Assumption Generation for Asynchronous Systems by Abstraction Refinement

Qiusong Yang[1], Edmund M. Clarke[3], Anvesh Komuravelli[3], and Mingshu Li[1,2]

[1] National Engineering Research Center of Fundamental Software
[2] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
[3] Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213, USA
{qiusong,mingshu}@nfs.iscas.ac.cn, {emc,anvesh}@cs.cmu.edu

Abstract. Compositional verification provides a way for deducing properties of a complete program from properties of its constituents. In particular, the assume-guarantee style of reasoning splits a specification into assumptions and guarantees according to a given inference rule and the generation of assumptions through machine learning makes the automatic reasoning possible. However, existing works are purely focused on the synchronous parallel composition of Labeled Transition Systems (LTSs) or Kripke Structures, while it is more natural to model real software programs in the asynchronous framework. In this paper, shared variable structures are used as system models and asynchronous parallel composition of shared variable structures is defined. Based on a new simulation relation introduced in this paper, we prove that an inference rule, which has been widely used in the literature, holds for asynchronous systems as long as the components' alphabets satisfy certain conditions. Then, an automating assumption generation approach is proposed based on counterexample-guided abstraction refinement, rather than using learning algorithms. Experimental results are provided to demonstrate the effectiveness of the proposed approach.

1 Introduction

Compositional verification provides a way for deducing properties of a complete program from properties of its constituents and it is a promising technique to address the state explosion problem. In particular, the assume-guarantee reasoning splits a specification into assumptions and guarantees [1]. A typical rule for assume-guarantee reasoning has the form of $\langle\varphi\rangle P\langle\phi\rangle$, where the *assumption* φ constrains the behavior of the environment and the *guarantee* ϕ specifies the behavior of the component P when φ is ensured. The rule means that in any execution where the environment behaves according to φ, it is guaranteed that P

C.S. Păsăreanu and G. Salaün (Eds.): FACS 2012, LNCS 7684, pp. 260–276, 2013.

behaves according to ϕ. For example, the following inference rule was proved in [2] for *synchronous* composition of Kripke Structures against properties ACTL*.

$$\frac{M_1 \parallel A \models G\ (\langle A \rangle M_1 \langle G \rangle) \qquad M_2 \preceq A\ (\langle\rangle M_2 \langle A \rangle)}{M_1 \parallel M_2 \models G\ (\langle\rangle M_1 \parallel M_2 \langle G \rangle)} \tag{1}$$

To prove that $M_1 \parallel M_2 \models G$, where $\parallel$ denotes the synchronous parallel composition operator, it is first to show that G is satisfied in $M_1 \parallel A$, assuming that its environment satisfies the assumption A. Then, the assumption A will be discharged on the other component M_2 by checking if $M_2 \preceq A$, where $\preceq$ is a strong simulation relation between two components. If the assumption is much smaller than M_2, checking $M_1 \parallel A \models G$ and $M_2 \preceq A$ might be more efficient than directly checking $M_1 \parallel M_2 \models G$.

However, in earlier works, human intervention was required to get an assumption satisfying an assume-guarantee rule. As it requires the interaction with an expert user, devising a proper assumption is not easy, even if not impossible, to accomplish for nontrivial verification problems. The pioneering work [3] presented an automatic assume-guarantee reasoning framework, in which the weakest assumption [4], represented as a finite-state automaton, is automatically learned using the L^* algorithm [5]. Since then, the problem of generating assumptions automatically has been extensively studied.

Prior work can be categorized according to the following three dimensions: system model, compositional pattern and learning algorithm used.

