

A Behavioral Analysis Approach for Efficient Partial Order Reduction

Yingying Zhang, Emmanuel Rodriguez, Hao Zheng, and Chris Myers

Abstract—Partial order reduction is essential to address state explosion when verifying concurrent systems by reducing states irrelevant to the verification results. However, traditional static approaches by analyzing system model structures often do not work well. To address such problem, this paper presents a new behavioral analysis approach where a compositional reachability analysis method is used to generate the over-approximate state spaces for all modules in a system, and then the independent transitions necessary for the partial order reduction are computed by examining these state spaces. Compared to the static analysis approaches, the independent transitions computed are more refined and accurate. The experimental results on some examples show that the presented approach is promising.

Index Terms—model checking, labeled petri-nets, partial order reduction, behavioral analysis, compositional verification.

I. INTRODUCTION

Model checking is a formal automated analysis method for verifying hardware and software systems. It systematically checks whether a model of a given system satisfies a desired property such as deadlock freedom and request-response properties [2]. Model checking has been developed into a mature and widely used approach in many applications. For an extensive review of model checking, please refer to [2], [4].

For synchronous systems, all state variables are updated simultaneously due to the global control of a clock. Instead of sharing a common clock and exchanging data on clock edges, asynchronous designs communicate through control protocols, and multiple components can perform executions concurrently. When verifying such a system, concurrently enabled executions need to be interleaved so that all possible orderings of executions are considered to avoid missing any behavior. This is the main cause of state explosion as the number of interleavings grows exponentially if a system has a high degree of concurrency, leading to an excessively large state space for even a relatively small system.

In order to address state explosion, partial order reduction (POR) have been developed [10], [6], [3], [5], which focuses on reducing states that do not affect the final verification results. With partial order reduction, instead of executing all enabled transitions in a state, POR finds an *ample* set [4]

which is a subset of the full enabled set. This can lead to a much smaller state space while still representing sufficient behavior for verification to derive the same results as the original full state space. Identifying unnecessary interleavings among the enabled transitions plays an important role in POR which is based on examining the dependency relations that exist between the transitions of a system. Since calculating the precise dependency relation between transitions may be as hard as verifying the whole system, POR often uses a conservative statically computed approximation for the dependency relation of the transitions [10], known as static POR, which guarantees a priori to include all relevant paths of the system. Dynamic POR [3], [6], which excludes the need to apply static analysis a priori by detecting data dependencies dynamically, may potentially miss certain relevant paths which can be added at a later step.

After a dependency relation is obtained, another key step is to produce a subset of enabled transitions, often referred to as *ample set*, from the full enabled set in every state. Ideally, the ample set generated in each state is minimized to achieve the maximal reduction. In this paper, we introduce a new behavioral analysis approach to find a more refined and accurate dependency relations used for POR. Unlike the traditional static analysis approaches, a dependency relation is derived by analyzing the abstract state space models computed by a compositional reachability analysis approach [11]. This new behavioral analysis approach generates a more refined and less conservative dependent relation, which leads to more reduction allowed by the static approaches.

This paper also introduces two new rules for ample set generation. One property focuses on how to choose independent transitions to construct a small ample set quickly. Another property considers the cycle detection condition [4] to improve the efficiency. Traditionally, when a cycle is formed, verification with POR requires at least one state in that cycle needs to be fully expanded, which means all enabled transitions need to be considered. This guarantees to avoid missing any behavior. However this requirement is often too conservative, and can lead to many redundant states generated. In our approach, we check if the transitions not in its ample set have been considered before this state is reached, and expand its ample set with those transitions that have not been considered yet. This improvement greatly improves the efficiency of POR.

In this paper, Section II introduces the Labeled Petri-Nets for modeling asynchronous systems. Section III introduces the concept of the independence relations, and describes the new behavioral analysis approach to derive more refined and

This material is based upon work supported by the National Science Foundation under Grant No. 0546492 and 0930510. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Yingying Zhang, Emmanuel Rodriguez and Hao Zheng are with the Dept. of Computer Science and Engineering at the University of South Florida, Tampa, FL. Chris Myers is with the Dept. of Electrical and Computer Engineering at the University of Utah, SLC, UT.

accurate independence relations. Section IV describes the properties for POR and shows how to use them to generate the minimized ample sets. The last two sections presents the experimental results, and concludes the paper.

II. BACKGROUND

A. Labeled Petri-Nets

This paper uses *Labeled Petri-Nets* to model asynchronous systems. Petri-Nets are a common modeling formalism for asynchronous designs [8], [1]. A Petri-net is a directed graph with a set of transitions and a set of places. A labeled Petri-Net is a Petri-net where transitions are labeled with various information representing a system's properties and behavior [9]. Its definition is given as follows.

Definition 2.1: A labeled Petri-net (LPN) is a tuple $N = \langle V, P, T, F, M_0, \text{init}, L \rangle$, where

- 1) V is a set of state variables of the integer type,
- 2) P is a finite set of places,
- 3) T is a finite set of transitions,
- 4) $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of the flow relations,
- 5) $M_0 \subseteq P$ is a finite set of initially marked places,
- 6) $\text{init} : V \rightarrow \mathbb{Z}$ is a labeling function that assigns each variable an initial value,
- 7) $L = \langle \text{Guard}, \text{Assign} \rangle$ is a pair of labeling functions for transitions in T , which is defined below.

