, the independent transitions are computed conservatively to ensure soundness of the verification results. This causes partial order reduction to be less effective or even useless in some situations. On the other hand, our method can effectively remove all transitions that correspond to equivalent paths in state space models because it considers the generated state space models where the necessary information is available for such reduction. Furthermore, our method can also remove states that do not affect the observable behavior after the equivalent paths are removed, while partial order reduction only tries to avoid generating the equivalent paths. Another difference is that partial order reduction is applied to the whole design, while the method in this paper builds a reduced global state space model compositionally.

This paper is organized as follows. Section 2 gives a brief overview of the modeling and verification of asynchronous designs. Section 3 presents the set of state space reductions for our compositional verification method. Section 3.1 describes a state space reduction approach that preserves the same observably equivalent behavior. Section 3.2 describes a set of techniques that remove redundant states and state transitions to augment the reduction presented in Section 3.1. Section 4 demonstrates our method on a number of non-trivial asynchronous design examples, and it analyzes the obtained results. The last section concludes the paper and points out some future work that can improve this method.

## 2 Preliminaries

### 2.1 State Graphs

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

**Definition 21 (State Graphs)** *A state graph $G$ 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. *$R \subseteq S \times \mathcal{A} \times S$ is the set of state transitions,*
4. *$init \in S$ is the initial state.*


UU IR Author Manuscript

For an SG, $\mathcal{A} = \mathcal{A}^I \cup \mathcal{A}^O \cup \mathcal{A}^X$. $\mathcal{A}^I$ 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. $\mathcal{A}^X$ represents the internal behavior that is invisible at the interface, and it is usually denoted as $\zeta$. In the above definition, $S$ also includes a special state $\pi$ which denotes the *failure state* of a SG, and it represents violations of some prescribed safety properties. The failure state $\pi$ does not have any outgoing transitions. The set of actions enabled at a state $s \in S$ is denoted as $enb(s) = \{a \mid (s, a, s') \in R\}$. The set of state transitions leaving a state $s$, $\{(s, a, s') \in R\}$, is denoted by $out(s)$. In the remainder of this paper, $R(s, a, s')$ also denotes that $(s, a, s') \in R$.

A path $\rho$ of $G$ is a sequence of alternating states and actions of $G$, $\rho = (s_0, a_0, s_1, a_1, s_2, \cdots)$ 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 state $s_j \in S$ is *reachable from* a state $s_i \in S$ if there exists a path $\rho = (s_i, \cdots, s_j, \cdots)$ in $G$. A state $s$ is reachable in $G$ if $s$ is reachable from the initial state *init*. The trace of path $\rho$, denoted by $\sigma(\rho)$, is the sequence of actions $(a_0, a_1, \cdots)$. Given a trace $\sigma(\rho)$ of a path $\rho = (s_0, a_0, \ldots, s_i, a_i, \ldots)$, its finite prefix, denoted by $\sigma(\rho, i)$, is $(a_0, \ldots, a_i)$. Two traces $\sigma = (a_0, a_1, \cdots)$ and $\sigma' = (a'_0, a'_1, \cdots)$ are *equivalent*, denoted by $\sigma = \sigma'$, iff $\forall_{i \geq 0}\ a_i = a'_i$. The set of all paths of $G$ forms the language of $G$, denoted by $\mathcal{L}(G)$.

Given a trace $\sigma = (a_0, a_1, \ldots)$, its projection onto $\mathcal{A}' \subseteq \mathcal{A}$, denoted by $\sigma[\mathcal{A}']$, is obtained by removing from $\sigma$ all the actions $a \notin \mathcal{A}'$ as shown below.

$$\sigma[\mathcal{A}'] = \begin{cases} a_0 \circ \sigma'[\mathcal{A}'] & \text{if } a_0 \in \mathcal{A}', \\ \sigma'[\mathcal{A}'] & \text{otherwise.} \end{cases}$$

where $\sigma' = (a_1, ...)$, and $\circ$ is the concatenation operator.

Given two paths, their equivalence is defined as follows.

**Definition 22** *Let $\rho = (s_0, a_0, s_1, a_1, \cdots)$ and $\rho' = (s'_0, a'_0, s'_1, a'_1, \cdots)$ be two paths of $G$. $\rho$ and $\rho'$ are equivalent, denoted as $\rho \sim \rho'$, iff $\sigma(\rho) = \sigma(\rho')$.*

The SG of a system is obtained by composing the component SGs asynchronously. Asynchronous parallel composition is defined as follows. This definition is similar to that in [3] except that more rules are created for situations involving $\pi$. Given $G_1 = (\mathcal{A}_1, S_1, R_1, init_1)$ and $G_2 = (\mathcal{A}_2, S_2, R_2, init_2)$, the parallel composition of $G_1$ and $G_2$, $G_1 \| G_2 = (\mathcal{A}, S, R, init)$, is defined as follows.