- System Model. Most of existing works [3,6,7,8,9,10] use *Labeled Transition Systems (LTSs)* as system models. In [11], *Kripke Structures* are used instead.
- Compositional Pattern. Existing works are focused purely, as least as we know, on the *synchronous* composition of LTSs or Kripke Structures. For LTSs, synchronous composition[1] usually allows the non-common actions between parallel components to be interleaved, while the executions of common actions must be synchronized [3,6,7,8,9,10]. Similarly, all components are forced to make transitions simultaneously in the synchronous composition of Kripke Structures [11].
- Learning Algorithm. As only safety properties are considered in most of existing works, the assumptions can be modeled as finite state automata and the L* algorithm and its variants are used for learning a regular set through membership and equivalence queries [3,6,7,8,9,10]. As a Kripke Structure is normally defined through its initial and transition predicates, the CDNF algorithm [12] is employed to learn Boolean functions in [11]. The CDNF algorithm can exactly learn a Boolean function by continuously asking membership and equivalence queries to a teacher, who can precisely answer every query.

[1] It should be noted that some authors called it as *asynchronous composition* of LTSs. In this paper, we only think a definition allowing interleaving on *common* actions as an asynchronous composition.

In this paper, for *asynchronous* systems, we propose an *alternative* approach for the automatic assumption generation based on predicate abstraction and interpolation, *instead of* using learning algorithms. That we are focused on asynchronous systems here is because real software programs are more naturally modeled as asynchronous systems, while synchronous models are more amenable to hardware systems. A concise example will be provided in the next section to demonstrate the differences between synchronous and asynchronous compositions. On the other side, the reason we propose an alternative to traditional learning-based approaches is that learning algorithms normally have a very high computational complexity. The running time of L* is bounded by a polynomial of n and m [5], where n denotes the number of states of the target automaton and m denotes the length of the longest counterexample, and CDNF bounded by a polynomial of the minimal CNF and DNF size of the target formula. However, in the assume-guarantee reasoning of asynchronous systems, the parameter n and the minimal CNF or DNF size are exponential in the number of global and local variables in the worst case.

The contribution of our paper is the following. First, we prove that the inference rule (1) holds for asynchronous systems with a redefined simulation relation between two components. The standard simulation relation in assume-guarantee reasoning, such as [9,10], is defined as the inclusion of trace languages, implying that, if $M_1 \preceq M_2$, the projection of every behavior of M_1 on the alphabet of M_2 is also a behavior of M_2. Under the asynchronous circumstances, the trace language inclusion induced simulation relation will be not monotonic with respect to the parallel composition operator, i.e. $M_1 \preceq M_2 \not\Rightarrow M_1 \parallel M_3 \preceq M_2 \parallel M_3$, which is essential for proving the inference rule (1), as a component can transit to an arbitrary state because of *jump*s in which shared variables are modified by other components while local variables are left untouched. The simulation relation introduced in Section 3 requires that the simulation relation is maintained between two components even if they make jumps, rather than real transitions.

Second, a method automating the assumption generation is proposed. The assumption A starts with the coarsest predicate abstraction, which is defined over a set of predicates over variables of M_2, such that $M_2 \preceq A$. Then, the assumption A will be refined in a series of iterations. In each iteration, $M_1 \parallel A \models G$ will be checked first. If it holds, we will have $M_1 \parallel M_2 \models G$. Otherwise, a finite counterexample will be returned as a result of $M_1 \parallel A \not\models G$. Then, we will check if the counterexample's projection on A is also feasible in M_2. If so, the algorithm can terminate as a real counterexample is found. Otherwise, a refined assumption, denoted as A', will be generated based on a new predicate obtained through counterexample analysis, using interpolation techniques. At the same time, the refinement ensures that: 1) $M_2 \preceq A'$. 2) A' contains strictly less behaviors than A in the sense that the counterexample will be not feasible in $M_1 \parallel A'$. When the algorithm terminates, an assumption satisfying the two premises of rule (1) is obtained, implying the property holds, or a real counterexample is found.

Finally, experimental results are provided to demonstrate that the proposed approach outperforms typical learning approaches based on CDNF.