A simple LPN example is shown in Fig 1. Fig.1(a) shows a simple asynchronous circuit consisting of three components, and Fig. 1(b) shows the LPNs for each component in the circuit. For each component, its LPN has 4 places and 4 transitions. The places are represented as circles, and the transitions are represented as boxes. Each place is preceded and followed by one or more transitions, and each transition is preceded and followed by one or more places. The flow relations are represented by the edges connecting the transitions and places [1]. The bullets found in some places are called tokens. Each place can have at most one token. A place is *marked* if it has a token. A marking of LPN, $M \subseteq P$ is a set of marked places.

The dynamic behavior of a concurrent system is captured by LPN transitions with labelings. Each transition $t \in T$ has a *preset* denoted by $\bullet t = \{p \in P \mid (p, t) \in F\}$, which is the set of places connected to t , and a *postset* denoted by $t \bullet = \{p \in P \mid (t, p) \in F\}$, which is the set of places to which t is connected. The *preset* and *postset* for places are defined similarly.

Before defining the transition labels formally, the grammar used by these labels is introduced first below [9]. The numerical portion of the grammar is defined as follows:

$$\begin{aligned} \chi ::= & c_i \mid v_i \mid (\chi) \mid -\chi \mid \chi + \chi \mid \chi - \chi \mid \chi * \chi \mid \\ & \chi / \chi \mid \chi \wedge \chi \mid \chi \% \chi \mid \text{NOT}(\chi) \mid \text{OR}(\chi, \chi) \mid \\ & \text{AND}(\chi, \chi) \mid \text{XOR}(\chi, \chi) \end{aligned}$$

where c_i is an integer constant from \mathbb{Z} , and v_i is an integer variable. The functions NOT, OR, AND, and XOR are bit-wise logical operations, and they are only applicable to integers and assume a 2's complement format with arbitrary precision. The

set \mathcal{P}_χ is defined to be all formulas that can be constructed from the χ grammar.

The Boolean portion of the grammar is as follows:

$$\begin{aligned} \phi ::= & \text{true} \mid \text{false} \mid v_i \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \chi \equiv \chi \mid \\ & \chi \geq \chi \mid \chi > \chi \mid \chi \leq \chi \mid \chi < \chi \end{aligned}$$

where the integer v_i is regarded as **true** if its value is nonzero, and **false** otherwise. In this sense, it is similar to the semantics of the C language. The set \mathcal{P}_ϕ is defined to be all formulas that can be constructed from the ϕ grammar.

As in Definition 2.1, each LPN transition is labeled with an enabling condition and a set of variable assignments. LPN transition labeling is defined by $L = \langle \text{Guard}, \text{Assign} \rangle$ where

- $\text{Guard} : T \rightarrow \mathcal{P}_\phi$ labels each LPN transition with a Boolean expression that defines its enabling condition.
- $\text{Assign} : T \times V \rightarrow \mathcal{P}_\chi$ labels each LPN transition $t \in T$ and variable $v \in V$ with a set of integer assignments made to v when t fires.

For convenience, $\text{supp}_G(t)$ is the set of variables appearing in the enabling condition of t , $\text{supp}_A(t)$ is the set of variables appearing in the expressions assigned to the variables of t , and $V_A(t)$ is the set of variables being assigned in t . Also, $\text{supp}(t) = \text{supp}_G(t) \cup \text{supp}_A(t)$.

For transition t_1 in Fig 1, the place in $\bullet t_1$ is marked, and $\bullet t_1$ is the same as $t_1 \bullet$. Its enabling condition $\text{Guard}(t_1) = (z = 0) \wedge (v = 0)$, and it has one assignment $\text{Assign}(t_1) = \{v := 1\}$.

B. Reachability Analysis

A basic approach for analyzing the dynamic behavior of a concurrent system modeled with LPNs is reachability analysis, which finds all possible state transitions and thus reachable states for such a system. The reachable state space is typically represented by a state graph. State graph (SG) is a directed graph where vertices represent states and edges represent state transitions.

The state of the LPNs is the pair (M, σ) where M denotes the marking and σ denotes the vector of variable values. Given a state s , $M(s)$ is a set of marked places in s and $\sigma(s)$ is the state vector of s . Also, for any expression $e \in \mathcal{P}_\chi \cup \mathcal{P}_\phi$, $\text{value}(e, s)$ denotes a function that returns the value of expression e in state s .

Before describing the reachability analysis approach, the LPN enabled transition is defined below:

Definition 2.2 (Enabled Transition): A LPN transition t is enabled at state s only if the following two conditions are met:

- 1) $\bullet t \subseteq M(s)$,
- 2) $\text{value}(e, s)$ is *true* or not zero for $e = \text{Guard}(t)$.

In Fig 1, every transition has its preset included in the initial marking. In the initial state, the values of variable u and z are 0, $\text{Guard}(t_{11})$, which is $u = 0 \wedge z = 0$, is evaluated to be true, therefore transition t_{11} is enabled in the initial state.