1. $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$,
2. $S \subseteq S_1 \backslash \pi \times S_2 \backslash \pi \cup \{\pi\}$.
3. $R \subseteq S \times \mathcal{A} \times S$ such that all the following conditions hold:
   (a) For each $((s_1, s_2), a, (s'_1, s'_2)) \in R$,
      i. $a \in \mathcal{A}_1 - \mathcal{A}_2 \Rightarrow R_1(s_1, a, s'_1) \wedge s'_2 = s_2$,
      ii. $a \in \mathcal{A}_2 - \mathcal{A}_1 \Rightarrow R_2(s_2, a, s'_2) \wedge s'_1 = s_1$,
      iii. $a \in \mathcal{A}_1 \cap \mathcal{A}_2 \Rightarrow R_1(s_1, a, s'_1) \wedge R_2(s_2, a, s'_2)$,
   (b) For each $((s_1, s_2), a, \pi) \in R$,
      i. $a \in \mathcal{A}_1 - \mathcal{A}_2 \Rightarrow R_1(s_1, a, \pi)$,


UU IR Author Manuscript

ii. $a \in \mathcal{A}_2 - \mathcal{A}_1 \Rightarrow R_2(s_2, a, \pi)$,
iii. $a \in \mathcal{A}_1 \cap \mathcal{A}_2 \Rightarrow ((R_1(s_1, a, \pi) \wedge a \in enb(s_2)) \vee ((R_2(s_2, a, \pi) \wedge a \in enb(s_1))$,

4. $init = (init_1, init_2)$.

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 any individual SG makes a state transition to the failure state, there is a corresponding state transition to the failure state in the composite SG. In the actual implementation, when composing two SGs, a reachability analysis algorithm is performed from the initial composite state following the definition for transition relation $R$, and therefore, the resulting composite SG contains only the reachable states.

### 2.2 Correctness Definition

A path is referred to as a *failure* if it leads to the failure state $\pi$. The set of all failures in $G$ is denoted as $\mathcal{F}(G)$ such that $\mathcal{F}(G) \subseteq \mathcal{L}(G)$ holds. A system is correct if $\mathcal{F}(G) = \emptyset$.

Given a failure $\rho' = (s'_0, a_0, \cdots, s'_i, a_i, \pi)$, the non-failure prefix of its trace is $\sigma(\rho', i)$. If another path $\rho$ has the same non-failure prefix of $\rho'$, $\rho$ is also regarded as a failure. In such cases, path $\rho$ is said to be *failure equivalent* to $\rho'$.

**Definition 23** *Let $\rho = (s_0, a_0, \ldots)$ and $\rho' = (s'_0, a'_0, \ldots)$ be two paths. If $\exists_{i>0}\ \sigma(\rho, i) = \sigma(\rho', i) \wedge s'_{i+1} = \pi$ holds, then $\rho$ is failure equivalent to $\rho'$, denoted as $\rho \sim_F \rho'$.*

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

**Definition 24 (Abstraction)** *Given SGs $G$ and $G_1$, $G_1$ is an abstraction of $G$, denoted as $G \preceq G_1$, if and only if the following conditions hold:*

1. *$\mathcal{A}^I = \mathcal{A}^I_1$ and $\mathcal{A}^O = \mathcal{A}^O_1$.*
2. *For every path $\rho \in \mathcal{L}(G)$, there exists a path $\rho_1 \in \mathcal{L}(G_1)$ such that $\rho[\mathcal{A}'] \sim \rho_1[\mathcal{A}']$ or $\rho[\mathcal{A}'] \sim_F \rho_1[\mathcal{A}']$ where $\mathcal{A}' = \mathcal{A}^I \cup \mathcal{A}^O$.*

The abstraction relation defines that for any path in $G$, there exists a path in $G_1$ such that they are observably equivalent. For any failure in $G$, there exists an equivalent failure in $G_1$.

The equivalence relation between two SGs is more restricted than the abstraction relation.

**Definition 25 (Equivalence)** *Let $G$ and $G_1$ be SGs. $G$ is equivalent to $G_1$, denoted as $G \equiv G_1$, if and only if $G \preceq G_1$ and $G_1 \preceq G$.*


UU IR Author Manuscript

The equivalence relation defines that two SGs contain the same set of observably equivalent paths. Therefore, if $G \equiv G_1$, the following property holds.

$$\mathcal{F}(G) = \emptyset \quad \Leftrightarrow \quad \mathcal{F}(G_1) = \emptyset. \tag{1}$$

Intuitively, the above property states that the concrete model $G$ is correct if $G_1$ is correct, and vice versa.

After a SG is generated, model checking can be applied for various properties to decide if they hold. In particular, our method checks the properties of safety and deadlock freedom of an asynchronous design. The correctness of a design is defined as the absence of failures caused by the violations of these properties. The failure state $\pi$ in our method can be used to capture violations of various safety properties. A design is safe if $\pi$ is unreachable. A design is said to deadlock if it cannot make progress in some state. It is defined as follows.

**Definition 26 (Deadlock)** *A SG is said to have a deadlock if* $\exists_{s \in S}\ enb(s) = \emptyset$.

A design is free of deadlock if no deadlock exists.

## 3 State Graph Reductions

In this method, it is assumed that a design consists of $n$ components, the state graphs $G_i (1 \leq i \leq n)$ for these components are obtained using the method described in [25]. The state graph for the whole design is obtained by composing the two component SGs in parallel at a time for all components. However, directly composing $G_i$ for verification defeats the purpose of compositional construction in that the interleaving of the invisible state transitions in $G_i$ can explode quickly during the parallel composition. Therefore, this section presents a number of state space reductions to simplify the component SGs and the intermediate SGs generated during the composition process before they are composed to control the complexity. The reduced state graphs are observably equivalent to the original ones, which implies that any properties hold or fail in the reduced SGs if, and only if, they hold or fail in the original ones. These reductions remove the redundant paths from the original SG but do not introduce any extra paths that do not exist in the original SG. They play an important role in compositional minimization. The end of this section compares these reductions with another existing state space abstraction approach.

