# Compositional Model Checking of Concurrent Systems

Hao Zheng, *Senior Member, IEEE*, Zhen Zhang, *Member, IEEE*, Chris J. Myers, *Fellow, IEEE*, Emmanuel Rodriguez, and Yingying Zhang

***Abstract*— This paper presents a compositional framework to address the state explosion problem in model checking of concurrent systems. This framework takes as input a system model described as a network of communicating components in a high-level description language, finds the local state transition models for each individual component where local properties can be verified, and then iteratively reduces and composes the component state transition models to form a reduced global model for the entire system where global safety properties can be verified. The state space reductions used in this framework result in a reduced model that contains the exact same set of observably equivalent executions as 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 this compositional verification framework on a number of non-trivial concurrent system models.**

## I. Introduction

Model checking is a very effective approach to finding concurrency bugs compared against simulation or testing. However, the biggest hurdle for model checking of highly concurrent systems is the large number of interleavings among local executions, which causes state explosion. On the other hand, many of these interleavings are irrelevant to the properties under verification. Therefore, they should be avoided as much as possible to simplify the verification complexity.

This paper presents a compositional model checking framework that aims to verify complex highly concurrent systems. Fig. 1 shows the verification flow that is the basis of the framework presented in this paper. In this framework, it is assumed that a concurrent system under verification is described as a network of communicating components in some high-level description language. In the first step, the components are decoupled from each other and their state transition models are generated using a local state space construction method [41]. For each component, its state transition model preserves all essential interface interactions with its neighbors. Therefore, it includes all behaviors of this component as allowed by its neighboring components. Once the component state transition models are obtained, local properties defined for the individual components can be verified. However, the generated component state transition models may include extra behaviors that would not exist when the whole system is considered, so local verification may cause false negative results. These

Hao Zheng, Emmanual Rodriguez, and Yingying Zhang are with the CSE dept., University of South Florida, Tampa, FL 33620. Zhen Zhang and Chris Myers are with the ECE dept., University of Utah, Salt Lake City, UT 84112.

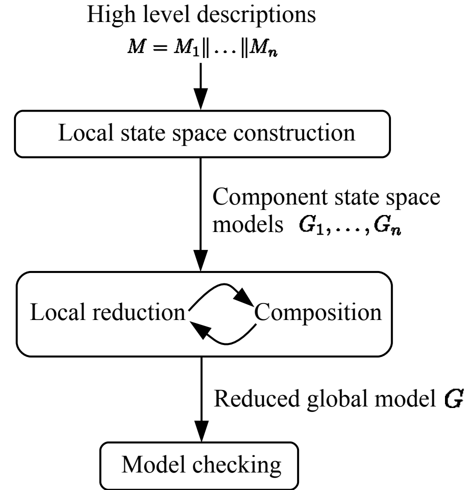


Fig. 1. The verification flow for the compositional framework.

false negatives are due to the inherent local nature of the component state space models. In the next step, the component state transition models are reduced and composed iteratively, and eventually a reduced global state transition model for the entire system is obtained where the model checking of global properties can be performed, and the negative results of the local properties can be discharged or confirmed. As the full global state space model for the whole system is never considered, much larger systems can be handled by this framework.

The key to the success of this framework is state space reduction. In some existing work [26], reduction is conservative in that more behavior may be introduced, but all essential behaviors are preserved during reduction. This is necessary since no real errors can be missed when verifying the reduced model. However, false errors may be introduced at the same time. When an error is found while verifying such a reduced model, it needs to be checked whether it is real on the concrete model. If reduction is too coarse, the number of false errors may become excessive, and checking these false errors can become the bottleneck. As shown later, this framework is integrated with a number of effective state space reductions [42] that can remove certain state transitions and states from a state transition model in such a way that the behavior of the model that is essential to verification remains the same. The reduced global state transition model produced at the end is equivalent to the concrete model of the whole system with respect to the properties under verification. This

method is sound and complete in that the reduced global model is verified to be correct if, and only if, the concrete model for the whole system is correct.

The reduction method presented in this paper is similar, in some degree, to the partial order reduction method [29] as both try to identify and remove certain transitions to eliminate stutter equivalent paths. Partial order reduction relies on the notion of transition dependencies to identify and remove redundant interleavings during state space search such that some stutter equivalent execution sequences are avoided, thus reducing the complexity of verifying the whole system. However, determining transition dependencies can be as difficult as computing the global state space. Therefore, they are often computed conservatively to ensure soundness of the verification results. This conservative computation causes partial order reduction to be less effective or even useless in some situations. On the other hand, our method can effectively remove all invisible transitions that correspond to stutter equivalent execution sequences as it considers the generated state space models where the necessary information is available for such reduction. Another difference is that partial order reduction is applied to the whole system, while this framework builds a reduced global state space model compositionally.

Although the individual pieces used in this framework have been presented separately in previous work [41], [42], the main contribution of this work is a framework that integrates them as a whole to support effective compositional model checking of global, as well as, local properties of large concurrent systems. This paper also presents the correctness proofs to show that this framework is sound and complete for safety properties. Finally, this framework is generalized to handle distributed algorithm and concurrent program models, in addition to asynchronous circuit designs.

This paper is organized as follows. Section II briefly reviews the necessary background for this paper. Section III describes a method for constructing local state space models from a high-level description of a system. Section IV presents a number of state space reductions that can remove all invisible state transitions and redundant states within the local state space models such that these models before and after reduction show the same behavior on the interface. This section also proves that the global state space model formed by composing the reduced local models shows the same interface behavior as the concrete global model. Section V demonstrates the effectiveness of this framework on a number of non-trivial concurrent system models including various mutual exclusion algorithms and some asynchronous circuit designs. The last section concludes the paper and points out some future work that can improve this method.

## II. Background

### A. System Modeling

This section presents a simple formalism to describe the behavior of concurrent systems. Let $V = (v_1, \ldots, v_n)$ be a finite set of variables that take their values from a single domain $Z \subset \mathbb{Z}$ where $\mathbb{Z}$ is the set of integers. Each $s \in Z^{|V|}$ is also referred to as a *state* over $V$.

This paper considers a finite system as a parallel composition of a number of components, $\|_{1 \le i \le n} M_i$, where each component is defined as follows.

*Definition 2.1:* (System Model) The model of a finite state component is a tuple $M_i = (V_i, init_i, A_i)$ where:

- $V_i$ is a finite set of variables,
- $init_i \in Z^{|V_i|}$ is the initial state,
- $A_i$ is a finite set of actions that define how states are changed when actions in $A_i$ are executed. Let $V_g \subseteq V_i$, $V_{supp} \subseteq V_i$, and $V_{upd} \subseteq V_i$. Each action $\alpha \in A_i$ is specified with $(g, va)$ where:
  - Guard $g : Z^{|V_g|} \to \{\mathsf{true}, \mathsf{false}\}$ maps an assignment over $V_g$ to a truth value.
  - Assignment $va : Z^{|V_{supp}|} \to Z^{|V_{upd}|}$ maps an assignment over $V_{supp}$ to an assignment over $V_{upd}$.

  Let $wr(\alpha)$ return $V_{upd}$, the set of variables that action $\alpha$ updates. ∎

Let $s(v)$ return the value of $v$ in state $s$. Two states $s_1$ and $s_2$ are consistent on a set of variables $C$, denoted as $s_1 \overset{C}{=} s_2$, iff $\forall v \in C, s_1(v) = s_2(v)$ holds.

An action $\alpha = (g, va)$ is enabled in a state $s$ if $g(s) = \mathsf{true}$. $enb(s)$ is used to denote all actions enabled in $s$. An action can be executed once it is enabled. The successor state $s'$ after executing an enabled action $\alpha$ is denoted by $\alpha(s)$. Executing action $\alpha$ updates variables in $wr(\alpha)$ while the variables in $V_i - wr(\alpha)$ remains unchanged, i.e. $s' \overset{V_i - wr(\alpha)}{=} s$.

This paper considers safety properties. Every component includes a special variable $safe_i \in V_i$. This variable is initialized to 1. It is reset to 0 when an action is executed if a safety condition is violated in a state. If $safe_i = 0$ in state $s$, $enb(s)$ is assumed to be empty. This can be done by conjoining a predicate $safe_i = 1$ to the guard of every action.

A large and complex system usually consists of components connected in a network where communications can be done through *shared variables*. Let $M_i = (V_i, init_i, A_i), 1 \le i \le n$, be $n$ components as defined in Definition 2.1. A concurrent system $\|_{1 \le i \le n} M_i$ is defined if the following condition holds,

$$\forall 1 \le i, j \le n, \left( init_i \overset{C_{ij}}{=} init_j \right) \wedge (A_i \cap A_j = \emptyset)$$

where $C_{ij} = V_i \cap V_j$. A $M_i$-state $s_i$ is referred to as a local state of $M_i$. A state of $M$ is referred to as a global state, which is a total assignment to all variables in $\cup_{1 \le i \le n} V_i$. A global state $s$ is defined by a $n-$tuple of local states $(s_1, \ldots, s_n)$ if

$$\forall 1 \le i, j \le n, s_i \overset{C_{ij}}{=} s_j,$$

and it is formed by merging variable assignments in all local states, denoted as $s = \Sigma(s_1, \ldots, s_n)$, such that:

$$\forall 1 \le i \le n, s(v) \overset{V_i}{=} s_i(v).$$

A concurrent system $\|_{1 \le i \le n} M_i$, if defined, is $M = (V, init, A)$ such that:

$$V = \cup_{1 \le i \le n} V_i, init = \Sigma(init_1, \ldots, init_n), A = \cup_{1 \le i \le n} A_i.$$

Consider a component $M_i = (V_i, init_i, A_i)$ in a concurrent system $M = \|_{1 \le i \le n} M_i$. Let $C_i = \cup_{i \ne j} C_{ij}$ be the set of variables of $M_i$ that are shared with some other components.