Given a LPN model, the set of transitions enabled in a state s is denoted by $\text{enabled}(s)$. The reachable state space of a LPN model can be found by exhaustively firing every enabled transitions starting at the initial state. Firing a transition leads

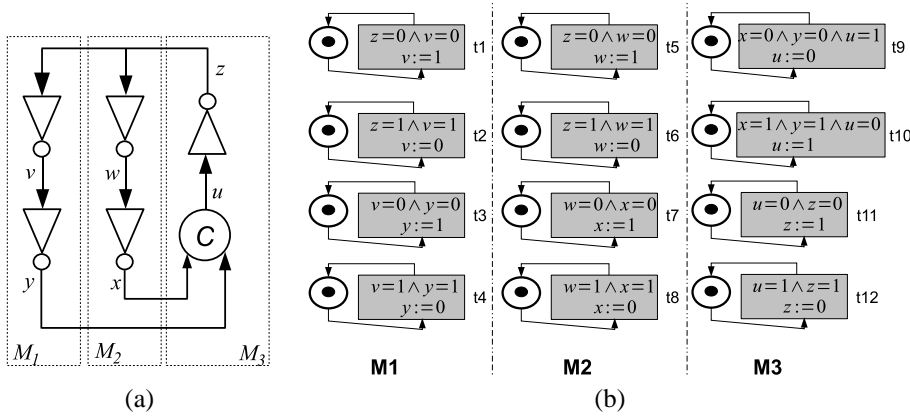


Fig. 1. (a) A simple asynchronous circuit. (b) The LPNs for module M_1 , M_2 , and M_3 . The initial values of variables u , v , w , x , y , and z are 0, 1, 1, 0, 0, and 0, respectively.

to a new state by generating a new marking and a new state vector according to the assignments labeled for such a transition. Detailed definition of transition firing can be found in [1]. In this paper, $s' = t(s)$ denotes that a new state s' is produced by firing transition t in state s .

The procedure to find the reachable state space of a given LPN model is given in Algorithm 1.

Algorithm 1: $search((T, P, F, M_0))$

```

1  $stack.push(s_0)$ ;
2  $stateTable.add(s_0)$ ;
3 while  $stack$  is not empty do
4    $s = stack.pop()$ ;
5   foreach  $t \in enabled(s)$  do
6      $s' = t(s)$ ;
7     if  $s' \notin stateTable$  then
8        $stack.push(s')$ ;
9        $stateTable.add(s')$ ;

```

III. COMPUTING INDEPENDENCE RELATIONS

From the previous section, the behavior of a concurrent system is modeled by executing transitions enabled in a state. If multiple transitions are enabled in a state, all possible orderings of executing these transitions need to be considered. This is generally referred to as interleaving, which is the major cause to state explosion in verifying concurrent systems. However, it has been discovered that most of the transition interleavings are redundant with respect to the final verification results, and identifying and removing the redundant interleavings can dramatically reduce the number of states explored. This approach is well known as partial order reduction (POR) [4]. In general, POR works by first computing an independence relation of all transitions in a LPN model, and then finding a reduced ample set for each state encountered during the state space search. The independence relation defines which transitions can be executed in different orderings but still lead to the same verification results. This section first reviews the general requirements of independent transitions, it then describes an

approach to deriving the independence relations from the LPN models. Since this approach is based on analyzing the syntactic structures of the LPN models, it is referred to as *static analysis*. The remainder of this section points out the limitations of the static analysis approach, and describes a *behavioral analysis* approach based on analyzing the abstract state space computed by a compositional reachability analysis method [11], which can extract more refined independence relations and lead to more reduction.

A. General Definition of Independent Relations

The state space of a concurrent system often contains many execution paths that correspond to different execution orderings of transitions in different parts of the system [7]. If transitions are independent, then their executions do not interfere with each other, which means that changing the execution orderings do not modify their the system behavior [3]. In this case, it is sufficient to consider one execution ordering. The notion of independent transitions can be formalized in the following definition [7].

Definition 3.1: $I \subseteq T \times T$ is an independence relation over transitions in T iff for each $(t_1, t_2) \in I$, the following conditions need to hold in every state $s \in S$.

- 1) $t_1 \in enabled(s)$,
- 2) $t_2 \in enabled(s)$,
- 3) If $t_1, t_2 \in enabled(s)$, then $t_2 \in enabled(s')$ where $s' = t_1(s)$,
- 4) If $t_1, t_2 \in enabled(s)$, then $t_1 \in enabled(s')$ where $s' = t_2(s)$,
- 5) If $t_1, t_2 \in enabled(s)$, $t_1(t_2(s)) = t_2(t_1(s))$, i.e., the execution effect t_1 followed by t_2 is indistinguishable from that of executing t_2 followed by t_1 .

In other words, independent transitions can neither disable nor enable each other, and independent transitions follow the commutative rule [10]. The following two sections describe how to derive the independent transitions from LPN models via static analysis and behavioral analysis approaches.

B. Static Analysis Approach

The static analysis approach derives independence relations by examining the structures of LPN models. Definition 3.2

shows a number of conditions that need to hold for two transitions to be independent. These conditions enforce the general requirements of the independent transitions as shown in Section III-A.