### 3.1 Observably Equivalent Reduction

Given a component, some of its outputs may become 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 [6]. This may cause extra behaviors and thus may introduce false failures. This section provides a different reduction approach that compresses a


UU IR Author Manuscript

sequence of invisible state transitions into a single visible state transition. This approach 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 reduction, the whole subpath is replaced with state transition $(s_i, a_j, s_{j+1})$. This reduction is referred to as an *observably equivalent reduction*. This reduction 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 the reduced graph $G$ 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 reduction shortens the existing paths, but no new paths are created. Therefore, no new failure traces are introduced.
3. Non-determinism may be introduced into the SG after reduction. 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 may be eliminated if $s_{j+1}$ or $s_{k+1}$ is redundant as described in the next section.

Let `reduce`$(G)$ be a procedure for the observably equivalent reduction on a SG $G$ as shown in Algorithm 1. The SG produced by `reduce`$(G)$ in Algorithm 1 inherits every element of $G$ except the updated $R$ and $S$. The algorithm `reduce`$(G)$ checks each invisible state transition $(s_1, a_1, s_2)$ in $G$, and calls another function `oer`$(G, s_1, s_2)$ if the start state $s_1$ of that invisible state transition has at least one incoming state transition that is visible. Function `oer`$(G, s_1, s_2)$, as shown in Algorithm 2, searches forward bypassing each invisible state transition from $s_2$ in the depth-first manner until a visible transition or the failure state $\pi$ is encountered. Then, new visible transitions are created to replace the sequences of invisible state transitions, and they are added into $R$. After all invisible transitions are handled, they are removed from $G$. Consequently, some other states and transitions may become unreachable, and they are also removed from $G$.

Fig. 1 shows an example how a SG in Fig. 1(a) is reduced by the observably equivalent reduction to become the one as shown in Fig. 1(b). In this example, suppose all invisible transitions are denoted by $\zeta$. Then, for each visible transition in states $s_{i+1}$, $s_{j+1}$, and $s_{k+1}$, a new transition on the same action is created for states $s_i$, $s_j$, and $s_k$, respectively. Four new state transitions are added to preserve the same observable behavior. In this case, only three invisible transitions are removed. Therefore, without further reduction, the reduced SGs can actually be more complex with more transitions added. In the next section, redundancy in the SGs is defined, and algorithms are described to identify and remove the redundancy to actually reduce the complexity of the SGs.

The following lemma asserts that `reduce`$(G)$ is equivalent to $G$.

**Lemma 1.** *Given a SG $G$, $G \equiv$ `reduce`$(G)$.*


UU IR Author Manuscript

**Algorithm 1**: `reduce` $(G)$

```
1 foreach (s1, a1, s2) ∈ R do
2     if a1 = ζ ∧ s2 = π then
3         foreach (s, a, s1) ∈ R do
4             R = R ∪ {(s, a, π)};
5     else if a1 = ζ ∧ s2 ≠ π then
6         if ∃_{(s,a,s1)∈R} a ≠ ζ then
7             oer(G, s1, s2);
8 Remove all invisible state transitions from G;
9 Remove unreachable states and state transitions from G;
```

**Algorithm 2**: `oer` $(G, s_1, s_2)$

```
1 foreach (s2, a2, s2') ∈ R do
2     if a2 = ζ ∧ s2' ≠ π then
3         oer(G, s1, s2');
4     else
5         R = R ∪ {(s1, a2, s2')};
```

**Proof**: The proof is based on how procedure `reduce`$(G)$ works. It is straightforward to see that for every path $\rho$ in $G$ that does not include any invisible transitions, the same path also exists in `reduce`$(G)$. For a path $\rho = (s_1, a_1, \ldots, s_i, \zeta, s_{i+1}, a_{i+1}, \ldots)$, there exists a path $\rho' = (s_1, a_1, \ldots, s_i, a_{i+1}, \ldots)$ in `reduce`$(G)$, and $\rho \sim \rho'$ or $\rho \sim_F \rho'$.

Conversely, for every path $\rho' = (s_1, a_1, \ldots, s_i, a_{i+1}, \ldots)$ in `reduce`$(G)$, either the same path exists in $G$, or it is reduced from a path $\rho = (s_1, a_1, \ldots, s_i, \zeta, s_{i+1}, a_{i+1}, \ldots)$ in $G$, and $\rho' \sim \rho$ or $\rho' \sim_F \rho$. This satisfies the conditions of the equivalence relation, therefore $G \equiv$ `reduce`$(G)$. ∎

### 3.2 Redundancy Removal

From the example shown in the last section, it can be seen that the observably equivalent reduction can introduce nondeterminism. Nondeterminism exists if there are two state transitions $(s, a, s_1)$ and $(s, a, s_2)$ such that $s_1 \neq s_2$. This is a result from reduction while preserving observable equivalence. However, the introduced nondeterminism can potentially contain redundancy, and removing the redundancy can simplify the complexity of SGs.