Related Work. Since the first approach for automatic assume-guarantee reasoning based on automata learning was proposed [3], there have been extensive studies on the automatic assumption generation for compositional verification. These include devising new inference rules [6], extensions and optimizations of the L* algorithm [13,14], automatic refinement of the assumption's alphabet [15], symbolic methods for assume-guarantee reasoning [16,17], implicit learning based on CDNF [11] and minimal separating automata-based reasoning [18,19]. However, as discussed before, all these works are focused only on the synchronous parallel composition of LTSs or Kripke Structures, while it is more natural to model real software programs in the asynchronous framework. The rules that have been used for synchronous systems might not hold when asynchronous composition is considered. The reasoning framework must also be changed accordingly.

Our assumption generation method given in Section 4 is essentially based on the CEGAR(CounterExample Guided Abstraction Refinement) [20]. In [21], the authors also present a CEGAR-based method for assume-grantee reasoning, instead of using learning algorithms. Similarly, a CEGAR-based method for the assume-guarantee reasoning of probabilistic systems is given in [22]. However, those works are also focused on the synchronous parallel composition of LTSs.

2 Preliminary Definitions

In this section, preliminary definitions and notations used in the rest of the paper are given. LTSs are not selected as system models because it is more natural to model an asynchronous system with *shared variable structures*.

Shared Variable Structures. An SVS $M = (\eta, \zeta(\eta), \tau(\eta, \eta'))$, simplified as (η, ζ, τ), consists of the following components:

- $\eta = \{u_1, \cdots, u_m\}$: A finite set of Boolean variables, containing data and control variables. The set of *states* of M are the valuations over η, denoted as 2^η. For a state $s \in 2^\eta$, we use the notation $s_{\restriction_{\eta_1}}$, with $\eta_1 \subseteq \eta$, to denote the projection of s on η_1. For a set $S \subseteq 2^\eta$, then $S_{\restriction_{\eta_1}} = \{s_{\restriction_{\eta_1}} | s \in S\}$.
- $\zeta(\eta)$: The *initial predicate* characterizing the initial states. All valuations over η such that the initial predicate evaluates to true are the initial state of M.
- $\tau(\eta, \eta')$: The *transition predicate* relating the values η of state $s \in 2^\eta$ to the values η' in a successor state $s' \in 2^\eta$. There is a transition from s to s', denoted as $s \rightarrow s'$, if and only if $\tau(s, s')$ evaluates to true.

Parallel Composition. We also need to decide how to combine those processes into a concurrent system. Let $M_1 = (\eta_1, \zeta_1, \tau_1)$ and $M_2 = (\eta_2, \zeta_2, \tau_2)$ be two SVSs. The *asynchronous* and *synchronous* parallel compositions of M_1 and M_2 are denoted as $M = M_1 \parallel_a M_2$ and $M = M_1 \parallel_s M_2$, respectively. They agree on the definitions of η and ζ as follows:

- $\eta = \eta_1 \cup \eta_2$. The variables of the combined system are union of those of the components. The set of states of M is $2^{\eta_1 \cup \eta_2}$. It should be noted that we use

$s \in 2^{\eta_1 \cup \eta_2}$ to represent a state of M instead of (s_1, s_2), with $s_1 \in 2^{\eta_1}$ and $s_2 \in 2^{\eta_2}$, as (s_1, s_2) is a state of M only if they agree on the shared variables in $\eta_1 \cap \eta_2$.
- $\zeta = \zeta_1 \wedge \zeta_2$. For a state s of M, it is the initial state of M if and only if $s_{\restriction \eta_1}$ and $s_{\restriction \eta_2}$ are the initial states of M_1 and M_2, respectively. That is, $\zeta_1(s_{\restriction \zeta_1}) = true$ and $\zeta_2(s_{\restriction \zeta_2}) = true$.