Definition 3.2: Given two LPN transitions t_1 and t_2 , t_1 independent with t_2 iff the following seven conditions are met:

- 1) $\bullet t_1 \cap \bullet t_2 = \emptyset$,
- 2) $t_1 \bullet \cap t_2 \bullet = \emptyset$,
- 3) $\bullet t_1 \cap t_2 \bullet = \emptyset$,
- 4) $V_A(t_1) \cap V_A(t_2) = \emptyset$,
- 5) $Supp(t_1) \cap V_A(t_2) = \emptyset$,
- 6) $Supp(t_2) \cap V_A(t_1) = \emptyset$,
- 7) $(V_A(t_1) \cap Supp(t_2) = \emptyset) \wedge (V_A(t_2) \cap Supp(t_1) = \emptyset)$,

In the above definition, condition 1 requires that two transitions do not share a common place in their presets so that firing one does not disable the other one. Condition 1 in Fig 2(a) shows an example of this case: transition t_1 and t_2 have the same *preset*, firing any one transition will disable the other transition. Condition 2 and 3 indicate that the enabling of a transition does not depend on the other one. If two transitions are executed sequentially, they can not be enabled at the same time. Condition 2 and 3 in Fig 2(a) show examples of this case, one transition can be enabled only after firing another transition if its preset is same as the postset of another transition.

If two transitions change the same variables, these two transitions must be interleaving. Fig 2(a)-Condition 4 shows example of this case. Two transitions change the same variable a , firing these two transitions in different orderings leads to different states.

The following are two situations where condition 5 and 6 can be violated.

- 1) Given two LPN transitions t_1 and t_2 , there exist $Guard(t_1) = true$ after executing $Assign(t_2)$, this means firing transition t_2 is necessary for enable transition t_1 . In Fig 2(b), there exists this kind of dependence relation between transition t_2 and t_6 , $Assign(t_2) = \{d := 1\}$ and $Guard(t_6) = \{d = 1\}$, firing transition t_2 can enable transition t_6 , therefore transition t_2 and t_6 are dependent transition.
- 2) Given two LPN transitions t_1 and t_2 , there exists $Guard(t_1) = false$ after executing $Assign(t_2)$, this means firing transition t_2 can possibly disable transition t_1 . In Fig 2(b), there exists this kind of dependence relation between transition t_1 and t_2 , $Assign(t_2) = \{d := 1\}$ and $Guard(t_1) = \{d = 0\}$, firing transition t_2 can disable transition t_1 and t_1 must be fired before t_2 , therefore transition t_1 and t_2 are dependent transitions.

For condition 7, if two transitions change the same variable set in another transition, then these two transitions must be set as dependent transitions. In Fig 2(b), this condition exists among transitions t_2, t_3 and t_5 . For transition t_2 , $V_A(t_2) = \{d\}$, for transition t_3 , $V_A(t_3) = \{c\}$, and for transition t_5 , $Supp(t_5) = \{d, c\}$. In this case, transition t_2 and t_3 must both be fired, then transition t_5 can be enabled.

If two transitions satisfy these seven conditions, they can be seen as independent, otherwise they are regarded as dependent.

After considering each pair of transitions in LPN model, an independent relation can be computed.

C. Behavioral Analysis Approach

The static analysis approach computes independence relations from a LPN model based on conservative but easy-to-check conditions as shown in the previous section. Using these conditions, which are typically carried out statically, often fails to find the independence between LPN transitions leading to exploring more transition interleavings than necessary [10]. For example, consider two transitions t_1 and t_2 where $Assign(t_1) = \{array[i] := 2\}$ and $Assign(t_2) = \{array[j] := 1\}$. If $i \neq j$, these two transitions are independent. However, it is difficult to determine whether $i \neq j$ holds just by looking at the LPN syntactic structures as $i \neq j$ can only be known after the whole state space is searched.

In order to deal with this problem, a behavioral analysis approach is developed which aims to determine the independence between transitions from the over-approximate state space models computed by using a compositional reachability analysis method. This paper assumes that a system model $M = M_1 || M_2 || \dots || M_n$ is the parallel composition of components $M_i (1 \leq i \leq n)$. Compositional reachability analysis constructs the state space for each component from an under-approximate context, and gradually expands its state space by including all state and transitions allowed by its neighboring components [11]. During this process, constraints are used to exchange interface information among components so that each component can determine which transition are allowed in a state based on the constraints on its inputs. The initial constraints are empty for the input of each component. With these initial input constraints, some components M_i may be able to fire some transitions on its output, and therefore produce new output constraints. Since the output of M_i may be the inputs of some other components M_j , then the output constraints from M_i are used as the input constraints of M_j . This, in turn, may allow some transitions to become enabled in M_j leading to more states and state transitions found. This process continues expanding the component state spaces by exchanging constraints until the output constraints produced by each component could not change anymore.

For example, Fig 1(a) shows a simple asynchronous circuit which is partitioned into three components M_1, M_2 and M_3 . For M_1 and M_2 , no transitions are enabled in the initial state because none of these transitions satisfies the initial constraint. For M_3 , the initial constraint allows transition $z+(z=1)$ to be enabled. After executing this transition, a new state is reached. Fig 3(a)-(c) shows the partial snapshot for this process and Fig 3(d) shows the final result for compositional reachability analysis. More details about this method can be found in [11].

After the component state graphs are generated, dependent transitions can be found first as defined in Definition 3.3. Transitions that are dependent on each other if firing them in different orders leads to different states. After finding all dependent transitions, the independent transitions can be derived easily.