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. It is formalized as failure equivalent state transitions in the following definition.

**Definition 31** *Given two state transitions $(s, a_1, s_1)$ and $(s, a_2, \pi)$ of a SG, $(s, a_1, s_1)$ is failure equivalent to $(s, a_2, \pi)$ if $a_1 = a_2$.*


UU IR Author Manuscript

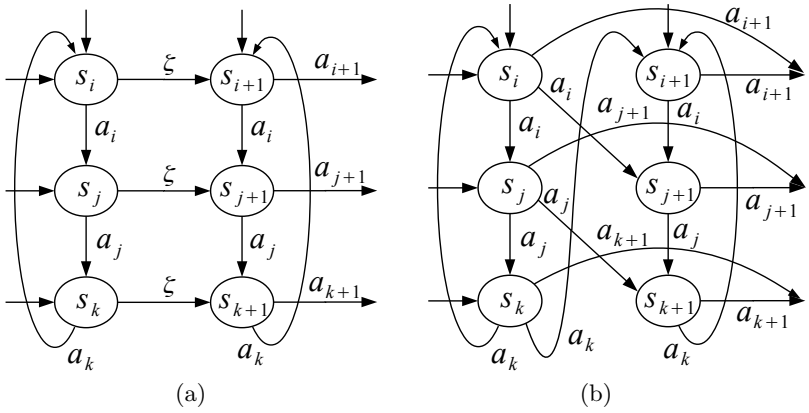


**Fig. 1.** (a) An example SG with invisible state transitions. (b) The SG from (a) after the observably equivalent reduction.

The failure equivalent transitions are redundant in that their existence does not affect the verification results, therefore, they can simply be removed. After removing the failure equivalent state transitions, it is possible that some other states become unreachable leading to more reduction.

The following lemma states that the SG resulting from removing failure equivalent transitions is equivalent to the original SG.

**Lemma 2.** *Let $G$ and $G'$ be a SG and the one after removing failure equivalent transitions. $G \equiv G'$.*

**Proof**: The following proof is drawn based on how the failure equivalent reduction works. First, all paths in $G$ also exist in $G'$ except for paths $\rho = (s_1, a_1, \ldots, s_i, a_i, s_{i+1}, \ldots)$ in $G$ such that there also exists a $(s_i, a_i, \pi)$ in $G$. In other words, for every path $\rho = (s_1, a_1, \ldots, s_i, a_i, s_{i+1}, \ldots)$ in $G$, if $(s_i, a_i, \pi)$ is also in $G$, there exists a path $\rho' = (s_1, a_1, \ldots, s_i, a_i, \pi)$, and we can see $\rho \sim_F \rho'$. This shows $G \preceq G'$.

Now, for every path $\rho'$ in $G'$, the path also exists in $G$ if it does not end in the failure state. If $\rho'$ ends in the failure state, the same path also exists in $G$. This shows that $G' \preceq G$. Therefore, $G \equiv G'$ holds. ∎

Fig. 2 shows an example of failure equivalent transitions. Fig. 2(a) is an example SG. After observably equivalent reduction is applied, the reduced SG is shown in Fig. 2(b). In this reduced SG, transition $(s_j, a_j, s_k)$ is failure equivalent to $(s_j, a_j, \pi)$. After removing this failure equivalent transition, state $s_k$ becomes unreachable, and it is also removed including all its outgoing transitions. The final reduced SG is shown in Fig. 2(c).

Next, a restricted case of redundancy is described. Let $incoming(s)$ be the set of state transitions $(s', a, s)$ such that $R(s', a, s)$ holds.


UU IR Author Manuscript

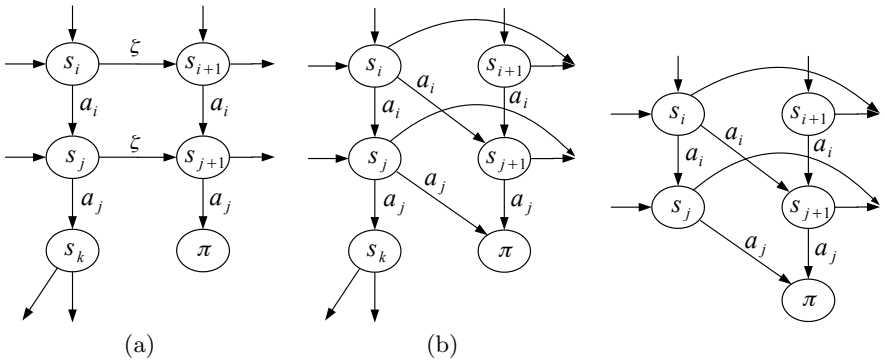


**Fig. 2.** (a) An example SG. (b) The SG from (a) after the observably equivalent reduction. (c) The SG from (b) after removing the failure equivalent transition $(s_j, a_j, s_k)$ and the unreachable state.

**Definition 32** *Let $G$ be a SG, and $s, s_1$, and $s_2$ be states of $G$. If the following conditions hold, then one of $s_1$ and $s_2$ is redundant.*

- *For every $(s, a, s_1) \in incoming(s_1)$, there exists a $(s, a, s_2) \in incoming(s_2)$.*
- *For every $(s, a, s_2) \in incoming(s_2)$, there exists a $(s, a, s_1) \in incoming(s_1)$.*

If such redundant states exist, one of them and its incoming and outgoing transitions can be removed as follows. Suppose $s_1$ is selected to remove.