Therefore, $V_i - C_i$ is the set of invisible variables of $M_i$. A $M_i$-action $\alpha$ is visible if $wr(\alpha) \cap C_i \neq \emptyset$. Such an action is also referred to as $M_i$-visible. Let $A_i^{vis} = \{\alpha \in A_i \mid \alpha \text{ is visible}\}$ be the set of all visible actions of $M_i$. A $M_j$-action $\alpha$ is external to $M_i$ if $wr(\alpha) \cap V_i \neq \emptyset$. Predicate $ext(M_i, \alpha)$ is defined such that it holds if action $\alpha$ local to a different component is external to $M_i$. For each component $M_i$ in $\|_{1 \leq i \leq n} M_i$, the set of external actions is defined as follows:

$$Ext_i = \{\alpha \mid \forall j \neq i,\ \alpha \text{ is a } M_j\text{-action and } ext(M_i, \alpha)\}.$$

$Intf_i = A_i^{vis} \cup Ext_i$ is the set of interface actions that update the shared variables $C_i$ of component $M_i$. For $s' = \alpha(s)$, the following property holds.

$$s' \overset{V-C_i}{=} s \text{ if } \alpha \in (A_i - Intf_i)$$

Fig. 2 shows a simple example of a concurrent system with three components. In this example, variable $x$ is shared by $M_2$ and $M_3$, $y$ is shared by $M_1$ and $M_3$, and $z$ is shared by all three components. All the other variables are invisible in their respective components. $M_1$-actions $t_{11}$ and $t_{12}$ are invisible as they only modify the invisible variable $v$, while $M_1$-actions $t_{13}$ and $t_{14}$ are visible, and they are external to $M_3$. Similar information about the invisible and external actions can be derived for $M_2$ and $M_3$, as well.

### *B. State Graph*

This paper uses *state graphs* (SGs) to represent the state transition level semantics for concurrent systems.

*Definition 2.2:* (State Graph) A state graph for a concurrent system $M$ is a tuple $G = (S, \imath, R, F)$ where

1) $S$ is a finite non-empty set of states,
2) $\imath \in S$ is the initial state.
3) $R \subseteq S \times (A \cup Ext) \times S$ is the set of state transitions.
4) $F \subseteq S$ is a set of unsafe states. ∎

The SG definition is used to represent both concurrent systems and their components. In the above definition, $Ext$ is the set of external actions that the environment of $M$ can execute and change the shared variables of $M$. As shown later, the method presented in this paper needs to construct local state graphs for the components in a concurrent system. If a component $M_i$ in a system $M = \|_{1 \leq i \leq n} M_i$ communicates with some other components through shared variables, its local state graph contains not only the state transitions on local actions but also those on external actions to take into account updates on shared variables caused by actions executed in the other components that are external to $M_i$. Then, in $G_i = (S_i, \imath_i, R_i, F_i)$ for $M_i$, $S_i$ is the set of local states, $R_i$ is the set of state transitions on local actions in $A_i$ of $M_i$ or on actions in $Ext_i$ external to $M_i$, $\imath_i = init_i$, and $F_i = \{s_i \in S_i \mid s_i(safe_i) = 0\}$. The local SG for a communicating component $M_i$ in a system captures the behavior as defined in $M_i$, as well as, the updates on its shared variables by the visible actions of other components in the system. If $G = (S, \imath, R, F)$ is for a system $M = \|_{1 \leq i \leq n} M_i$, $Ext = \emptyset$, $S$ and $R$ are the set of global states and the set of global state transitions of $M$, respectively, $\imath = \Sigma(init_1, \ldots, init_n)$, and $F \subseteq S$ such that for each $s \in F$, if there exists $1 \leq i \leq n$ such that $s(safe_i) = 0$.

Fig. 3 shows the local state graphs for the components of the system shown in Fig. 2. The variable assignments for the local states of each component are shown to the left of the corresponding local state graph. Each local state graph in Fig. 3 includes some external transitions drawn with dotted lines. For example, in $G_1$, $(p_0, t_{34}, p_1)$ and $(p_3, t_{33}, p_4)$ are external as $t_{33}$ and $t_{34}$ are defined in $M_3$. When either action is executed, variable $z$ is changed, and this change results in state transitions in $M_1$, $M_2$, as well as, $M_3$ since variable $z$ is shared by all three components.

Executions of a concurrent system are represented by paths in SGs. Given a state graph $G = (S, \imath, R, F)$, a path of $G$ is a sequence $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ such that $\imath = s_0$ and $\forall i \geq 0,\ \exists \alpha\ s.t.\ (s_i, \alpha_i, s_{i+1}) \in R$. The trace of a path $\rho$, denoted as $tr(\rho)$, is $\alpha_0 \alpha_1, \ldots$. Given a zero length path which has no state transitions, its trace is $\epsilon$. In $G$, $s' = \alpha(s)$ if $(s, \alpha, s') \in R$. Similarly, state $s'$ is reachable from $s$ through a sequence of actions is denoted as $s' = \alpha_0 \ldots \alpha_n(s)$ if $\forall 0 \leq i \leq n,\ (s_i, \alpha_i, s_n) \in R$, $s_0 = s$ and $s_{n=} s'$. State $s$ is reachable in $G$ if $s$ is reachable from the initial state through a trace. A path is referred to as a *failure* if it leads to a state $s \in F$. The set of all failures in $G$ is denoted as $\mathcal{F}(G)$. The set of all the other paths in $G$ is denoted as $\mathcal{L}(G)$, and $\mathcal{L}(G) \cap \mathcal{F}(G) = \emptyset$.

Given a concurrent system $M = \|_{1 \leq i \leq n} M_i$, its state graph can be constructed using reachability analysis by executing exhaustively every enabled action in every state starting from the initial state. A general depth-first search algorithm for reachability analysis is shown in Algorithm 1. Checking safety properties can be done on-the-fly. Alternatively, a local state graph can be constructed for each component first, and then the global state graph of the entire system can be constructed by composing the local state graphs.

*Definition 2.3:* (Parallel Composition) Let $G_i = (S_i, \imath_i, R_i, F_i)$ be the local SGs for components $M_i$ in a system $\|_{1 \leq i \leq n} M_i$. Also, let $A = \cup_{1 \leq i \leq n} A_i$. The parallel composition $\|_{1 \leq i \leq n} G_i$ is defined as $G = (S, \imath, R, F)$ where

1) $S = \{\Sigma(s_1, \ldots, s_n)\}$ such that $\forall \Sigma(s_1, \ldots, s_n) \in S$, $\forall 1 \leq i, j \leq n,\ s_i \in S_i \wedge s_j \in S_j \wedge s_i \overset{V_i \cap V_j}{=} s_j$.
2) $\imath = \Sigma(\imath_1, \ldots, \imath_n) \in S$.
3) $R \subseteq S \times A \times S$ such that for each $(s, \alpha, s') \in R$, $s = \Sigma(s_1, \ldots, s_n) \notin F$ and $s' = \Sigma(s'_1, \ldots, s'_n)$, and for each $1 \leq i \leq n$,
   a) $(s_i, \alpha, s'_i) \in R_i$ if $\alpha$ is a $M_i-$action or $ext(M_i, \alpha)$,
   b) $s_i = s'_i$, otherwise.
4) $F = \{s \in S \mid \exists 1 \leq i \leq n\ s.t.\ s(safe_i) = 0\}$ ∎

In the above definition, when several processes execute concurrently, they synchronize on updates on shared variables, and proceed independently, otherwise. If any local state is not safe, the whole global state is regarded as not safe. The parallel composition is commutative and associative as the construction of composite states and composite state transitions is independent of order of composition. While the proof of this fact is straightforward by Definition 2.3, it is lengthy, so it is omitted. The global state graph for the whole system shown in Fig. 2 generated by Algorithm 1 or by the parallel composition of the state graphs shown in Fig. 3 is shown in Fig. 4.

$$
\begin{array}{rcl}
M_1 &=& (V_1, init_1, A_1);\\
\hline
V_1 &=& \{z, v, y\};\\
init_1 &=& (z=0, v=1, y=0);\\
A_1 &=& \{t_{11}, t_{12}, t_{13}, t_{14}\};\\
\text{where} && \\
t_{11} &=& (v=1 \wedge z=1, v:=0);\\
t_{12} &=& (v=0 \wedge z=0, v:=1);\\
t_{13} &=& (y=1 \wedge v=1, y:=0);\\
t_{14} &=& (y=0 \wedge v=0, y:=1);
\end{array}
\quad
\begin{array}{rcl}
M_2 &=& (V_2, init_2, A_2);\\
\hline
V_2 &=& \{z, w, x\};\\
init_2 &=& (z=0, w=1, x=0);\\
A_2 &=& \{t_{21}, t_{22}, t_{23}, t_{24}\};\\
\text{where} && \\
t_{21} &=& (w=1 \wedge z=1, w:=0);\\
t_{22} &=& (w=0 \wedge z=0, w:=1);\\
t_{23} &=& (x=1 \wedge w=1, x:=0);\\
t_{24} &=& (x=0 \wedge w=0, x:=1);
\end{array}
\quad
\begin{array}{rcl}
M_3 &=& (V_3, init_3, A_3);\\
\hline
V_3 &=& \{x, y, u, z\};\\
init_3 &=& (x=0, y=0, u=0, z=0);\\
A_3 &=& \{t_{31}, t_{32}, t_{33}, t_{34}\};\\
\text{where} && \\
t_{31} &=& (u=1 \wedge x=0 \wedge y=0, u:=0);\\
t_{32} &=& (u=0 \wedge x=1 \wedge y=1, u:=1);\\
t_{33} &=& (z=1 \wedge u=1, z:=0);\\
t_{34} &=& (z=0 \wedge u=0, z:=1);
\end{array}
$$

Fig. 2. An example of a simple concurrent system with three components communicating over shared variables $x$, $y$, and $z$.

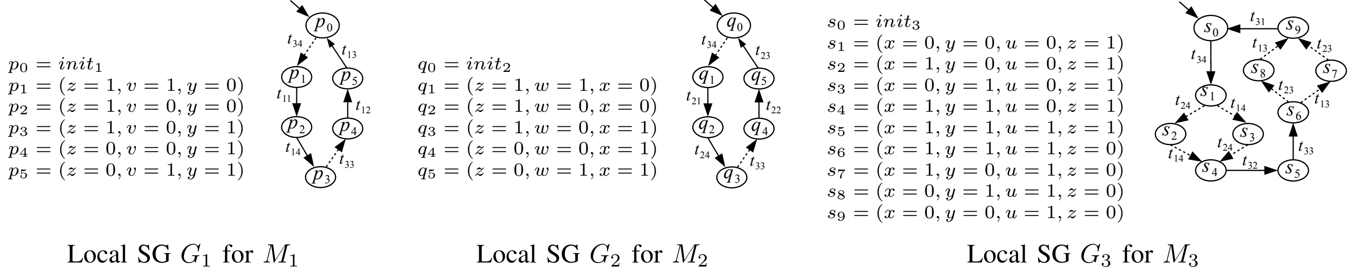


Fig. 3. The local state graphs and the state labelings for the components in the system shown in Fig. 2. State transitions drawn in dotted lines are due to the execution of the external transitions.