As for the transition predicate τ, it is respectively defined as follows:

- In ***asynchronous*** composition, $\tau = \tau_1 \vee \tau_2$. For states s and s' of $M_1 \parallel_a M_2$, $s \rightarrow s'$ if and only if $\tau_1(s_{\restriction \eta_1}, s'_{\restriction \eta_1})$ *or* $\tau_2(s_{\restriction \eta_2}, s'_{\restriction \eta_2})$. During the transition, exactly one component, either M_1 or M_2, will make a move. If only $\tau_1(s_{\restriction \eta_1}, s'_{\restriction \eta_1})$ $(\tau_2(s_{\restriction \eta_2}, s'_{\restriction \eta_2}))$ evaluates to true, we say that $s \rightarrow s'$ is resulted from a transition of M_1 (M_2). If both $\tau_1(s_{\restriction \eta_1}, t_{\restriction \eta_1})$ and $\tau_2(s_{\restriction \eta_2}, t_{\restriction \eta_2})$ evaluates to true, then M_1 or M_2 will be non-deterministically selected to make a move.
- In ***synchronous*** composition, $\tau = \tau_1 \wedge \tau_2$. For states s and s' of $M_1 \parallel_s M_2$, $s \rightarrow s'$ if and only if $\tau_1(s_{\restriction \eta_1}, s'_{\restriction \eta_1})$ *and* $\tau_2(s_{\restriction \eta_2}, s'_{\restriction \eta_2})$. The components M_1 and M_2 will make a move simultaneously .

P_1 ::
l_{10} : $x_1 = 1$;
l_{11} : $x_2 = 1$;
l_{12} :

P_2 ::
l_{20} : $x_1 = 0$;
l_{21} :

Fig. 1. Process P_1 and P_2

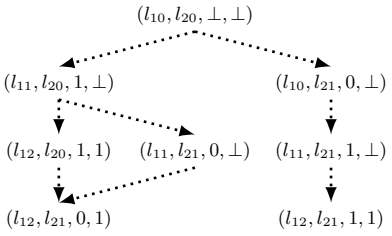

Fig. 2. Composition of M_1 and M_2

With the short program provided in Fig. 1, we give a sense of SVSs and demonstrate the difference between synchronous and asynchronous compositions. The Boolean variable x_1 is shared between the two processes P_1 and P_2. P_1 has also a local Boolean variable, named x_2. We can construct two SVSs $M_1 = (\eta_1, \zeta_1, \tau_1)$ and $M_2 = (\eta_2, \zeta_2, \tau_2)$ to receptively represent P_1 and P_2 as follows:

- $\eta_1 = \{x_1, x_2, pc_{11}, pc_{12}\}$ and $\eta_2 = \{x_1, pc_2\}$, where pc_{11}, pc_{12} and pc_2 are variables introduced to encode the program counters of P_1 and P_2. The term $!pc_{11} \wedge !pc_{12}$ corresponds to l_{10}, $pc_{11} \wedge !pc_{12}$ to l_{11}, $!pc_{11} \wedge pc_{12}$ to l_{12}, $!pc_2$ to l_{20}, and pc_2 to l_{21}.
- $\zeta_1 = !pc_{11} \wedge !pc_{12}$ and $\zeta_2 = !pc_2$, The valuation 00 of pc_{12} and pc_{11} corresponds to l_{10}, 01 to l_{11}, and so on.
- $\tau_1 = ((!pc_{11} \wedge !pc_{12} \wedge pc'_{11} \wedge !pc'_{12} \wedge x'_1 \wedge x'_2 = x_2) \vee (pc_{11} \wedge !pc_{12} \wedge !pc'_{11} \wedge pc'_{12} \wedge x'_2 \wedge x'_1 = x_1))$ and $\tau_2 = (!pc_2 \wedge pc'_2 \wedge !x'_1)$.

Let $\perp$ denote an arbitrary value of 1 or 0. The only path of M_1 can be represented as $(l_{10}, \perp, \perp) \rightarrow (l_{11}, 1, \perp) \rightarrow (l_{12}, 1, 1)$, where the first element of each state records the value of PC_1 and the other two elements record the values of x_1 and x_2. M_2's only path is $(l_{20}, \perp) \rightarrow (l_{21}, 0)$. The parallel composition of M_1 and M_2 is given in Fig. 2, in which a dotted edge represents a transition that could only occur in the asynchronous composition. However, the synchronous composition does not have any transitions because there is no state in $M_1 \parallel_s M_2$ such that x_1 evaluates to true *and* false at the same time.