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

Therefore, removing redundant states always results in a smaller number of states and state transitions. It is also obvious to see that $G \equiv G'$ where $G'$ is the SG after redundant states are removed from $G$.

In the remaining part of this section, a more general definition of redundancy is given by checking all possible behaviors originating from two states. Basically, if all possible behaviors originating from these two states are equivalent, these two states are regarded as equivalent. Therefore, one of them is redundant, and can be removed. The state equivalence is formally defined as follows.

**Definition 33** *Let $s$ and $s'$ be two states of a SG. $s$ and $s'$ are equivalent, denoted as $s \equiv s'$, if the following conditions hold.*

- *For each path $\rho = (s_0, a_0, s_1, a_1, \ldots,)$ such that $s_0 = s$, there exists another path $\rho' = (s_0', a_0, s_1', a_1, \ldots,)$ such that $s_0' = s'$, $\rho \sim \rho'$ or $\rho \sim_F \rho'$.*
- *For each path $\rho' = (s_0', a_0, s_1', a_1, \ldots,)$ such that $s_0' = s'$, there exists another path $\rho = (s_0, a_0, s_1, a_1, \ldots,)$ such that $s_0 = s$, $\rho \sim \rho'$ or $\rho \sim_F \rho'$.*


UU IR Author Manuscript

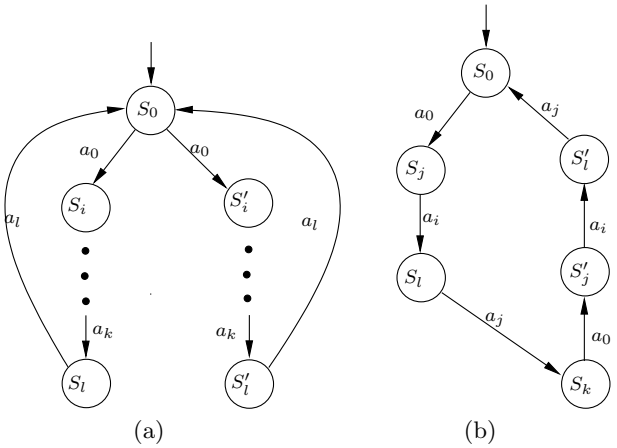


**Fig. 3.** Examples of equivalent states that can be resulted from reductions. States $s_i$ and $s'_i$ in (a) and $s_0$ and $s_k$ in (b) are equivalent since the paths coming out of these states are equivalent.

Fig. 3 shows two examples of SGs which contain equivalent states that possibly result from the reduction described in the previous section. In Fig.3(a), there are two loops. State $s_i$ on one loop is equivalent to state $s'_i$ on the other loop since the paths out of these states are equivalent. Similarly, the successor states of these two states are also equivalent. It can be shown that every state in one loop is equivalent to a corresponding state in the other loop. Fig. 3(b) shows a different case where equivalence exists. It can be shown that state $s_0$ is equivalent to $s_k$ since each of these two states is the starting state of a path, and these two paths are equivalent.

The above observation directly leads to an algorithm to find equivalent states. To simplify the presentation, assume a SG with $\mathcal{A}^X = \emptyset$ after observably equivalent reduction is applied. The algorithm works as follows. Initially, the set $Eq$ of all pairs of states is found such that for each $(s, s') \in Eq$, the following conditions hold.

- $\forall_{(s,a,s_1)\in outgoing(s)}\ \exists_{(s',a',s'_1)\in outgoing(s')}\ a = a'.$
- $\forall_{(s',a',s'_1)\in outgoing(s')}\ \exists_{(s,a,s_1)\in outgoing(s)}\ a = a'.$

Two states are obviously not equivalent if one has some enabled action that is not enabled in another state. This step excludes these obviously inequivalent states, and keeps the pairs that are potentially equivalent. Then, the algorithm iteratively removes from the set $Eq$ any pairs $(s, s')$, until a fixpoint is reached, if one of the following conditions holds

$$\exists_{s_1\in succ(s)}\forall_{s'_1\in succ(s')}\ (s_1, s'_1) \notin Eq \tag{2}$$

$$\exists_{s'_1\in succ(s')}\forall_{s_1\in succ(s)}\ (s_1, s'_1) \notin Eq \tag{3}$$


UU IR Author Manuscript

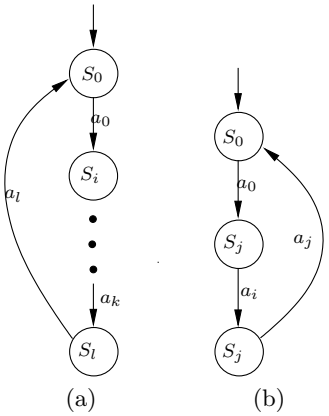


**Fig. 4.** SGs for the examples with redundant states in Fig. 3 after being reduced.

where $succ(s)$ includes all states that are reachable in one transition from $s$. Finally, if $Eq$ is not empty, then states in every pair $(s, s') \in Eq$ are equivalent. The correctness of the above algorithm is stated and proved in the following lemma.