**Algorithm 1:** $DFS(\|_{1 \le i \le n} M_i)$

**Input**: A system description of $n$ components.
**Output**: A global state graph.

1 $\imath := \Sigma(init_1, \ldots, init_n)$;
2 $S := S \cup \imath$;
3 $stack.push((\imath, enb(\imath)))$;
4 **while** *stack is not empty* **do**
5 $\quad (s, E) := stack.top()$;
6 $\quad$ **if** $E = \emptyset$ **then**
7 $\quad\quad stack.pop()$;
8 $\quad\quad$ **continue**;
9 $\quad$ Select $\alpha \in E$ to execute, and remove it from $E$;
10 $\quad s' := \alpha(s)$;
11 $\quad$ **if** $\exists 1 \le i \le n \ s.t. \ s_i(safe_i) = 0$ **then**
12 $\quad\quad F := F \cup \{s\}$;
13 $\quad\quad$ **continue**;
14 $\quad R := R \cup \{(s, \alpha, s')\}$;
15 $\quad$ **if** $s' \notin S$ **then**
16 $\quad\quad stack.push((s', enb(s')))$;
17 $\quad\quad S := S \cup s'$;

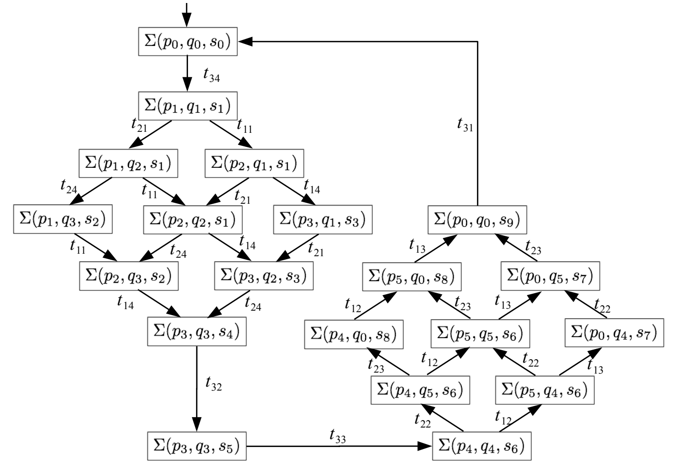


Fig. 4. The global state graph for the system as described in Fig. 2.

### C. State Graph Equivalence

The procedure described in this paper achieves its efficiency by applying reductions described in Section IV to produce a simpler, equivalent SG that represents the behavior of the same component in a system with less details. Two SGs, $G_i$ and $G'_i$, for the same component, $M_i$, are equivalent if when they are composed with a SG, $G_j$, for another component, $M_j$, failure paths exist in the composite with $G_i$ if and only if they exist in the composite with $G'_i$, and they have the same set of non-failure paths that are stutter equivalent. The rest of the section formalizes *state graph equivalence*.

First, it is necessary to introduce the concept of autofailure. Given a failure path of a component, it may enter the failure state through a sequence of transitions on local actions. However, the real cause of the failure can be traced back to a state resulting from an external action in the environment. This fact is because if the environment changes shared variables to a state in which it becomes possible for a failure to be reached by local actions, then no component exists that can be composed with this component that prevents this failure path from being possible. This situation is referred to as *autofailure manifestation* in [21]. Given a failure path $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$, if there exists $i \ge 0$ such that $\alpha_i$ is an external action, and for all $k > i$, $\alpha_k$ is a local action, then $s_0 \xrightarrow{\alpha_0} \ldots, s_i \xrightarrow{\alpha_i} s_{i+1}$ is the *autofailure prefix*, denoted as $Pref_{AF}(\rho)$. At this point in the trace $\rho$, there is no action by the environment that can prevent a failure from occurring. When $\rho$ is not a failure path, $Pref_{AF}(\rho) = \rho$. If $\rho$ does not have any external actions, $Pref_{AF}(\rho) = s_0$, and its trace, is empty, i.e. $\epsilon$. Such a path

is referred to as an *initial autofailure*. Note that all initial autofailures of a component are regarded as being equivalent.

In $G_i$ for a component $M_i$, $s' \stackrel{C_i}{=} s$ holds for $s$ and $s'$ in a state transition $(s, \alpha, s')$ if $\alpha \notin Intf_i$. Such state transitions are called *invisible*. Two paths are called *stutter equivalent* if they only differ in their corresponding invisible state transitions [22]. Two failure paths $\rho$ and $\rho'$ are stutter equivalent if $Pref_{AF}(\rho)$ is stutter equivalent to $Pref_{AF}(\rho')$. These ideas are formalized in the definition below.

*Definition 2.4 (Stutter Equivalence):* Let $G_i$ and $G'_i$ be two SGs for $M_i$, and $\rho \in \mathcal{L}(G_i) \cup \mathcal{F}(G_i)$ and $\rho' \in \mathcal{L}(G'_i) \cup \mathcal{F}(G'_i)$ be two paths, respectively. $\rho$ and $\rho'$ are stutter equivalent, denoted as $\rho \approx \rho'$, if

$$tr(Pref_{AF}(\rho))_{/Intf_i} = tr(Pref_{AF}(\rho'))_{/Intf_i}. \quad \blacksquare$$

Note that in the definition above $t_{/X}$ denotes the *projection* of the trace $t = \alpha_0\alpha_1 \ldots$ over a set of actions $X$, and it returns another trace where all actions not in $X$ are removed and all other actions remain in place.

Suppose $\rho'$ is a failure path in $G'_i$. If $\rho'$ is not an autofailure, and there is a path $\rho$ in $G_i$ such that it has a prefix that is stutter equivalent to the autofailure prefix of $\rho'$, then $\rho$ is also regarded as a failure. In such a case, path $\rho'$ is referred to as a *failure approximation* to $\rho$.

*Definition 2.5 (Failure Approximation):* Let $G_i$ and $G'_i$ be two SGs for $M_i$, and $\rho \in \mathcal{L}(G_i) \cup \mathcal{F}(G_i)$ and $\rho' \in \mathcal{F}(G'_i)$ be two paths. $\rho'$ is a failure approximation to $\rho$, denoted as $\rho \approx_{F_i} \rho'$, if

$$tr(\rho')_{/Intf_i} \text{ is a prefix of } tr(\rho)_{/Intf_i}. \quad \blacksquare$$

The equivalence of two state graphs $G_i$ and $G'_i$ for a single component $M_i$ can now be defined as follows:

*Definition 2.6 (State Graph Equivalence):* $G_i$ and $G'_i$ are two equivalent SGs for $M_i$, denoted as $G_i \approx G'_i$, if all the following conditions hold.

$$\begin{cases} \forall \rho \in \mathcal{F}(G_i), \exists \rho' \in \mathcal{F}(G'_i) \; s.t. \; \rho \approx_{F_i} \rho' \text{ and} \\ \forall \rho' \in \mathcal{F}(G'_i), \exists \rho \in \mathcal{F}(G_i) \; s.t. \; \rho \approx \rho' \end{cases} \quad (1)$$

and

$$\begin{cases} \forall \rho \in \mathcal{P}_i, \exists \rho' \in \mathcal{P}'_i \; s.t. \; \rho \approx \rho' \text{ and} \\ \forall \rho' \in \mathcal{P}'_i, \exists \rho \in \mathcal{P}_i \; s.t. \; \rho \approx \rho' \end{cases} \quad (2)$$

where

$$\mathcal{P}_i = \mathcal{L}(G_i) - \{\rho \in \mathcal{L}(G_i) \mid \exists \rho' \in \mathcal{F}(G'_i) \; s.t. \; \rho \approx_{F_i} \rho'\}$$
$$\mathcal{P}'_i = \mathcal{L}(G'_i) - \{\rho \in \mathcal{L}(G'_i) \mid \exists \rho' \in \mathcal{F}(G'_i) \; s.t. \; \rho \approx_{F_i} \rho'\} \quad \blacksquare$$

According to the above definition, for two SGs $G_i$ and $G'_i$, $\mathcal{F}(G_i) = \emptyset$ iff $\mathcal{F}(G'_i) = \emptyset$, and all their non-failure paths that do not have failure approximations in $\mathcal{F}(G'_i)$ are stutter equivalent. The reason that paths with failure approximations in $\mathcal{F}(G'_i)$ are not considered in the definition is that these paths are redundant in terms of finding failures.

Next, the preservation of the SG equivalence by the parallel composition is considered. Consider $G_1 \| G_2$ for $M_1 \| M_2$. If there is another SG $G'_1$ such that $G_1 \approx G'_1$, then the following property shows that the behavior of the composition of two SGs remains unchanged if one of the SGs is replaced with another equivalent one.

$$G_1 \| G_2 \approx G'_1 \| G_2. \quad (3)$$

The complete proof for Property 3 involves constructing paths in $G_1 \| G_2$ and $G'_1 \| G_2$ from the paths in $G_1$, $G'_1$ and $G_2$, respectively, by following Definition 2.3, and showing that they are stutter equivalent or one is a failure approximation to the other. It is mechanical and long, and omitted due to the page limit. Instead, this property can be intuitively understood with the following argument.

Let $A_{12} = (A_1 \cap Ext_2) \cup (A_2 \cap Ext_1)$. Also, let $\rho_1 \in \mathcal{F}(G_1)$ and $\rho'_1 \in \mathcal{F}(G'_1)$ such that $\rho_1 \approx_{F_1} \rho'_1$. Note that this step always succeeds as $G_1 \approx G'_1$. Also, let $\rho_2 \in \mathcal{L}(G_2) \cup \mathcal{F}(G_2)$ such that $tr(\rho_1)_{/A_{12}} = tr(\rho_2)_{/A_{12}}$ and $tr(\rho'_1)_{/A_{12}}$ is a prefix of $tr(\rho_2)_{/A_{12}}$. Compose $\rho_1$ and $\rho_2$ first. If there are two transitions on $\rho_1$ and $\rho_2$, respectively, in a composite state during the composition such that their actions are not in $A_{12}$, one is selected to build the composite state transition and the next composite state. Remember the order of all selections made when composing $\rho_1$ and $\rho_2$. Let $\rho$ be the resulting path from composing $\rho_1$ and $\rho_2$ following that order of action selections. Then, apply the same order of action selections when $\rho'_1$ and $\rho_2$ are composed, and let $\rho'$ be the resulting path. It can be seen that $\rho$ and $\rho'$ have the same trace on $Intf_1 \cup A_2$ up to the last action in $tr(\rho'_1)$. Since $\Sigma(\imath_1, \imath_2) = \Sigma(\imath'_1, \imath_2)$, $\rho'$ is a failure approximation to $\rho$ by Definition 2.5. The same conclusion can be drawn for the above two compositions with all possible orders of action selections. Apply the same argument to all paths $\rho_1 \in \mathcal{F}(G_1)$ and $\rho'_1 \in \mathcal{F}(G'_1)$ and to all paths in $\mathcal{F}(G_2)$, it can be seen that Condition 1 in Definition 2.6 holds.