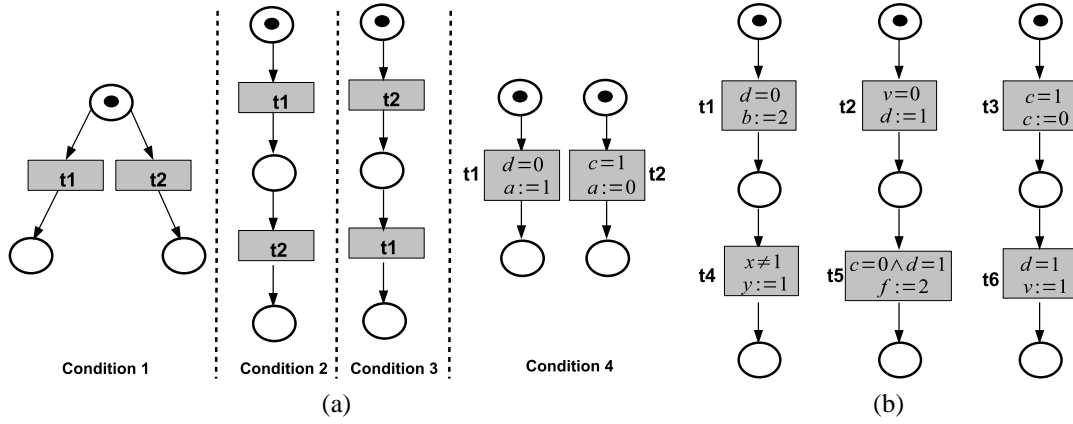


Fig. 2. (a) Examples of dependent transitions as Condition 1-4 are violated. (b) Examples of dependent transitions as Condition 5-7 are violated.

Definition 3.3: Let $G = (init, S, R)$ be a state graph. $D \subseteq T \times T$ is an dependence relation over transitions in T iff for each $(t_1, t_2) \in D$, one of the following conditions holds in any state $s \in S$.

- 1) $\exists(s, t_1, s') : t_2 \in enabled(s) \wedge t_2 \notin enabled(s')$,
- 2) $\exists(s, t_2, s') : t_1 \in enabled(s) \wedge t_1 \notin enabled(s')$,
- 3) $\exists s \in S : t_1, t_2 \in enabled(s) \wedge t_1(t_2(s)) \neq t_2(t_1(s))$,

The independence relation is defined as $I = T \times T - D$.

In the above definition, two transitions depend on each other if firing one can disable the other one as stated in the first two conditions, or firing these two transitions in different orders leads to different states. Fig 4 illustrates dependent transitions found when one of the conditions in Definition 3.3 above holds. Note that in this method if any of these conditions holds for transitions in any states, they are regarded as dependent globally. Additionally, the dependency checking is applied to transitions enabled in a state. If two transitions are never enabled in any state, they are regarded as independent by the definition. This may be counter-intuitive, however, treating transitions never enabled at the same time do not affect the results.

In this behavioral analysis approach, much more accurate information on how different orderings of firing transitions, for example, whether a transition can possibly be disabled by another transition, is readily available, much more refined dependence relations, and equivalently independence relations, can be derived. This allows many transitions enabled in each state to be removed during state space search, thus leading to enormous reduction in state space, as shown by the experimental results in section V. On other hand, the component state graphs generated are over-approximations as these state graphs may contain additional states and state transitions that would not exist if the monolithic state graph is generated for the whole design. This is proved in [11]. Because of these additional states and state transitions, dependent transitions found are still conservative. This is equivalent to say that some transitions that should be independent may not be found due to the additional states and state transitions. This does not affect the correctness of this method, and it would only cause reduction less effective.

IV. PARTIAL ORDER REDUCTION

After the independence relation is obtained, the next step is to compute a subset, $ample(s)$, of the enabled transitions $enabled(s)$ in each state s during the state space exploration. To reduce the complexity of the state space exploration, the size of $ample(s)$ should be as small as possible, and computing $ample(s)$ should have low overhead. To preserve the sufficient behavior to verify all properties soundly, four rules need to be satisfied as described in [4]. Among all the conditions, at-least-one successor and invisibility conditions can be readily satisfied. This section considers dependent-transition and cycle rules to compute $ample(s)$. The basic search algorithm in Algorithm 1 augmented with POR implementing the ideas discussed below is shown in Algorithm 2.

A. Dependent-Transition Rule

First of all, transitions included in $ample(s)$ should not disable any transitions not included in $ample(s)$. This requirement is specified in the following condition.

Condition 1: every transition in $enabled(s) - ample(s)$ is independent on every transition in $ample(s)$.

This condition guarantees that after firing a transition in $ample(s)$ the remaining transitions in $enabled(s)$ can still be enabled in the next state. However, this condition alone may cause some behavior to be missed. Refer to Fig. 5 for a simple example. In the initial state s_0 , transitions in $\{t_1, t_2, t_4\}$ are enabled, and every transition is independent on every other transition in this set. Therefore, choosing any single transition to be included in the ample set satisfies **Condition 1**, and it can be seen that the other transitions can still remain enabled after firing the transition in $ample(s_0)$. Suppose $ample(s_0) = \{t_1\}$. It is possible to reach a future state s where t_k and t_4 are enabled. Since firing t_4 first can disable t_k , firing t_k and t_4 in different orders may lead to different state spaces. Now suppose $ample(s) = \{t_4\}$ where firing t_1 is delayed. Since transition t_4 fires before t_1 , t_k may never get a chance to fire, thus possibly causing loss of states to be found. To avoid this situation, the following condition needs also to be satisfied.