**Lemma 3.** *For each pair* $(s, s') \in Eq$, $s \equiv s'$.

**Proof:** Suppose $(s, s')$ is an arbitrary pair in $Eq$.

Let $\rho = (s_0, a_0, s_1, a_1, \ldots)$ be an arbitrary path such that $s_0 = s$. Since $(s, s') \in Eq$, there exists $(s', a_0, s'_1) \in outgoing(s')$ corresponding to $(s, a_0, s_1)$. Additionally, $(s_1, s'_1) \in Eq$ because $(s, s') \in Eq$. Repeat the above argument for $(s_1, s'_1)$ and their successors recursively, we can construct another path $\rho' = (s', a_0, s'_1, a_1, \ldots)$, and it is straightforward to see that for any path from $s$, there is another path $\rho'$ such that $\rho \sim \rho'$.

Next, let $\rho' = (s'_0, a_0, s'_1, a_1, \ldots)$ be an arbitrary path such that $s'_0 = s'$. By following the above steps similarly, we can conclude that for any path from $s'$, there is another path $\rho$ such that $\rho \sim \rho'$.

Therefore, for every pair $(s, s') \in Eq$, $s \equiv s'$ by Definition 33. ∎

If $Eq(s, s')$ is not empty, for every pair $(s, s')$ in the set, either $s$ or $s'$ and its outgoing transitions can be safely removed, and its incoming transitions are re-directed to $s'$ or $s$. In this case, the interface behavior of the transformed SG remains the same as that of the original one according to the definition of the state equivalence. The examples shown in Fig. 3 after being reduced are shown in Fig. 4.

### 3.3 Comparison Between Reduction and Abstraction

Efficient and effective state space reductions are key to the success of compositional minimization. In [26], a different abstraction technique is presented. This section briefly compares it with the presented reductions in this paper.


UU IR Author Manuscript

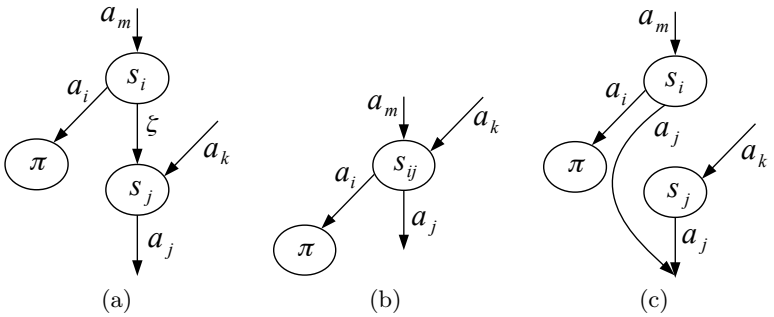


**Fig. 5.** Comparison of a traditional state space abstraction technique with the observably equivalent reduction. (a) An example SG. (b) The SG after the state space abstraction. (c) The SG after the observably equivalent reduction.

The state-based abstraction in [26] removes every invisible state transition $(s_i, \zeta, s_j) \in R$ from an SG, and merges $s_i$ and $s_j$ to form a merged state $s_{ij}$. All state transitions entering $s_i$ and $s_j$ now enter $s_{ij}$, and all state transitions leaving $s_i$ or $s_j$ now leave $s_{ij}$. To preserve failure traces, if $s_j$ is the failure state $\pi$, then the merged state $s_{ij}$ is also the failure state. This abstraction can remove all invisible state transitions from an SG, which is illustrated in Fig. 5. It is efficient to simply remove one invisible transition at a time without checking any conditions as required in the paper. However, it may introduce a lot of extra behavior including failures. In Fig. 5(b), there is a path $\rho = (\ldots, a_k, s_{ij}, a_i, \ldots)$ that does not exist in the SG in Fig. 5(a). This extra path causes a false failure in the final reduced SG.

The observably equivalent reduction presented in this paper removes invisible state transitions while keeping the exact same set of observable paths in the original SG. Another example of this reduction is shown in Fig. 5(c). For an invisible state transition $(s_i, \zeta, s_j) \in R$, this reduction adds a new state transition $(s_i, a_j, s_h)$ into $R$ for every $(s_j, a_j, s_h) \in \mathit{outgoing}(s_j)$. Then, it removes $(s_i, \zeta, s_{i+1})$. In Fig. 5(a), there exists a path $\rho = (\ldots, s_i, \zeta, s_j, a_j, s_h, \ldots)$, and in Fig. 5(c) there exists a path $\rho' = (\ldots, s_i, a_j, s_h, \ldots)$, and $\rho[\mathcal{A}'] \sim \rho'[\mathcal{A}']$ where $\mathcal{A}' = \mathcal{A}^I \cup \mathcal{A}^O$. For all other paths that do not involve $(s_i, \zeta, s_{i+1})$, they are preserved after the reduction. This reduction does not introduce any extra paths that do not exist in the original. On the other hand, it may introduce a large number of redundant paths that may cause the reduced SG to be much larger than the original one. Fortunately, the redundancy removal techniques presented in this paper can help to remove a lot of these redundancy introduced by the observably equivalent reduction to significantly simplify the complexity of SGs.