Similarly, apply the same argument to paths in $\mathcal{L}(G_1)$ and $\mathcal{L}(G'_1)$, and to the paths in $\mathcal{L}(G_2)$, and Condition 2 in Definition 2.6 can be shown to hold. Therefore, Property 3 can be shown to hold by Definition 2.6.

## III. Local State Graph Construction

This section describes an approach for constructing local state graphs for a concurrent system. This approach reduces complexity by not considering interleavings of the invisible state transitions from different components. This paper generalizes the method described in [41] for asynchronous circuit verification to the system models and state graphs in Section II.

Consider a concurrent system $M = \|_{1 \le i \le n} M_i$. The local state graph construction method builds a SG $G_i$ for each component $M_i$ from an empty context, and gradually expands it by including all states and state transitions allowed by its neighboring components. The main idea is as follows. Initially, the value of the shared variables that each component depends on is fixed to be what is defined in the initial state. Then, this method iteratively performs the following two steps.

1) For every individual component, ignore the changes on the shared variables caused by other components, and use the standard state space search method such as the one shown in Algorithm 1 to find all states and state transitions as defined.
2) For every two local state graphs, $G_i$ and $G_j$, if the above step finds a new state transition $(s_j, \alpha, s'_j)$ in $G_j$ such that $ext(M_i, \alpha)$ holds, then a state transition on $\alpha$ reflecting the changes on the shared variables as in

**Algorithm 2:** $findLocalSG(M_1, \ldots, M_n)$

**Input**: $M_i = (V_i, init_i, A_i), 1 \leq i \leq n$: components.
**Output**: $G_i$ $(1 \leq i \leq n)$: local state graph for $M_i$.

```
1  foreach M_i do
2  |  Create a SG G_i := DFS(M_i);
3  |  new_i := true;
4  while ⋁_{1≤i≤n} new_i do
5  |  foreach 1 ≤ i ≤ n do
6  |  |  new'_i := false;
7  |  |  foreach 1 ≤ j ≤ n s.t. i ≠ j do
8  |  |  |  if ((new_i ∨ new_j) ∧ ∃α ∈ A_j s.t. ext(M_i, α))
   |  |  |  then
9  |  |  |  |  expand(M_i, G_i, G_j);
10 |  |  |  |  if new transitions are added into G_i then
11 |  |  |  |  |  new'_i := true;
12 |  foreach 1 ≤ i ≤ n do
13 |  |  new_i := new'_i;
```

**Algorithm 3:** $expand(M_i, G_i, G_j)$

**Input**: $M_i = (V_i, init_i, A_i)$.
**Input**: $G_\chi = (S_\chi, init_\chi, R_\chi), \chi \in \{i, j\}$.
**Output**: $G_i$ expanded with external transitions from $G_j$.

```
1  C := V_i ∩ V_j;
2  foreach (s_j, α, s'_j) ∈ R_j ∧ α ∈ A_j ∧ ext(M_i, α) do
3  |  foreach s_i ∈ S_i do
4  |  |  if s_i =^C s_j then
5  |  |  |  Create a new state s'_i such that s'_i =^C s'_j ∧
   |  |  |      s_i =^{V_i−C} s'_i ∧ s'_i(safe_i) = s'_j(safe_j);
6  |  |  |  R_i := R_i ∪ (s_i, α, s'_i);
7  |  |  |  if s'_i ∉ S_i then
8  |  |  |  |  S_i := S_i ∪ s'_i;
9  |  |  |  |  if s'_i(safe_i) = 0 then
10 |  |  |  |  |  F_i := F_i ∪ {s'_i};
11 |  |  |  |  else
12 |  |  |  |  |  DFS(M_i, G_i, s'_i);
```

$(s_j, \alpha, s'_j)$ is added to $G_i$. Adding these external state transitions into $G_i$ may introduce new states and state transitions. Then, apply step 1 again.

The above two tasks are performed repeatedly until no new state transitions can be added to any local state graph.

Algorithm 2 shows this method at the top level where the input is the set of components of a concurrent system, and the output is the set of the generated local state graphs. Initially, a local state graph $G_i$ is generated for each individual component $M_i$ using the DFS as shown in Algorithm 1 without considering how other components might change the shared variables (line 2). Each $G_i$ is also assigned with a variable $new_i$ to indicate if it is expanded with new state transitions (line 3). As long as any $new_i$ is true, the algorithm repeats the steps in line $4 - 15$. For every $G_i$ and $G_j$ such that $G_j$ includes state transitions that are external to $G_i$ and some new state transitions are added into at least one of them, function $expand(G_i, G_j)$ is called to factor the changes on the shared variables from $G_j$ into $G_i$. If $G_i$ is expanded with new state transitions, variable $new'_i$ is set to true to prepare for the next iteration.

Function $expand$ is defined in Algorithm 3. It takes as input a component, $M_i$ and two local state graphs, $G_i$ and $G_j$, and considers every state transition $(s_j, \alpha, s'_j)$ in $G_j$ such that $\alpha$ is external to $G_i$, and it is checked against every state $s_i$ in $G_i$. If $s_i$ and $s_j$ are consistent on their shared variables $C = V_i \cap V_j$, a new state $s'_i$ and a new external state transition $(s_i, \alpha, s'_i)$ are added into $G_i$ such that $s'_i \stackrel{C}{=} s'_j$ while $s'_i \stackrel{V_i-C}{=} s_i$. The rationale behind this operation is as follows. The state of the invisible variables of $G_j$ is not visible to $G_i$, and from the point of view of $G_i$, it only knows that whenever the values of the shared variables in $C$ are those in $s_j$, their values may be changed to those in $s'_j$ after $\alpha$ is executed in $G_j$. The changes on the shared variables are carried over to $G_i$ as external state transitions, which may cause some new states to be added into $G_i$. If so, a state space search procedure modified from Algorithm 1 is applied to $G_i$ to find all reachable states and state transitions from these new states, and add them into $G_i$. This procedure, $DFS(M_i, G_i, s'_i)$, takes as input a component description, $M_i$, its state graph, $G_i$, and a state, $s'_i$, performs the standard depth-first search, and expands $G_i$ with new reachable states and state transitions from $s'_i$. Since this algorithm is very similar to Algorithm 1, it is omitted.

Now consider an example where the above algorithms are applied to build the local state graphs for the simple concurrent system as shown in Fig 2. The partial local state graphs generated during the course of applying these algorithms are shown in Fig. 5. The local state graphs in Fig. 5(a) are the results of applying Algorithm 1 to individual components ignoring all other components at the beginning of Algorithm 2. In the first iteration, since $G_1$ and $G_2$ have transitions external to $G_3$, and vice versa, Algorithm 3 is applied to pairs consisting of $G_3$ and either $G_1$ or $G_2$. $expand(G_3, G_1)$ and $expand(G_3, G_2)$ have no effect on $G_3$ as $G_1$ and $G_2$ do not have any state transitions. $expand(G_1, G_3)$ and $expand(G_2, G_3)$ add an external state transition on $t_{34}$ to $G_1$ and $G_2$, respectively. These external state transitions subsequently result in more local states added into $G_1$ and $G_2$. The partial local state graphs after the first iteration are shown in Fig. 5(b). In the next iteration, $expand(G_3, G_1)$ and $expand(G_3, G_2)$ add more external state transitions into $G_3$, and subsequently more states are found for $G_3$ as a result. The partial local state graphs after the second iteration are shown in Fig. 5(c). After two more iterations, every local state graph reaches a fixpoint, and the algorithms terminate. The final local state graphs are shown in Fig. 3.

The following lemma shows that the global SG as the parallel composition of the local state graphs, $G_i$, constructed with Algorithm 2 for $\|_{1 \leq i \leq n} M_i$ is the same as the global SG constructed directly from $\|_{1 \leq i \leq n} M_i$ with Algorithm 1.

*Lemma 3.1:* Let $M = \|_{1 \leq i \leq n} M_i$ be a concurrent system, and $G = DFS(M)$. If $G_i$ are local state graphs constructed for $M_i$ with Algorithm 2, then the following statement is true:

$$G = \|_{1 \leq i \leq n} G_i.$$

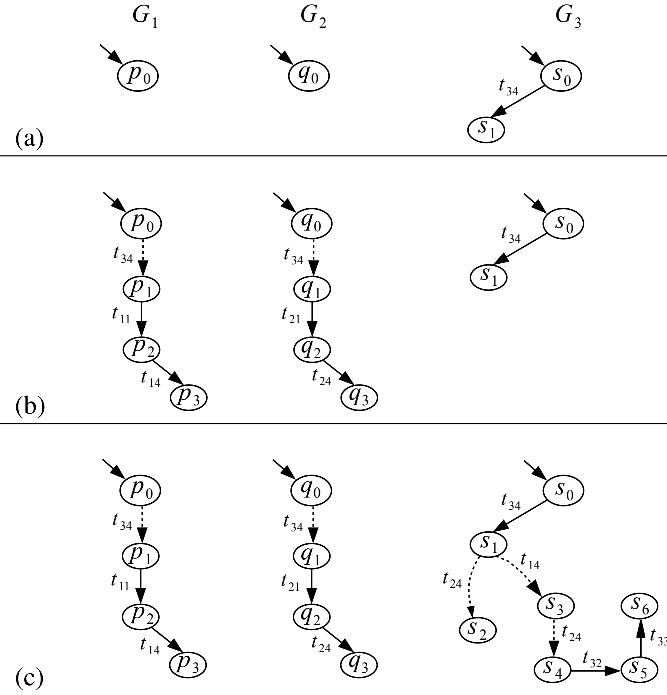


Fig. 5. Figures in (a)-(c) show the partial local state graphs generated in the first few steps when running Algorithm 2.

**Proof**: Let $G' = \|_{1\leq i\leq n} G_i$ and $V = \cup_{1\leq i\leq n} V_i$. Obviously, $\imath = \imath'$ as both are $\Sigma(\imath_1, \ldots, \imath_n)$.