Condition 2: In every state s , if a transition $t \in enabled(s)$ is dependent on any transition $t' \notin enabled(s)$, then firing t is delayed until t' is enabled.

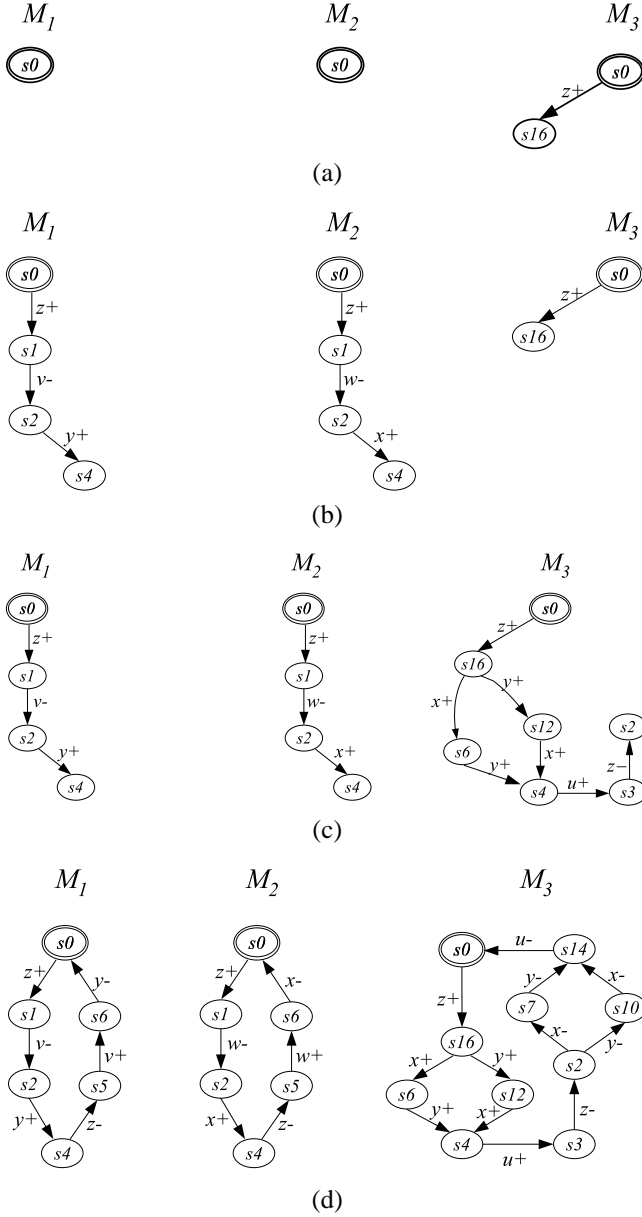


Fig. 3. (a)-(c) Snapshots of the state graphs generated during the compositional reachability analysis. (d) The state graphs generated when the compositional reachability analysis is done.

In the example shown in Fig. 5, since $D(t_4, t_k)$ holds, and $t_k \notin \text{enabled}(s_0)$. Therefore, in the initial state s_0 , $\text{ample}(s_0) = \{t_1\}$. In other words, with **Condition 2**, firing t_4 needs to be delayed until both t_4 and t_k are enabled, therefore firings of t_2 and t_k can be interleaved.

The basic idea behind **Condition 2** is that dependent transitions, if ever enabled in a state, need be interleaved, and the computed ample set must avoid loss of this interleaving. On the same basis, if two transitions are independent, but they both depend on some other transition, they need to be interleaved. In other words, in a state where these transitions are enabled, if one of them is selected to be included in the ample set, then the other one must also be included in the ample set.

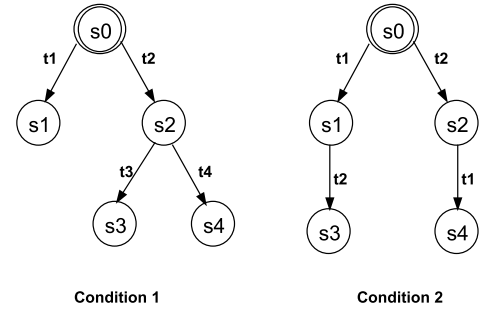


Fig. 4. Examples of dependent transitions that can be found in the state graphs as defined in Definition 3.3.

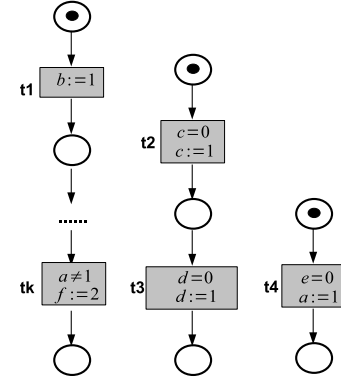


Fig. 5. Examples of dependent and independent transitions' Analysis for Ample Set.