Interpolant. Let (C_1,C_2) be a pair of sets of clauses, where a *clause* is a disjunction of literals and a *literal* is either a Boolean variable or its negation. If C_1 and C_2 are inconsistent, meaning that the conjunction of C_1 and C_2 is unsatisfiable. An interpolant for an inconsistent pair (C_1, C_2) is a formula $\mathcal{I}$ with the following properties:

- $C_1 \Rightarrow \mathcal{I}$.
- $\mathcal{I} \wedge C_2$ is unsatisfiable.
- $\mathcal{I}$ is defined over the common variables of C_1 and C_2.

In practice, an interpolant can be generated from a proof by resolution that C_1 and C_2 are inconsistent. Several SMT solvers, such as MathSAT [23] and iZ3 [24], have included supports for interpolant generation. The generation procedure is actually very simple and can be finished in linear time [25].

3 Compositional Verification of Asynchronous Systems

Let $M_1 = (\eta_1, \zeta_1, \tau_1)$ and $M_2 = (\eta_2, \zeta_2, \tau_2)$ be shared variable structures with $\eta_2 \subseteq \eta_1$ [2]. We define a *simulation relation* ($\preceq$) between two shared variable structures, using which we show that the inference rule (1) is sound and complete. One fundamental requirement of this is that the simulation relation should be compositional, i.e. whenever $M_1 \preceq M_2$, we have that $M_3 \parallel M_1 \preceq M_3 \parallel M_2$ for any other shared variable structure M_3 (maybe under some suitable extra assumptions). For simplicity, we will use $\parallel$ to denote $\parallel_a$ from here on.

Let $H \subseteq 2^{\eta_1} \times 2^{\eta_2}$. First, we consider *strong simulation*[3] [26], which is widely used in compositional reasoning, and show that it is not compositional for *asynchronous* systems.

Definition 1 (Strong Simulation [26]). *H is a* strong simulation *w.r.t a set of observable variables $\eta_o \subseteq \eta_2$ iff for $s \in 2^{\eta_1}, t \in 2^{\eta_2}$, $H(s,t)$ implies*

- *$s_{\restriction \eta_o} = t_{\restriction \eta_o}$, and*
- *for every $s' \in 2^{\eta_1}$ with $\tau_1(s,s')$, there exists a $t' \in 2^{\eta_2}$ such that $\tau_2(t,t')$ and $H(s',t')$.*

[2] This assumption comes naturally from the target application of assume-guarantee reasoning.

[3] A less restricted version of it appears in [11]; we will comment on it later in the section.

$M_1 \preceq_{\eta_o} M_2$ iff there is a strong simulation H such that for every $s_1^0 \in 2^{\eta_1}$ with $\zeta_1(s_1^0)$, there exists $s_2^0 \in 2^{\eta_2}$ with $\zeta_2(s_2^0)$ and $H(s_1^0, s_2^0)$.

In other words, every transition in M_1 is *simulated* by some transition of M_2. If this is everything, compositionality may not hold of asynchronous systems because a transition in $M_3 \parallel M_1$ resulting from a transition in M_3 can change the values of some variables common between M_1 and M_3[4].

To see this, let us consider a simple example. Let $\eta_1 = \eta_2 = \eta_o = \{x_1, x_2\}$. Let $\zeta_1 = \zeta_2 = (!x_1 \wedge !x_2)$. Also, let

$$\begin{aligned}\tau_1 &= (!(!x_1 \wedge x_2) \wedge (!x_1' \wedge !x_2')) \\ &\quad \vee ((!x_1 \wedge x_2) \wedge (x_1' \wedge x_2')), \\ \tau_2 &= (!x_1' \wedge !x_2').\end{aligned}$$

It is easy to see that $M_1 \preceq_{\eta_o} M_2$ with $H = \{(\langle 0,0\rangle, \langle 0,0\rangle)\}$ as the strong simulation. Now, let $M_3 = (\eta_3, \zeta_3, \tau_3)$ with $\eta_3 = \{x_2\}$, $\zeta_3(0)$ and $\tau_3 = x_2'$.