Consider an arbitrary state transition $(\imath, \alpha, s)$ in $G$ such that $s = \alpha(\imath)$ in $G$. Suppose $\alpha$ is a $M_i$-action. Then, $s \overset{V_i}{=} \alpha(\imath)$ and $s \overset{V-V_i}{=} \imath$. We need to show that state transition $(\imath', \alpha, s')$ such that $s' \overset{V_i}{=} \alpha(\imath')$ and $s' \overset{V-V_i}{=} \imath'$ exists in $G'$. Since $\alpha$ is a $M_i$-action, $(\imath_i, \alpha, s'_i)$ such that $s'_i = \alpha(\imath_i)$ exists in $G_i$. Then, for every $1 \leq i \leq n$ such that $i \neq j$,

- If $ext(M_j, \alpha)$ holds, $(\imath'_j, \alpha, s'_j)$ exists in $G_j$ by Algorithm 3 such that $s'_j \overset{V_i\cap V_j}{=} s'_i \wedge s'_j \overset{V_j-V_i}{=} \imath'_j$.
- If $ext(M_j, \alpha)$ does not hold, $s'_j = \imath'_j$.

By Definition 2.3, a state $s' = \Sigma(s'_1, \ldots, s'_n)$ and a state transition $(\imath', \alpha, s')$ are added into $S'$ and $R'$ of $G'$ such that

$$s' \overset{V_i}{=} \alpha(\imath') \wedge s' \overset{V-V_i}{=} \imath'.$$

This implies $s = s'$.

Therefore, for every action $\alpha$, $R(\imath, \alpha, s)$ holds iff $R(\imath', \alpha, s')$ holds such that $s = s'$. Add $s$ into $S$, and $s'$ into $S'$. Also note that $s_i(safe_i) = 0$ iff $s'_i(safe_i) = 0$. Therefore, $s$ is added into $F$ iff $s'$ is added into $F'$.

Then, for each pair of such successor states $s$ and $s'$ of $\imath$ and $\imath'$, respectively, we can show that $s$ and $s'$ have the same set of outgoing state transitions and the same set of successor states based on the similar reasoning for the initial states. By applying such reasoning recursively on the new pairs of states, eventually, we can show that $S = S'$, $\imath = \imath'$, $R = R'$, and $F = F'$, thus $G = G'$. ∎

## IV. State Graph Reductions

The state graph of a system can be obtained by composing the local state graphs after they are constructed as shown in the last section. However, directly composing the local state graphs defeats the purpose of compositional construction in that the interleaving of the invisible state transitions can cause the state space to explode quickly during the parallel composition. To address this problem, this section presents several state graph reductions to simplify the local state graphs before they are composed in order to control the complexity. It is shown that the reduced state graphs are equivalent to the original ones. This implies that any safety properties hold or fail in the reduced state graphs if, and only if, they hold or fail in the original ones without using the reductions.

**Algorithm 4:** $\texttt{afr}(G_i)$

**Input**: An SG $G_i = (S_i, \imath_i, R_i, F_i)$ for component $M_i = (V_i, init_i, A_i)$.
**Output**: $G_i$ after autofailure reduction.

1 **foreach** $bad \in F_i$ **do**
2  **foreach** $(s_1, \alpha, s_2) \in R_i$ *such that* $s_2 = bad$ **do**
3   **if** $s_1 = \imath_i$ and $\alpha$ *is a* $M_i$*-action* **then**
4    **return** $(\{\imath_i\}, \imath_i, \emptyset, \{\imath_i\})$;
5   **if** $\alpha$ *is a* $M_i$*-action* **then**
6    $R_i := R_i - (s_1, \alpha, s_2)$;
7    $s_1(safe_i) := 0$;
8    $F_i := F_i \cup \{s_1\}$;
9    Remove unreachable states and transitions;

### A. Autofailure Reduction

Autofailure reduction is to replace a failure trace, $\rho$, with a new failure trace $Pref_{AF}(\rho)$ where the local actions preceding the failure state are removed. The idea of autofailure reduction is introduced in [21], but it is only used to canonicalize trace structures for hierarchical verification. Autofailure reduction is first adopted as part of an interface refinement method for modular model checking in [39]. In this paper, it is integrated with other reductions into this framework.

Algorithm 4 shows how autofailure reduction works. For each state transition $(s_1, \alpha, s_2)$ such that $s_2$ is a failure state, it is skipped if $\alpha$ is an external action; otherwise, it is removed, and $s_1$ is changed to a failure state. The algorithm repeats until all transitions entering the set of failure states are on external actions, or a transition $(s_1, \alpha, s_2)$ is encountered where $s_1$ is the initial state and $\alpha$ is a local action. In the latter case, an empty state graph with a single initial failure state is returned to indicate that the corresponding component can fail irrespective of how the environment behaves.

The following lemma shows that the state graph after the autofailure reduction is equivalent to the original one.

*Lemma 4.1:* Let $G_i$ be a local state graph for component $M_i$ in $\|_{1\leq i\leq n} M_i$. Then, the following statement holds.

$$G_i \approx \texttt{afr}(G_i)$$

**Proof**: $\texttt{afr}(G_i) = G_i$ if $F_i$ of $G_i$ is empty or all state transitions in $R_i$ that enter $F_i$ are on external actions. Therefore, $G_i \approx \texttt{afr}(G_i)$.

Now, suppose that $G_i$ contains $(s_1, \alpha, s_2)$ such that $\alpha$ is a $M_i$-action, and $s_2 \in F_i$. Let $\rho$ be a failure path in $G_i$ such

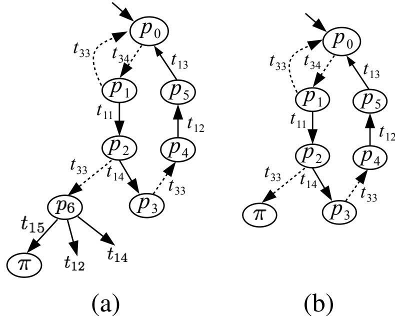


Fig. 6. (a) The state graph for the modified $M_1$ in Fig. 2, (b) The resulting state graph after autofailure reduction.

that $tr(\rho) = \alpha_0 \ldots \alpha_n$ and $s_n \in F_i$. Suppose that all actions $\alpha_i$ in $tr(\rho)$ such that $k+1 \leq i \leq n$ ($k \in \{0\} \cup \mathbb{Z}^+$ and $0 \leq k \leq n$) are $M_i$-actions. With Algorithm 4, $\rho$ can reduce to become $\rho'$ such that $tr(\rho') = \alpha_0 \ldots \alpha_k$ and $s_k \in F_i$. This shows that for every $\rho \in \mathcal{F}(G_i)$, there is a $\rho' \in \mathcal{F}(\mathtt{afr}(G_i))$ such that $\rho \approx_{F_i} \rho'$. From the above discussion, it can be seen that for every such $\rho' \in \mathcal{F}(\mathtt{afr}(G_i))$, there is a $\rho \in \mathcal{F}(G_i)$ such that $Pref_A(\rho) = \rho'$. Therefore, this proves Condition (1) in Definition 2.6.

Next, consider every path $\rho \in \mathcal{L}(G_i)$ that does not have a failure approximation in $\mathcal{F}(\mathtt{afr}(G_i))$. No state on $\rho$ is converted to a failure state by Algorithm 4, therefore $\rho$ exists in $\mathtt{afr}(G_i)$. Since every non-failure path in $\mathtt{afr}(G_i)$ exists in $G_i$, this shows that Condition (2) in Definition 2.6 holds. Hence, $G_i \approx \mathtt{afr}(G_i)$. ∎

As an example, consider a modified version of the model in Fig. 2 that allows action $t_{33}$ in $M_3$ to execute right after $t_{34}$, and adding an action $t_{15}$ into $M_1$ such that it causes an assertion failure when state $(z = 0, v = 0, y = 0)$ is reached in $M_1$. The partial state graph for the modified $M_1$ is shown in Fig. 6(a). The state labeled with $\pi$ is a failure state. In this state graph, when $t_{33}$ is executed in state $p_2$, a new state $p_6 = (z = 0, v = 0, y = 0)$ is reached where three actions, $t_{12}$, $t_{14}$ and $t_{15}$, are enabled. After executing $t_{15}$, a failure state is reached. Actually, the root cause to this failure is the execution of $t_{33}$ in state $p_2$. After autofailure reduction, $p_6$ is converted to a failure state, while states reachable from $p_6$ are removed. The reduced state graph is shown in Fig. 6(b).

### B. Stutter Equivalent Reduction

The interleaving of invisible state transitions in different local state graphs is the major source for state explosion during parallel composition. The traditional abstraction techniques collapse the invisible state transitions into single states [12]. This often introduces extra paths and false failures. To address this problem, this paper presents the *stutter equivalent reduction*. The basic idea is to remove all invisible state transitions in a state graph while maintaining the same set of stutter equivalent paths with respect to its interface. As a result, it produces a stutter equivalent path with only visible transitions for each path in a state graph. Alternatively, it can be viewed as a step of shortening a given path by passing over the invisible transitions, but it does not introduce any new paths. Therefore, no false failures can be created.

**Algorithm 5:** $\mathtt{ser}\ (G_i)$

1 **foreach** $(s_1, \alpha_1, s_2) \in R_i$ *s.t.* $\alpha_1 \notin Intf_i$ **do**
2 $\quad R_i := R_i - \{(s_1, \alpha_1, s_2)\}$;
3 $\quad$ **if** $s_2 \in F_i$ **then**
4 $\quad\quad F_i := F_i \cup \{s_1\}$;
5 $\quad$ **else**
6 $\quad\quad \mathtt{reducePath}(G_i, s_1, s_2)$;
7 Remove all invisible state transitions from $G_i$;
8 Remove unreachable states and state transitions from $G_i$;

**Algorithm 6:** $\mathtt{reducePath}\ (G_i, s_1, s_2)$

1 **foreach** $(s_2, \alpha_2, s_3) \in R$ **do**
2 $\quad$ **if** $\alpha_2 \in Intf_i$ **then**
3 $\quad\quad R_i := R_i \cup \{(s_1, \alpha_2, s_3)\}$;
4 $\quad$ **else**
5 $\quad\quad R_i := R_i - \{(s_2, \alpha_2, s_3)\}$;
6 $\quad\quad$ **if** $s_3 \in F_i$ **then**
7 $\quad\quad\quad F_i := F_i \cup \{s_1\}$;
8 $\quad\quad$ **else**
9 $\quad\quad\quad \mathtt{reducePath}(G, s_1, s_3)$;

Algorithm 5 shows the top level procedure $\mathtt{ser}(G_i)$ for stutter equivalent reduction on a state graph $G_i$. For each invisible state transition $(s_1, \alpha_1, s_2)$, it searches forward from $s_2$ following invisible state transitions in a depth-first manner until a visible transition or a failure state $\pi$ is encountered. If a failure state is encountered after a sequence of invisible state transitions are traversed, $s_1$ is converted to a failure state. Otherwise, Algorithm 6 creates a new visible transition to replace the sequences of invisible state transitions traversed, and it is added into $R_i$. After all invisible transitions are handled, they are removed from $G_i$. Consequently, some other states and transitions may become unreachable, and are also removed.