Based on these two conditions, $\text{ample}(s)$ can be computed from $\text{enabled}(s)$ as follows. First, it is separated to two subsets $\text{dep}(s)$ and $\text{indep}(s)$. $\text{dep}(s)$ includes all transitions t enabled in s such that $D(t, t')$ for some transition t' , and $\text{indep}(s) = \text{enabled}(s) - \text{dep}(s)$ is the set of transitions enabled in s such that each transition $t \in \text{indep}(s)$ does not depend on any other transitions. If $\text{indep}(s) \neq \emptyset$, we can choose any single transition of $\text{indep}(s)$ to be included in $\text{ample}(s)$. If $\text{indep}(s) = \emptyset$, then $\text{dep}(s)$ is set to be $\text{ample}(s)$. More detail is shown in Algorithm 2.

B. Cycle Property

Only considering the above conditions is not enough as it may lead some transitions never to be fired when a cycle is formed. For example, suppose there exist two independent processes A and B where all transitions in A are independent with all transitions in B . Also suppose that transitions in process A are always fired first. If cycles are always formed eventually after firing transitions from process A successively, then transitions from Process B will never get a chance to be fired. The original cycle property in [4] requires that at least one state in each cycle during state space exploration to be fully expanded so that the reduced transitions are put back to the ample set. However, this requirement is often too conservative as the transitions put back to the ample set may have already been considered in other states in the same cycle, therefore leading to generating unnecessary states.

Suppose transitions t_1 and t_2 are enabled in state s , and they are independent. Also suppose that the ample set of s only includes t_1 . In our approach, when a cycle is formed from state s to s' by firing transition t_1 , instead of adding t_2 back into the ample set, we check if t_2 is included in the ample set of s' . If t_2 is included in the ample set of s' , it is not necessary to consider t_2 in state s because t_2 has been considered in state s' in that cycle. By avoiding adding the unnecessary transitions back for consideration, the state space can be dramatically reduced.

Algorithm 2: $search_{POR}((T, P, F, M_0))$

```

1  $stack.push(s_0)$ ;
2  $stateTable.add(s_0)$ ;
3 while  $stack$  is not empty do
4    $s = stack.pop()$ ;
5    $T_e = enable(s)$ ;
6    $T_d = dep(s)$ ;
7   if  $T_e \neq \emptyset$  then
8      $indep(s) = T_e / T_d$ ;
9     if  $indep(s) \neq \emptyset$  then
10       $ample(s) = \{e\}$  for  $e \in indep(s)$ ;
11    else
12       $ample(s) = dep(s)$ ;
13    for each transition  $t \in ample(s)$  do
14       $s' = t(s)$ ;
15      if  $s' \notin stateTable$  then
16         $stack.push(s')$ ;
17         $stateTable.add(s')$ ;
18         $continue$ ;
19      if  $s' \in stack$  then
20         $ample(s) = \emptyset$ ;
21      for each transition  $t \in T_e$  do
22        if  $t \notin fired(s) \cap t \notin ample(s')$  then
23           $ample(s).add(t)$ ;

```

C. Example

This section illustrates the idea of our partial order reduction method using a simple asynchronous circuit example shown in Fig 1. The initial value is $\{u \equiv 0, v \equiv 1, w \equiv 1, x \equiv 0, y \equiv 0, z \equiv 0\}$. Fig 6(a) shows a fully explored state graph which uses original DFS algorithms to traverse all possible states and transitions. There are 20 states found in this graph. In the state graphs, given a variable v , $v+$ and $v-$ are shorthand meaning v is changed to 1 and 0, respectively. Fig 6(b) shows a partial explored state graph. In the initial state s_0 , there is only one enable transition, then $ample(s_0) = enable(s_0) = \{z+\}$. After firing $\{z+\}$, state s_1 is reached, where the enabled transitions are $enable(s_1) = \{w-, v-\}$. Because these two transitions are independent, choosing one of them for $ample(s_1)$ is enough. In this case, we choose transition $\{w-\}$. After firing $\{w-\}$, the enabled transition set in the new state s_2 is $enable(s_2) = \{x+, v-\}$. These two transitions are also independent, therefore we can construct

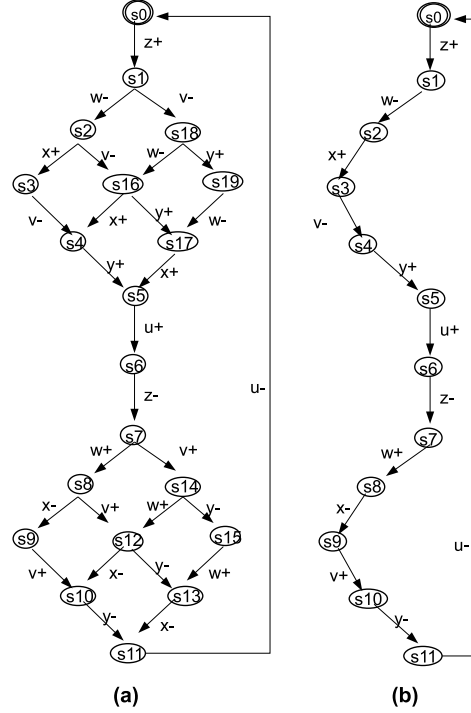


Fig. 6. State graphs for design in Fig. 1(a) before and after partial order reduction.

$ample(s_2) = \{x+\}$ by choosing $x+$. Repeat these steps, and eventually a reduced state graph with 12 states is found.

V. EXPERIMENTAL RESULTS

A prototype of the partial order reduction method with two realized approach described in this paper is implemented into an asynchronous system verification tool *Platu*, an explicit model checker. The asynchronous designs are modeled using LPNs and experiments have been performed on several asynchronous designs.