UU IR Author Manuscript

**Table 1.** Comparison of the results from using the monolithic, partial-order reduction and the reduction methods. Time is in seconds, and memory is in MBs. $|S|$ is the numbers of states found. For the results under CompMin, $|S|$ is the number of states of the largest SG encountered during the whole course of compositional minimization.

| Designs | | Monolithic | | | SPIN | | | CompMin | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | $\|V\|$ | Time | Mem | $\|S\|$ | Time | Mem | $\|S\|$ | Time | Mem | $\|S\|$ |
| fig3a | 6 | 0.044 | 2.7 | 20 | 0 | 2.195 | 20 | 0.037 | 3.14 | 10 |
| arbN3 | 26 | 0.315 | 2.4 | 3756 | 0.015 | 2.781 | 3756 | 0.087 | 3.89 | 52 |
| arbN5 | 44 | 8.105 | 61.538 | 227472 | 1.65 | 71.695 | 227472 | 0.18 | 4.3 | 52 |
| arbN7 | 62 | – | – | – | – | – | – | 0.46 | 6.61 | 52 |
| arbN9 | 80 | – | – | – | – | – | – | 0.89 | 7.43 | 52 |
| arbN15 | 134 | – | – | – | – | – | – | 1.33 | 9.87 | 52 |
| fifoN3 | 14 | 0.119 | 4.8 | 644 | 0 | 2.195 | 644 | 0.015 | 3.39 | 20 |
| fifoN5 | 22 | 0.733 | 16.253 | 20276 | 0.08 | 6.593 | 20276 | 0.017 | 3.62 | 20 |
| fifoN8 | 34 | 199.353 | 845 | 3572036 | 30.2 | 1087.211 | 3572036 | 0.11 | 4.03 | 20 |
| fifoN10 | 42 | – | – | – | – | – | – | 0.08 | 4.38 | 20 |
| fifoN20 | 82 | – | – | – | – | – | – | 0.11 | 4.7 | 20 |
| fifoN50 | 202 | – | – | – | – | – | – | 0.35 | 6.14 | 20 |
| fifoN100 | 402 | – | – | – | – | – | – | 0.76 | 7.67 | 20 |
| fifoN200 | 802 | – | – | – | – | – | – | 1.56 | 11.1 | 20 |
| fifoN300 | 1202 | – | – | – | – | – | – | 3.02 | 14.3 | 20 |
| dmeN3 | 33 | 3.589 | 26.1 | $267,999$ | 0.265 | 19.706 | 117270 | 0.71 | 4.44 | 248 |
| dmeN4 | 44 | 1235 | 1032 | $15.7M$ | 15.5 | 553.421 | 4678742 | 0.8 | 5.74 | 248 |
| dmeN5 | 55 | – | – | – | – | – | – | 2.23 | 10.19 | 248 |
| dmeN8 | 88 | – | – | – | – | – | – | 3.57 | 16.4 | 447 |
| dmeN9 | 99 | – | – | – | – | – | – | 5.86 | 20.9 | 900 |
| dmeN10 | 110 | – | – | – | – | – | – | 58.9 | 46.6 | 3211 |
| TU | 48 | – | – | – | 4.37 | 144.984 | 786672 | 0.219 | 5.085 | 278 |
| PC | 50 | – | – | – | – | – | – | 0.842 | 7.567 | 864 |
| MMU | 55 | – | – | – | – | – | – | 0.688 | 10.143 | 2071 |

## 4 Experimental Results

We have implemented a prototype of the automated compositional verification with the reductions described in this paper in a concurrent system verification tool *Platu*, an explicit state model checker. This model checker is programmed in Java, and can perform traditional depth-first search and compositional verification. Experiments have been performed on several non-trivial asynchronous circuit designs obtained from previously published papers. To verify a design using the compositional minimization method in this paper, all components in the design need to be converted to SGs first. The component SGs can be obtained using a compositional reachability analysis method as shown in [25]. Detailed description of this method is out of scope of this paper. In this paper, it is assumed that the component SGs are already obtained somehow.

The first three designs are a self-timed first-in-first-out (FIFO) design [17], a tree arbiter (ARB) of multiple cells [10], and a distributed mutual exclusion element (DME) consisting of a ring of DME cells [10]. Despite all these designs having regular structures to be scaled easily, the regularity is not exploited in our method, and all components are treated as black boxes. The fourth example is a tag unit circuit (TU) from Intel's RAPPID design [21]. 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 fifth example is a pipeline controller (PC) for an asynchronous processor TITAC2 [24]. The last example is a circuit implementation of a memory management unit (MMU) from [19]. All examples are too large for traditional monolithic approaches to complete on a typical workstation.

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 TU example, 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 PC example is partitioned into five components, each of which contains ten gates. The MMU example is partitioned by following the structure provided in [19] such that each component defines an output that are used by other components.

All experiments are performed on a Linux workstation with an Intel dual-core CPU and 2 GB memory. The results are shown in Table 1. In Table 1, the first two columns show the design names and the number of variables used in the corresponding models. Since all examples are asynchronous circuits, the type of the variables used in the models is Boolean. Three different methods are used in the experiments for better comparison. The columns under Monolithic show the results from using the traditional DFS search method on the whole designs. The columns under SPIN show the results from using the SPIN model checker with the partial-order reduction turned on. The last three columns under CompMin show the results from using the compositional minimization method described in this paper. In these columns, Time is the total runtime, Mem is the total memory used, and $|S|$ shows the total number of states found. Specifically, the column $|S|$ under CompMin shows the total number of states in the largest SG


UU IR Author Manuscript

found during the entire course of the compositional minimization process. The largest SGs are recorded because their sizes in general determine whether the whole process of compositional minimization can be finished or not, therefore, their sizes need to be carefully controlled. For examples which use too much memory, the corresponding entries are filled with −.