Fig. 7 shows an example how a SG in Fig. 7(a) is reduced by stutter equivalent reduction to become the one shown in Fig. 7(b). In this example, suppose all invisible transitions are labeled by $\zeta$. Then, for each visible transition from states $s_{i+1}$, $s_{j+1}$, and $s_{k+1}$, a new transition is created for states $s_i$, $s_j$, and $s_k$, respectively. Six new state transitions are added to preserve the same visible 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, algorithms are described to identify and remove redundancies in the reduced SGs.

The following lemma asserts that a SG and the one resulting from the stutter equivalent reduction are equivalent.

*Lemma 4.2:* Given a state graph $G_i$, $G_i \approx \mathtt{ser}(G_i)$.

**Proof**: The proof is based on how procedure $\mathtt{ser}(G_i)$ works. It is straightforward to see that for every path $\rho$ in $G_i$ that does not include any invisible transitions, the same path also exists in $\mathtt{ser}(G_i)$. For a path $\rho = \ldots s_i \xrightarrow{\alpha_i} s_{i+1} \xrightarrow{\alpha_{i+1}} s_{i+2} \ldots$ such that $\alpha_i$ is invisible, then there exists a path $\rho' = \ldots s_i \xrightarrow{\alpha_{i+1}} s_{i+2} \ldots$ in $\mathtt{ser}(G_i)$, and $\rho \approx \rho'$.

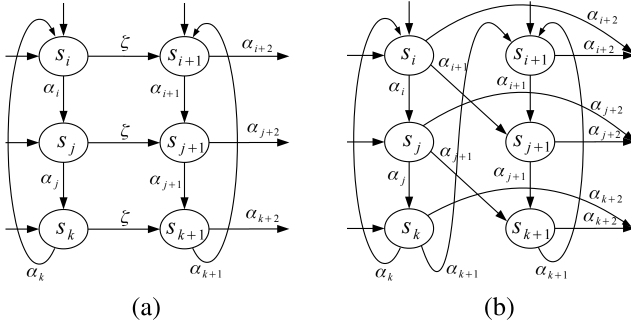


Fig. 7. In the figures, invisible transitions are labeled with $\zeta$. (a) An example SG with invisible state transitions. (b) The SG from (a) after the observably equivalent reduction handles an invisible transition $(s_i, s_{i+1})$. (c) The SG from (a) after all invisible transitions are handled and removed.

Conversely, for every path $\rho' = \ldots s_i \xrightarrow{\alpha_{i+1}} s_{i+2} \ldots$ in $\texttt{ser}(G_i)$, either the same path exists in $G_i$, or it is reduced from a path $\rho = \ldots s_i \xrightarrow{\alpha_i} s_{i+1} \xrightarrow{\alpha_{i+1}} s_{i+2} \ldots$ in $G_i$ such that $\alpha_i$ is invisible, and it holds that $\rho \approx \rho'$. ∎

### *C. Failure Equivalent Reduction*

The stutter equivalent reduction in the last section can introduce nondeterminism. Nondeterminism exists if there are two state transitions $(s, \alpha_1, s_1)$ and $(s, \alpha_2, s_2)$ such that $\alpha_1 = \alpha_2$ and $s_1 \neq s_2$. This is a result from the reduction while preserving the equivalence, and often leads to redundancy. Therefore, removing such redundancy can simplify the complexity of the state graphs.

This section considers a simple form of redundancy due to nondeterminism based on the following understanding: if the same action executed in a state may or may not cause a failure nondeterministically, it is always regarded as causing a failure. It is formalized in the following definition.

*Definition 4.1:* Let $(s, \alpha_1, s_1)$ and $(s, \alpha_2, s_2)$ be two state transitions in a state graph $G$ such that $s_2 \in F$ and $s_1 \neq s_2$. $(s, \alpha_1, s_1)$ is failure equivalent to $(s, \alpha_2, s_2)$ if $\alpha_1 = \alpha_2$.

Failure equivalent transitions are redundant in that their existence does not affect the verification results, therefore, they can simply be removed. Consequently, all the resulting unreachable states are also removed, leading to more reductions.

Let $\texttt{fer}(G)$ be a procedure to remove failure equivalent transitions in $G$. The following lemma asserts that the reduced state graph is equivalent to the original one.

*Lemma 4.3:* Given a state graph $G$, $G \approx \texttt{fer}(G)$.

The proof is obvious, and not shown due to the page limit.

Fig. 8 shows an example of failure equivalent transitions. Fig. 8(a) is an example state graph. After the stutter equivalent reduction, there are two nondeterministic state transitions in the resulting state graph: $(s_j, \alpha_j, s_k)$ and $(s_j, \alpha_j, \pi)$. $(s_j, \alpha_j, s_k)$ is redundant as it is failure equivalent to $(s_j, \alpha_j, s_k)$ . After it is removed by the failure equivalent reduction, $s_k$ becomes unreachable, and is also removed. The reduced state graph is shown in Fig. 8(b).

### *D. Bisimulation Equivalent Reduction*

The stutter equivalent reduction removes all invisible state transitions from state graphs, therefore all paths in the reduced state graphs are stutter free. The reduced state graph often contains a lot of redundancies. The failure equivalent reduction described in the last section only handles a special case of such redundancies. This section considers removing redundant states in general. Two states are redundant if every path from one state has a stutter equivalent path from the other state, and vice versa.

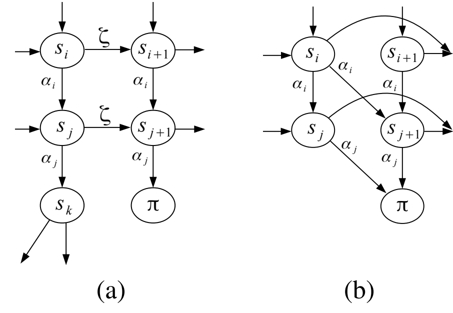


Fig. 8. (a) An example state graph with a failure state. (b) The state graph from (a) after the stutter equivalent and the failure equivalent reductions with the unreachable state $s_i$ removed.

**Algorithm 7:** $\texttt{Reduce}$ $(G)$

```
1 G = afr(G);
2 G = ser(G);
3 G = fer(G);
4 G = eqr(G);
5 return G;
```

To remove all redundant states, we use an algorithm to compute the bisimulation quotient of the state graphs as described in [3], and this reduction is referred to as $\texttt{eqr}(G)$ in this paper. The following lemma is the direct result of the well known fact that a state transition model and its bisimulation quotient are path equivalent.

*Lemma 4.4:* Given a state graph $G$, and let $\texttt{eqr}(G)$ return the bisimulation quotient of $G$. $G \approx \texttt{eqr}(G)$.

**Proof**: Directly from Lemma 7.6 in [3]. ∎

### *E. Overall Reduction*

All these reductions are integrated into a single reduction function $\texttt{Reduce}(G)$ in Algorithm 7 which is used in compositional minimization. In function $\texttt{Reduce}(G)$, the autofailure reduction is applied first as it can remove a lot of states and state transitions while preserving all failure paths. This may lead to a lower complexity in state graphs, making the following steps simpler. The failure equivalent reduction and the bisimulation quotient computing are applied after the stutter equivalent reduction. This is because the redundancies are mainly introduced after the stutter equivalent reduction is applied. The following theorem shows that using these reductions together results in a reduced SG that is equivalent to the original one.

*Theorem 4.1:* Given a state graph $G$, $G \approx \texttt{Reduce}(G)$.

**Proof:** By Lemma 4.1, 4.2, 4.3, and 4.4. ∎

The following theorem asserts that the reduced global state graph by the parallel composition of the reduced local state

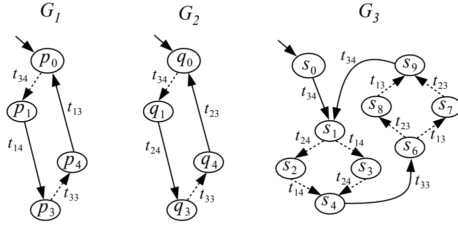


Fig. 9. Local State graphs reduced from those in Fig. 3.

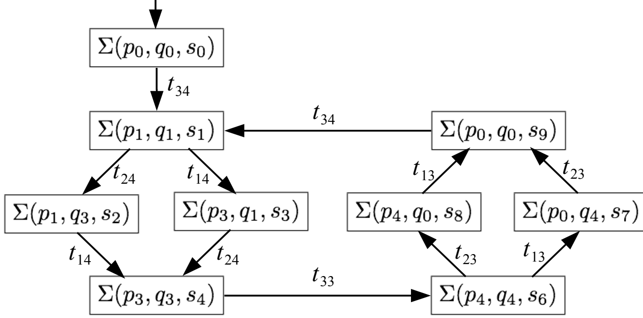

Fig. 10. The reduced global state graph by the parallel composition of the reduced local state graphs in Fig. 9.

graphs is equivalent to the global state graph obtained directly for the whole system.

*Theorem 4.2:* Let $M = \|_{1\leq i\leq n} M_i$ be a concurrent system, and $G = DFS(M)$. If $G_i$ are local state graphs constructed for $M_i$ with Algorithm 2, then the following statement is true:

$$G \approx \|_{1\leq i\leq n}\texttt{Reduce}(G_i).$$

**Proof:** By Property 3, Lemma 3.1, and Theorem 4.1. ∎

### F. Example

Refer to the state graphs shown in Fig. 3. After applying all reductions described in this paper, they are reduced to the ones as shown in Fig. 9. Composing these reduced local state graphs results in the reduced global state graph as shown in Fig. 10. This reduced global state graph contains 9 states, compared to 20 states in the unreduced one for the same example as shown in Fig. 4. Although this small example does not show significant benefits from the reductions, for large designs this compositional minimization framework often leads to dramatic improvements in runtime and memory usage, and allows much larger designs to be handled than what can be handled by the monolithic methods.

## V. Experimental Results

The compositional model checking framework presented in this paper has been implemented in a concurrent system verification tool, $Platu$, an explicit state model checker written in Java. Experiments have been performed on a set of examples including several non-trivial asynchronous circuit designs including a first-in-first-out buffer design (fifo) in [32], a tree arbiter (arb) and a distributed mutual exclusion element (dme) in [21], a pipeline controller (pipectrl) in an asynchronous microprocessor design in [40], an asynchronous implementation of a memory management unit (mmu) in [34], and a number of models of mutual exclusion algorithms from [36]. There are a large number of models in [36]. Since $Platu$ does not support channel communications, only the models without channels are selected. Furthermore, selected models that can be handled easily with the monolithic approach are also ignored.