The experimental results are shown in Table I. The first two columns show the designs and their sizes in terms of number of variables. The table also includes the results from the original reachability analysis and the POR based on a static analysis approach to show the effectiveness of our behavioral analysis approach. The result obtained by this behavioral analysis approach are shown in columns under Method 3 in Table I, while the results from the original reachability analysis and the static analysis approach are shown in columns under Method 1 and 2 in Table I. All experiments are performed on a Linux workstation with a Intel Pentium Dual-Core CPU and 4 GB memory. In the table I, column *Mem* and *Time* show the maximal memory and the total time for each run, column $|S|$ shows the total number of states reached in the design.

First thing to notice from the table is the time and memory usage. For the small examples such as fig3a and arb1, POR (method 2 and method 3) requires more time and memory because these methods need additional time and memory to

TABLE I

COMPARISON DIFFERENT POR METHODS WITH ORIGINAL METHOD (TIME IS IN SECONDS, AND MEMORY IS IN MBS.). $|S|$ AND $|R|$ ARE THE NUMBERS OF STATES AND STATE TRANSITIONS OF THE STATE GRAPH.

Designs		Method 1:Reachability Analysis			Method 2:POR-Static Analysis			Method 3:POR-Behavioral Analysis		
Name	$ V $	Time	Mem	$ S $	Time	Mem	$ S $	Time	Mem	$ S $
fig3a	6	0.044	2.7	20	0.059	2.9	20	0.013	2.9	12
arb1	10	0.056	2.9	52	0.058	3.2	34	0.022	3.4	28
fifoN3	13	0.119	4.8	644	0.244	5.9	456	0.039	3.7	51
arbN3	27	0.315	2.4	3,756	0.483	1.6	1372	0.155	6.9	431
dmeN3	51	3.589	26.1	267,999	1.726	36.6	15,270	0.346	13.6	522
dmeN4	68	1235	1032	15.7M	9.234	198	292,664	0.88	2.7	1919
pipectrl	50	--	--	--	--	--	--	0.347	8.3	387
mmu	55	--	--	--	--	--	--	30	389	362,112

-- indicates that the example times out.

compute the independence relations. The overhead associated with these methods outweighs the gains, even though they can reduce some redundant states. But for the larger examples, dmeN3, dmeN4, pipectrl and mmu, method 2 and 3 are far more efficient by requiring much less time and memory. For dmeN4, the original reachability analysis approach (method 1) requires 1235 seconds and over 1 GB of memory. The static analysis approach (method 2) avoids exploring some unnecessary states, and takes 9 seconds. The memory is reduced by 80.81 percent to 198 MB. The behavioral analysis based approach (method 3) improves the time and memory even more dramatically. The runtime is reduced to less than a second, and the memory is reduced by over 99 percent to only 2.7 MB. From the experimental results, we can conclude that the larger the examples the more effective the POR is.

From the table, it can be seen that the POR based on the behavioral analysis is much more effective than the static analysis based approach by reducing runtime and memory much more significantly. For pipectrl and mmu, two large designs, the static approach cannot find a sufficient independence relation to allow verification to be finished, while the behavioral approach finishes verification for them without much difficulty. This is because the static approach derives the independence relations by examining the structure of the LPN models as described in the previous section, and the quality of the derived independence relations largely depends on the structures of the LPN models. On the other hand, the behavioral analysis derives the more accurate independence relations from state space models, allowing more transitions to be identified as independent therefore much more reduction during reachability analysis.

VI. CONCLUSION

This paper introduces a new approach to computing independence relations for partial order reduction. Since the analysis is applied on the state space models, the derived independence relations are more refined and accurate, which lead to more effective partial order reduction as shown by the experimental results. In the future, we plan to integrate POR with compositional verification in order to scale formal verification for large designs, and also extend the idea presented in this paper to real-time system verification.

REFERENCES

- [1] J. Ahrens. A compositional approach to asynchronous design verification with automated state. Master's thesis, University of South Florida, 2007.
- [2] C.Baier and J.-P.Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [3] C.Flanagan and P.Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages*, pages 110–121, 2005.
- [4] E.M.Clarke, O.Grumberg, and D.Peled. *Model Checking*. MIT Press, 2000.
- [5] E.M.Clarke, O.Grumberg, and M.Minea. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.
- [6] G.Gueta, C.Flanagan, and E.Yahav. Cartesian partial-order reduction. In *SPIN Workshop on Model Checking Software*, volume 4595 of *LNCS*. Springer, 2007.
- [7] G.J.Holzmann and D.Peled. An improvement in formal verification. In *roc. Seventh FORTE Conf. Formal Description Techniques*, 1994.
- [8] T. Murata. Petri nets: Properties, analysis, and applications. In *Proceedings of the IEEE* 77(4), pages 541–580, 1989.
- [9] R.A.Thacker, K.R.Jones, C.J.Myers, and H. Zheng. Automatic abstraction for verification of cyber-physical systems. In *1st International Conference on Cyber-Physical Systems*, 2010.
- [10] V.Kahlon, C.Wang, and A.Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proc. Int. Conf. on Computer Aided Verification*, volume 5643 of *LNCS*. Springer, 2009.
- [11] H. Zheng. Compositional reachability analysis for efficient modular verification of asynchronous designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(3):329–340, 2010.