From Table 1, it can be seen that the traditional monolithic search method fails to finish quickly for most of the designs. This is understandable due to the state explosion problem. However, it is surprising to see that SPIN with partial-order reduction does not do any better. For all ARB and FIFO examples, SPIN cannot find any reduction, and the numbers of states found by SPIN are exactly the same as those found by the monolithic approaches for ARB and FIFO. For DME and TU, SPIN does slightly better in terms of reduction in of the number of states found. On the other hand, SPIN quickly blows up the 2 GB memory for most of the examples too. One possible explanation is that the partial-order reduction implemented in SPIN relies on the information about the independence among transitions, and this information is obtained by examining the structures of the Promela models. Since these examples are asynchronous circuit designs, the models for these examples are connections of descriptions of basic logic gates, and they may be difficult for SPIN to extract sufficient independence information for effective reduction.

On the other hand, the compositional minimization approach with all reductions described in this paper can finish all examples in the table quickly. For ARB and FIFO examples, the total runtime and memory usage grow polynomially in the number of components in the examples. For DME examples, the runtime and memory usage show a similar growth curve until the examples become too large. For dmeN10, there is a big jump on runtime and memory usage. This growth is due to an intermediate SG that contains too many state transitions after the equivalent reduction, and it takes a big part of total runtime to identify the equivalent states. The results for dmeN11 are not shown as the runtime for this example exceeds the 5 minute threshold. On the other hand, the memory usage still grows polynomially as the design size grows. For the three irregular designs, TU, PC and MMU, where SPIN also fails, they are finished with compositional minimization using very small amount of runtime and memory. For the PC example, a safety failure is found. The same safety failure is also found by the monolithic approach after about 30 minutes on a much more powerful machine.

From these results, one may conclude that compositional minimization works much better than partial-order reduction. This is true to some degree. For designs that do not contain any flaws, compositional minimization can prove the correctness very efficiently. On the other hand, for designs that contain one or more bugs, compositional minimization can also finish and return counter-examples quickly. However, as a lot of design details are removed during the minimization process, the returned counter-examples are very abstract, therefore not very useful for users to understand the causes of the bugs. In this case, concrete counter-examples corresponding to those returned by compositional minimization need

to be generated. This can be done by the traditional search on the whole design guided by the returned counter-examples. Since these counter-examples are so abstract, the step of generating the concrete counter-examples may, in some cases, be as difficult as searching the state space of the whole design.

## 5 Conclusion

This paper presents a compositional minimization approach with a number of state graph reductions to lower the verification complexity while not introducing extra paths that might cause false failures nor reducing any essential behaviors. In other words, the reduction methods are sound and complete. Based on initial experimental results, these reductions work well on a number of asynchronous circuit examples. In the future, it is necessary to experiment on more diverse examples including communication protocols and multithreaded programs to fully demonstrate its potential. Additionally, it is necessary to develop efficient approaches that make abstract counter-examples in the reduced SG be concrete by recovering the reduced information for better debugging.

## Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0930510 and 0930225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

1. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. Int. Conf. on Computer Aided Verification*, volume 3576 of *LNCS*, pages 548 – 562. Springer-Verlag, 2005.
2. 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.
3. M. Bobaru, C.S.Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Proc. Int. Conf. on Computer Aided Verification*, LNCS. Springer-Verlag, 2008.
4. 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.
5. 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.
6. S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transations on Software Engineering and Methodology*, 5(4):334–377, 1996.
7. S. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.


UU IR Author Manuscript

8. 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.
9. J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, pages 331–346. Springer-Verlag, 2003.
10. D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.
11. D. Giannakopoulou, C.S.Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, pages 297–320, 2005.
12. 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.
13. O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
14. 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.
15. G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 197–211, London, UK, UK, 1995. Chapman & Hall, Ltd.
16. 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.
17. 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.
18. K. L. Mcmillan. A methodology for hardware verification using compositional model checking. Technical report, Cadence Berkeley Labs, 1999.
19. C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.
20. W. Nam and R. Alur. Learning based symbolic assume-guarantee reasoning with automatic decomposition. In *Proc. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4218 of *LNCS*, 2006.
21. 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.
22. R. A. Thacker, K. R. Jones, C. J. Myers, and H. Zheng. Automatic abstraction for verification of cyber-physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 12–21, 2010.
23. H. Yao and H. Zheng. Automated interface refinement for compositional verification. *IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems*, 28(3):433–446, 2009.
24. T. Yoneda and T. Yoshikawa. Using partial orders for trace theoretic verification of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.
25. H. Zheng. Compositional reachability analysis for efficient modular verification of asynchronous designs. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 29(3), March 2010.


UU IR Author Manuscript

26. H. Zheng, J. Ahrens, and T. Xia. A compositional method with failure-preserving abstractions for asynchronous design verification. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 27, 2008.
27. H. Zheng, E. Mercer, and C. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Transactions on Computer-Aided Design*, 22(9):1138–1153, 2003.
28. 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.



UU IR Author Manuscript