In the experiments, the dme, arb, and fifo examples are partitioned according to their natural structures. In other words, each cell is a component. The pipectrl example is partitioned into five component, each of which contains ten logic gates. The mmu example is partitioned by following the structure provided in [34] where each component defines an output that is used by other components. Each BEEM example is organized as a network of communicating processes, therefore each process is treated naturally as a component.

All models are experimented with using both $Platu$ and SPIN, a well known model checker that is widely used in various applications. For every model, each component is specified with several local properties that are guaranteed to hold. The purpose is to show that no failures are reported by $Platu$ as shown in previous sections. Moreover, this allows a comparison with SPIN on correct systems where the full state space needs to be traversed as SPIN can find failures very fast.

In all experiments, upper bounds on time and memory are set to 900 seconds and 2 GB. The results collected include the actual runtime, memory usage, and the total number of reachable states found at termination of the search algorithm. All experiments are performed on a MacBook Pro notebook with a Intel Dual-core processor. However, only a single thread is used for all experiments. The results are shown in Table I.

In the table, the first column shows the design names. For asynchronous circuit implementations, a number is associated to indicate the number of variables in the corresponding model. For asynchronous circuit implementations, the variables used in the models are Boolean. For the examples from the BEEM benchmark, more information about their models can be found in [36]. The columns under Monolithic show the results from using the traditional DFS on the whole designs. The columns under SPIN show the results from using the SPIN model checker with partial-order reduction. The last three columns under CompMin show the results from using $Platu$. 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 found during the entire course of the compositional model checking process. The largest SGs are recorded because their sizes in general determine whether the whole process can be finished or not, therefore, their sizes need to be carefully controlled. In cases of time-out or memory-out, the corresponding entries in the table are filled with $-$. Comparing the results from running SPIN and Platu is not exactly fair as SPIN is implemented in C while Platu is in Java, which usually has noticeably higher memory overhead. Therefore, we believe if this framework is implemented in C, the results could be further improved.

From Table I, it can be seen that the traditional monolithic search method fails to finish quickly for most of the asynchronous circuit examples. This is understandable due to the

TABLE I

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 | Time | Mem | $\|S\|$ | Time | Mem | $\|S\|$ | Time | Mem | $\|S\|$ |
| arbN3 (26) | 0.315 | 2.4 | 3756 | 0.01 | 2.781 | 3756 | 0.087 | 3.89 | 52 |
| arbN5 (44) | 8.105 | 61.538 | 227472 | 1.35 | 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 | 0.01 | 2.195 | 644 | 0.015 | 3.39 | 20 |
| fifoN5 (22) | 0.733 | 16.253 | 20276 | 0.06 | 6.593 | 20276 | 0.017 | 3.62 | 20 |
| fifoN8 (34) | 199.353 | 845 | 3572036 | 27.2 | 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 | $267,999$ | 0.225 | 19.706 | 117270 | 0.15 | 9 | 248 |
| dmeN4 (44) | 1235 | 1032 | $15.7M$ | 13.5 | 553.421 | 4678742 | 0.18 | 10.4 | 248 |
| dmeN5 (55) | – | – | – | – | – | – | 0.2 | 11.9 | 248 |
| dmeN8 (88) | – | – | – | – | – | – | 0.31 | 16.8 | 248 |
| dmeN9 (99) | – | – | – | – | – | – | 0.32 | 17.3 | 248 |
| dmeN10 (110) | – | – | – | – | – | – | 0.38 | 18.2 | 248 |
| pipectrl (50) | – | – | – | – | – | – | 0.84 | 7.8 | 864 |
| mmu (55) | – | – | – | – | – | – | 1.7 | 17.2 | 2812 |
| at.3 | 37 | 283 | 1711621 | 3.9 | 167 | 1771622 | 11.7 | 220 | 1599744 |
| at.4 | 102 | 1018 | 6597246 | 17 | 509 | 6597247 | 58 | 815 | 6461412 |
| fischer.3 | 52 | 409 | 2896750 | 7.7 | 251 | 2896707 | 15 | 231 | 1676595 |
| fischer.4 | 25 | 342 | 1272254 | 2.8 | 150 | 1272356 | 4.2 | 175 | 399256 |
| fischer.6 | 312 | 1372 | 8237316 | 21 | 654 | 8321730 | 21.6 | 687 | 2192512 |
| lamport.5 | 10 | 135 | 1066799 | 0.2 | 8.7 | 152131 | 3.5 | 49.6 | 94776 |
| lamport.6 | 0.88 | 66 | 976246 | 0.9 | 68 | 976246 | 6.9 | 55 | 4960 |
| lamport.7 | – | – | – | 9.6 | 305 | 4720821 | 31.9 | 291 | 1479156 |
| Peterson.2 | 2 | 80 | 124704 | 0.15 | 77 | 114516 | 2.3 | 29 | 515 |
| Peterson.3 | 3 | 92 | 170156 | 0.1 | 73 | 35142 | 2.3 | 31 | 754 |
| Phils.6 | – | – | – | – | – | – | 0.37 | 10.9 | 96 |
| Phils.8 | – | – | – | – | – | – | 0.45 | 11.2 | 96 |
| syzmanski.3 | 18 | 174 | 1128424 | 1.59 | 118 | 998974 | 4.4 | 56 | 9632 |
| syzmanski.4 | 20.4 | 282 | 2313863 | 3.7 | 179 | 2137070 | 3.9 | 52.8 | 6375 |
| syzmanski.5 | – | – | – | – | – | – | 97 | 386 | 298054 |

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 some dme examples, SPIN does slightly better in terms of reduction in the number of states. On the other hand, SPIN quickly uses up 2 GB memory for the other dme examples. One possible explanation is that the partial-order reduction implemented in SPIN relies on the information about transition independency, which is obtained by examining structures of the Promela models. Since these asynchronous circuit examples are connections of basic logic gates, it may be difficult for SPIN to extract sufficient transition independency information from their Promela models for effective reduction.

On the other hand, the approach described in this paper can finish all examples in the table quickly. For all arb, fifo, and dme examples with regular structures, the total runtime and memory usage grow polynomially in the number of components in the examples. For pipectrl and mmu with irregular structures where SPIN fails, $Platu$ finishes them quickly using a very small amount of runtime and memory. For the pipectrl 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.

The significant improvements from the above experiments are mainly due to the loosely coupled nature of the asynchronous circuit examples. For all arb, fifo, and dme examples, each component connects to two or three neighboring components with simple communication interfaces. This loosely coupled structure allow the reductions to remove significantly larger portions of the irrelevant details from the initial and the intermediate SGs before the final reduced global SG is constructed. Usually, the largest SGs encountered during the compositional model checking process are the ones generated initially from high-level descriptions. Take fifoN# as an example. This design is a number of buffer cells connected in a chain, and each cell interfaces with exactly two neighbors. Note that all invisible transitions of each component can be removed as all properties are local. First, $G_1$ and $G_2$ are

TABLE II
EXPERIMENTS OF USING DIFFERENT ORDERINGS TO COMPOSE LOCAL SGS TO FORM THE GLOBAL REDUCED SG.

| | CompMin | | | CompMin$-1$ | | |
|---|---|---|---|---|---|---|
| Designs | Time | Mem | $\|S\|$ | Time | Mem | $\|S\|$ |
| pipectrl | 0.84 | 7.8 | 864 | 8.1 | 22.3 | 4631 |
| mmu | 1.7 | 17.2 | 2812 | 15.9 | 63 | 47303 |
| fischer.4 | 4.2 | 175 | 399256 | 33.4 | 721 | 1539684 |
| syzmanski.4 | 3.9 | 52.8 | 6375 | 6.2 | 117 | 109715 |

reduced, and then composed to form $G_{12}$. Then, all invisible transitions in this intermediate SG with respect to $G_3$ are removed resulting in $G'_{12}$. Next, $G'_{12}$ is composed with the reduced $G_3$. The final reduced SG can be found by repeating these steps until all local SGs are composed. In each step, the size of the intermediate SGs is never larger than the size of the component SGs initially generated with the local SG construction method. For mmu, it has a more complex structure, and the interface of some components are more complex. In this case, the largest SG is encountered when constructing an intermediate SG, and more time and memory are required to finish this example even though it has a smaller number of state variables than some other examples.

The results from running the examples from the BEEM benchmark show a similar situation, however, reduction is not as impressive. All these examples are coupled much more tightly than the previous asynchronous circuit examples. For all BEEM examples, communications among different processes are through shared variables, and there is little or no internal behavior in each process. Take at.3 and at.4 as examples. Each process has a very small number of invisible variables, therefore each process SG cannot be reduced too much. This leads to the final reduced SGs being almost as large as the ones from running the monolithic search on the entire models. In cases where SPIN can finish, it can finish faster and use less memory. This is because the compositional approach needs to construct the local SGs first whose complexity can blow up as the shared variables are usually not fully constrained. The other factors contributing to the worse performance include the complexity of all the SG reductions, especially the bisimulation quotient computation, and the large size of the intermediate SGs during the SG composition stage. On the other hand, even for these tightly coupled examples, the limited reduction obtained still leads to significant reduction in the overall runtime and total memory required in comparison with the monolithic approach.

Among all the reduction techniques, two of those involve local states that are failures. Both autofailure and failure equivalent reductions can potentially trim large portions of the state space, thus leading to significant reductions in the complexity of the SGs. To show this point, mmu is run again, but without using the above two reductions. The results are much worse as now it takes $38.2$ seconds and $496$ MB memory to build the reduced global SG with over $320,000$ states found in the largest SG.

During the SG composition stage, it is critical to keep the interface of the intermediate SGs as small and as constrained as possible in order to contain the size of the intermediate SGs. Therefore, the ordering for composing local SGs to form the global reduced SG has a big impact on the overall performance of the compositional method. To illustrate this point, several examples are selected to run the compositional method again but with different composition orderings. The results are shown in Table II. For convenience, the results for the same examples under method CompMin in Table I are copied into Table II under the same name, while the results obtained by using a different composition ordering are shown under CompMin$-1$ also in Table II. In this experiment, the orderings used to obtain the results in Table I are perturbed so that the interface of the intermediate SGs is not tightly constrained. From Table II, the results show that a slight change to the composition orderings can cause significant negative impact on the performance of this method. For example, finishing fischer.4 now takes about 33 seconds and 721 MB memory while with the better ordering it only takes about $4$ second and 175 MB memory to finish. The same pattern can be observed for the other examples. Also note that there exist other orderings that would cause this compositional method not to be able to finish within the time and memory limits. Therefore, it is critical to determine an appropriate composition ordering for any example to achieve good performance. A fully automatic approach for selecting a good composition order is described in [18]. For a composition strategy including a bounded number of connected processes, heuristics in [18] use a metric to estimate combined effect of the internal and interleaving transitions created after the composition, and pick the composition strategy with the highest metric score.