Consider $M_3 \parallel M_1$ and the initial state $\langle 0,0\rangle$. If M_3 takes a step, $M_3 \parallel M_1$ goes from $\langle 0,0\rangle$ to $\langle 0,1\rangle$. This step can also be taken in $M_3 \parallel M_2$. Now, let M_1 take a step and from τ_1 defined above, $M_3 \parallel M_1$ moves to $\langle 1,1\rangle$. But the only transitions from $\langle 0,1\rangle$ in $M_3 \parallel M_2$ are to either $\langle 0,0\rangle$ (if M_2 takes a step) or $\langle 0,1\rangle$ (if M_3 takes a step), neither of which is compatible with $\langle 1,1\rangle$.

To fix this problem, we add a condition to the definition of simulation relation resulting in the following.

Definition 2 (Strong Jump Simulation). *H is a* strong jump simulation *w.r.t. a set of observable variables $\eta_o \subseteq \eta_2$ iff for $s \in 2^{\eta_1}, t \in 2^{\eta_2}$, $H(s,t)$ implies*

- *$s_{\restriction \eta_o} = t_{\restriction \eta_o}$, we also say that s and t are compatible on η_o, and*
- *for every $s' \in 2^{\eta_1}$ with $\tau_1(s,s')$, there exists a state $t' \in 2^{\eta_2}$ such that $\tau_2(t,t')$ and $H(s',t')$, and*
- *for every $s' \in 2^{\eta_1}$ such that $s' \restriction_{\eta_1 \setminus \eta_o} = s \restriction_{\eta_1 \setminus \eta_o}$, there exists a state $t' \in 2^{\eta_2}$ such that $t' \restriction_{\eta_2 \setminus \eta_o} = t \restriction_{\eta_2 \setminus \eta_o}$ and $H(s',t')$.*

$M_1 \preceq^{\mathcal{J}}_{\eta_o} M_2$ iff there is a strong jump simulation H such that for every $s_1^0 \in 2^{\eta_1}$ with $\zeta_1(s_1^0)$, there exists $s_2^0 \in 2^{\eta_2}$ with $\zeta_2(s_2^0)$ and $H(s_1^0, s_2^0)$.

When the context is clear, $\preceq^{\mathcal{J}}_{\eta_o}$ will be written as $\preceq^{\mathcal{J}}$. Thus, in addition to being a strong simulation, a strong jump simulation needs M_2 to simulate any *jump* in M_1 which keeps the values of the variables in $\eta_1 \setminus \eta_o$, where $\setminus$ denotes the set minus operator, intact while changing the remaining variables arbitrarily. Such a jump effectively models any asynchronous transition in a system which M_1 is part of.

Now, it is not hard to see that in the example considered above, $M_1 \not\preceq M_2$. This is because a jump in M_1 from $\langle 0,0\rangle$ to $\langle 0,1\rangle$ can not be simulated by a similar jump in M_2 as the only transition from $\langle 0,1\rangle$ in M_2 is back to $\langle 0,0\rangle$.

[4] This is not possible in *synchronous composition* [11], because such a transition would be synchronized in both M_3 and M_1.

In fact, $\preceq^{\mathcal{J}}$ is compositional as we show below. The left part of Fig. 3 shows the relation between η_1, η_2 and η_3, in general. We observe that when $M_3 \parallel M_1$ takes a step, the variables in the region marked $\stackrel{?}{=} \emptyset$ can be changed for which there may not be a corresponding step in $M_3 \parallel M_2$ to a compatible state. For this reason, this region is assumed to be empty, leading to the assumption $\eta_1 \cap \eta_3 \subseteq \eta_2$ (note that this also holds in the above example); see the right part of the figure. As we will see soon, this is not an unreasonable assumption.