$Platu$ can prove correctness for systems without any flaws much more effectively. However, the framework implemented in $Platu$ carries noticeable overhead of constructing and reducing local SGs. For faulty systems, failures can be shown only after the reduced global SG is constructed at the end while the monolithic approach usually terminates after the first failure is found. Therefore, a highly optimized model checker like SPIN can often finish more quickly for systems with flaws.

Moreover, the performance of the presented framework critically depends on the efficiency and effectiveness of the local SG construction method. It works well for systems where the interfaces among components are simple. If sophisticated communications are used among components, the performance of the local SG construction can become a bottleneck, or even cause the whole framework to fail from the beginning. Take anderson.2 from [36] as an example. This example has a tiny state space and the monolithic approach can finish it instantly. However, the local SG construction uses up 2 GB memory and cannot finish building the local SGs for this example. Peterson.4 in [36] is another example. These negative results indicate the need of a better and more effective local SG construction method in order to make the presented framework more practical to a wider range of applications.

## VI. RELATED WORK

*Compositional verification* is essential to address the state explosion problem in model checking large systems. The

soundness and completeness of compositional verification are proved systematically in [43] based on a CSP-like process-algebraic language. The compositional methods can be roughly classified into *compositional reasoning* or *compositional minimization*. *Assume-guarantee* based compositional reasoning [5], [14], [27], [28], [33] 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. The semantic foundations of compositional state-based reasoning about concurrency is presented in [19]. In recent years, various approaches to automated assumption generation for compositional reasoning have been proposed and [6] provides a survey of several major approaches. The framework presented in [2] introduces a heuristic form of the assume-guarantee modular reasoning for asynchronous systems. In the *learning-based* approaches, assumptions represented by deterministic finite automata are generated with the $L^*$ learning algorithm and analysis of local counter-examples [35], [1], [15], [25], [11], [7], [24], [37]. 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 [35], [1]. An automated interface refinement method is presented in [39] 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 compositional reasoning methods has been demonstrated by verifying large examples, and advances of these methods have demonstrated the ability to reason about a global property represented as a conjunctive form of local specifications in [31], it is difficult in general for them to handle inherently global properties such as deadlock freedom. To address issues with learning based methods, [16], [4] present alternative compositional reasoning methods based on computing local invariants. These methods support checking deadlocks [4] or arbitrary LTL properties [16].

Compositional minimization [10], [26], [30], 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 [12], [13]. Heuristics that automatically determine the ordering of composition has been proposed in [17] and [18] for action-based labeled transition systems for process-algebraic languages. Compositional minimization has been successfully used in the verification of various systems [9], [8], [23], [38].

Compared to [15], [25], [7], this work allows global as well as local properties to be verified while the work in [15], [25], [7] considers local properties specified for individual components. This work is similar to [30]. However, [30] does not specify how the individual component state transition models are generated in the first place. On the other hand, this work uses a compositional reachability analysis to generate component state transition models automatically [41]. Another difference is that this work is based on a different state transition model than the labeled transition systems used in [15], [25], [7], [30]. Finally, compared to [26], the verification method in this work is complete and sound indicating that no false counter-examples are possible.

## VII. Conclusion

This paper presents a compositional verification framework 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 and the models of several mutual exclusion algorithms from the BEEM benchmark suite. Future work includes experiments on more diverse examples including communication protocols and multithreaded programs to fully demonstrate its potential. Additionally, it is necessary to develop effective approaches that make abstract counter-examples in the reduced SG 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 authors 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] N. Amla, E. A. Emerson, K. S. Namjoshi, and R. J. Trefler. Visual specifications for modular reasoning about asynchronous systems. In *Proceedings of the 22Nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, FORTE '02, pages 226–242, London, UK, UK, 2002. Springer-Verlag.

[3] C. Baier and J. P. Kateon. *Principles of Model Checking*. MIT Press, Cambridge, Mass., 2008.

[4] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *CAV*, 2009.

[5] 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.

[6] S. Berezin, S. V. A. Campos, and E. M. Clarke. Compositional reasoning in model checking. In de Roever et al. [20], pages 81–102.

[7] 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.

[8] H. Boudali, P. Crouzen, B. R. Haverkort, M. Kuntz, and M. Stoelinga. Architectural dependability evaluation with arcade. In *DSN*, pages 512–521. IEEE Computer Society, 2008.

[9] H. Boudali, P. Crouzen, and M. Stoelinga. A compositional semantics for dynamic fault trees in terms of interactive markov chains. In *ATVA*, pages 441–456, 2007.

[10] 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.

[11] 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.

[12] S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transations on Software Engineering and Methodology*, 5(4):334–377, 1996.

[13] S. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.

[14] 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.

[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] A. Cohen, K. S. Namjoshi, and Y. Sa'ar. Split: A compositional ltl verifier. In *Proc. of the 22nd Int. Conf. on Computer Aided Verification*, CAV'10, pages 558–561. Springer-Verlag, 2010.

[17] P. Crouzen and H. Hermanns. Aggregation ordering for massively compositional models. In L. Gomes, V. Khomenko, and J. M. Fernandes, editors, *ACSD*, pages 171–180. IEEE Computer Society, 2010.

[18] P. Crouzen and F. Lang. Smart reduction. In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software*, FASE'11/ETAPS'11, pages 111–126, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] F. S. de Boer and W. P. de Roever. Compositional proof methods for concurrency: A semantic approach. In de Roever et al. [20], pages 632–646.

[20] W. P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*. Springer, 1998.

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

[22] E.M.Clarke, O.Grumberg, and D.Peled. *Model Checking*. MIT Press, 2000.

[23] H. Garavel and H. Hermanns. On combining functional verification and performance evaluation using cadp. In L.-H. Eriksson and P. A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer, 2002.

[24] M. Gheorghiu, D. Giannakopoulou, and C. Psreanu. Refining interface alphabets for compositional verification. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 292–307. Springer Berlin Heidelberg, 2007.

[25] D. Giannakopoulou, C.S.Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, pages 297–320, 2005.

[26] S. Graf, B. Steffen, and G. Luttgen. Compositional minimization of finite state systems using interface specifications. *Formal Aspects of Computation*, 8(5):607–616, 1996.

[27] O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. on Prog. Lang. and Sys.*, 16(3):843–871, May 1994.

[28] 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.

[29] 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.

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

[31] A. Lomuscio, B. Strulo, N. G. Walker, and P. Wu. Assume-guarantee reasoning with local specifications. *Int. J. Foun. Comp. Sci.*, 24(4), 2013.

[32] A. J. Martin. Self-timed fifo: An exercise in compiling programs into vlsi circuits. Technical Report 1986.5211-tr-86, California Institute of Technology, 1986.

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

[34] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.

[35] 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.

[36] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[37] R. Singh, D. Giannakopoulou, and C. Psreanu. Learning component interfaces with may and must abstractions. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *LNCS*, pages 527–542. Springer Berlin Heidelberg, 2010.

[38] F. Tronel, F. Lang, and H. Garavel. Compositional verification using cadp of the scalagent deployment protocol for software components. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 2003.

[39] 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.

[40] 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.

[41] 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.

[42] H. Zheng, E. Rodriguez, Y. Zhang, and C. Myers. A compositional minimization approach for large asynchronous design verification. In *Proc. of the 19th Int. Conf. on Model Checking Software*, SPIN'12, pages 62–79, Berlin, Heidelberg, 2012. Springer-Verlag.

[43] J. Zwiers, W. P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In W. Brauer, editor, *ICALP*, volume 194 of *Lecture Notes in Computer Science*, pages 509–519. Springer, 1985.

**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. He join the Computer Science and Engineering Department at University of South Florida in 2004, and currently he is an associate professor. His research interests include formal methods in cyber-physical system design and verification, parallel and distributed computing, and reconfigurable computing. His recent research includes development algorithms and methods that make model checking scalable to large complex systems. Dr. Zheng received an NSF CAREER award in 2006, and an USF Outstanding Research Achievement award in 2007.

**Zhen Zhang** Zhen Zhang received the B.E. degree in Electrical and Electronic Engineering in 2007 from Dublin Institute of Technology, Dublin, Ireland, and the M.Phil degree in Computer Science in 2009 from the University of Manchester, Manchester, United Kingdom. He is currently pursuing his Ph.D. degree at the University of Utah, Salt Lake City, UT. His research interest is in the stochastic verification of cyber-physical systems. His current research is on the theory and algorithmic implementation of state space reduction techniques for the model checking of large concurrent systems.

**Chris J. Myers** Chris J. Myers received the B.S. degree in Electrical Engineering and Chinese history in 1991 from the California Institute of Technology, Pasadena, CA, and the M.S.E.E. and Ph.D. degrees from Stanford University, Stanford, CA, in 1993 and 1995, respectively. He is a Professor in the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT. Dr. Myers is the author of over 120 technical papers and the textbooks Asynchronous Circuit Design and Engineering Genetic Circuits. He is also a co-inventor on 4 patents. His research interests include asynchronous circuit design, formal verification of analog/mixed signal circuits and cyber-physical systems, and modeling, analysis, and design of genetic circuits. Dr. Myers received an NSF Fellowship in 1991, an NSF CAREER award in 1996, and best paper awards at the 1999 and 2007 Symposiums on Asynchronous Circuits and Systems. Dr. Myers is a Fellow of the IEEE, and he is a Member of the Editorial Boards for the IEEE Transactions on VLSI Systems, IEEE Design & Test Magazine, and Springer journal on Formal Methods in System Design.

**Emmanuel Rodriguez** Emmanuel Rodriguez received his B.S. degree in Computer Engineering from the University of South Florida in 2011. He started working with Dr Hao Zheng as an undergraduate research assistant in 2010 and then a gradate research assistant in 2011. His main effort was focused on developing and implementing novel and more efficient algorithms and methods for model checking of large and complex concurrent programs. His work was the basis of this journal paper and two other conference papers. He also helped to maintain an online version control system for Dr Zheng's research. He is currently working for Jagged Peak as a Software Engineer.

**Yingying Zhang** Yingying Zhang received a M.S. from Beijing University of Chemical Technology in 2006, and a M.S. degree in Computer Engineering from University of South Florida in 2013. Currently, she is a software engineer at Medtel Serivces Inc., Tampa, Florida. Her main research interest is formal verification for software and hardware, primarily in the area of state space reduction and timing analysis techniques for model checking.