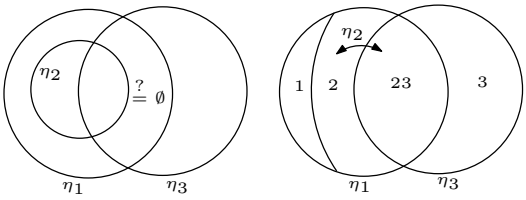

Fig. 3. Inclusion Relationships between Sets

Lemma 1. *If* $M_1 \preceq^{\mathcal{J}}_{\eta_{23}} M_2$ *and* $\eta_1 \cap \eta_3 \subseteq \eta_2$*, where* $\eta_{23} = \eta_2 \cap \eta_3$*, then* $M_3 \parallel M_1 \preceq^{\mathcal{J}}_{\eta_3} M_3 \parallel M_2$.

Proof. The assumption $\eta_1 \cap \eta_3 \subseteq \eta_2$ divides the space of the state variables into the four disjoint regions shown in Fig. 3 and given a state, we identify the corresponding components of the state by using the notation in the figure for subscripts. For example, the components for a state s in $M_3 \parallel M_1$ are identified as s_1, s_2, s_{23} and s_3.

Let H_{12} be a strong jump simulation between M_1 and M_2 satisfying the condition for the initial states. We show that $H = \{(s,t) | s \in 2^{\eta_1 \cup \eta_3}, t \in 2^{\eta_2 \cup \eta_3}, H_{12}(s_{23} \cdot s_2 \cdot s_1, t_{23} \cdot t_2), s_3 = t_3\}$ witnesses $M_3 \parallel M_1 \preceq^{\mathcal{J}}_{\eta_{23}} M_3 \parallel M_2$, where $\cdot$ denotes that concatenation of two state vectors. Let $H(s,t)$. We need to first show the three conditions of Definition 2 for H to be a strong jump simulation. Note that the target simulation relation is respect to η_3. So, any jump in $M_3 \parallel M_1$ or in M_1 needs to only keep the variables in $\eta_1 \setminus \eta_3$ intact.

By the assumption on H, $s_3 = t_3$ and $H_{12}(s_{23} \cdot s_2 \cdot s_1, t_{23} \cdot t_2)$. As H_{12} only has compatible pairs, the latter implies that $s_{23} = t_{23}$. Together, $s_{\restriction_{\eta_3}} = t_{\restriction_{\eta_3}}$.

Let $s \to s'$ be a transition in $M_3 \parallel M_1$. This can be due to a step in M_1 or in M_3.

In the first case, we have that $s_{23} \cdot s_2 \cdot s_1 \to s'_{23} \cdot s'_2 \cdot s'_1$ in M_1. As $H_{12}(s_{23} \cdot s_2 \cdot s_1, t_{23} \cdot t_2)$, there is a transition $t_{23} \cdot t_2 \to t'_{23} \cdot t'_2$ in M_2 with $H_{12}(s'_{23} \cdot s'_2 \cdot s'_1, t'_{23} \cdot t'_2)$. This transition in M_2 also means the transition $t \to t_3 \cdot t'_{23} \cdot t'_2 = t'$ exists in $M_3 \parallel M_2$. As $s_3 = t_3$, clearly $H(s', t')$.

In the second case, where the step is in M_3, we have that $s_3 \cdot s_{23} \to s'_3 \cdot s'_{23}$ in M_3. As $s_3 = t_3$ and $s_{23} = t_{23}$ from above, $t_3 \cdot t_{23} \cdot t_2 \to s'_3 \cdot s'_{23} \cdot t_2 = t'$ in $M_3 \parallel M_2$. Now, $s_{23} \cdot s_2 \cdot s_1 \to s'_{23} \cdot s_2 \cdot s_1$ is a jump. As $H_{12}(s_{23} \cdot s_2 \cdot s_1, t_{23} \cdot t_2)$, there exists a state $t'_{23} \cdot t_2$ of M_2 such that $H_{12}(s'_{23} \cdot s_2 \cdot s_1, t'_{23} \cdot t_2)$. Similarly, it also implies that $t'_{23} = s'_{23}$. As $s'_3 = s'_3$, clearly $H(s', t')$.

Finally, if there is a jump $s_3 \cdot s_{23} \cdot s_2 \cdot s_1 \rightarrow s'_3 \cdot s'_{23} \cdot s_2 \cdot s_1$ in $M_3 \parallel M_1$. It is the same to the second case given above.

The condition on the initial states can easily be checked. □

Finally, we show that the inference rule (1) is sound and complete for shared variable structures. In order to do so, we show that $\preceq^{\mathcal{J}}$ is reflexive and that it preserves LTL properties.

Lemma 2. *$\preceq^{\mathcal{J}}$ is reflexive, i.e. $M \preceq^{\mathcal{J}} M$ for any shared variable structure M.*

Proof. We only have jumps and the proof is straightforward. □

Lemma 3. *Let $M_1 \preceq^{\mathcal{J}}_{\eta_o} M_2$ (with $\eta_o \subseteq \eta_2 \subseteq \eta_1$). Let f be an LTL formula defined over η_o. Then, $M_2 \models f$ implies $M_1 \models f$.*

Proof. Let H be a strong jump simulation witnessing $M_1 \preceq^{\mathcal{J}}_{\eta_o} M_2$. As H is also a strong simulation and, for $H(s,t)$, an atomic proposition of f is labeled in s if and only if it is labeled in t, preservation follows [27]. □

Theorem 1 (Compositional Verification). *Let G be an LTL formula defined over η_G. Let $\eta_I = (\eta_1 \cap \eta_2)$. If $\eta_G \subseteq \eta_1 \cup \eta_I$ and $\eta_I \subseteq \eta_A \subseteq \eta_2$, then the inference rule*

$$\frac{\begin{array}{c} M_1 \parallel A \models G \\ M_2 \preceq^{\mathcal{J}}_{\eta_I} A \end{array}}{M_1 \parallel M_2 \models G} \tag{2}$$

is sound and complete.

Proof. *Soundness.* Assume $M_1 \parallel A \models G$ and $M_2 \preceq^{\mathcal{J}}_{\eta_I} A$. Lemma 1 gives us $M_1 \parallel M_2 \preceq^{\mathcal{J}}_{\eta_1} M_1 \parallel A$. From the former and $\eta_G \subseteq \eta_1 \cup \eta_I$, Lemma 3 gives us $M_1 \parallel M_2 \models G$.

Completeness. Assume $M_1 \parallel M_2 \models G$. Let $A = M_2$. From Lemma 2, $M_2 \preceq^{\mathcal{J}}_{\eta_I} M_2$. The other premise is what we just assumed. □

Note that the assumptions on the variables in the above theorem are quite reasonable and $\eta_1 \cap \eta_2 \subseteq \eta_A$ means that η_A should include all the common variables of M_1 and M_2 which is what is expected of an assumption.

4 Automatic Assumption Generation

Let $M_1 = (\eta_1, \zeta_1, \tau_1), M_2 = (\eta_2, \zeta_2, \tau_2), A = (\eta_A, \zeta_2, \tau_2)$ and G be an LTL formula defined over the alphabet η_G. Let $\eta_I = \eta_1 \cap \eta_2$ and $\eta_G \subseteq \eta_1 \cup \eta_I$. They will be fixed in the rest of this section. Although the rule (2) holds for all LTL formulae, we will only consider G as safety properties from here on to ensure that a finite counterexample is returned when $M_1 \parallel A \models G$ does not hold. As discussed later, the work presented is this paper can be easily extended to liveness properties.

The basic idea of our algorithm is the following. The assumption A starts with the coarsest over-approximation. Then, the assumption A will be refined in a series of iterations. In every iteration, $M_1 \parallel A \models G$ will be model checked first. If it holds, we will have $M_1 \parallel M_2 \models G$, as A is an over-approximation of M_2, and the algorithm terminates. Otherwise, a finite counterexample will be returned as a result of $M_1 \parallel A \not\models G$. Then, we will check if the counterexample's projection on A is also feasible in M_2. If so, the algorithm can terminate as a real counterexample is found. Otherwise, a refined version of A, denoted as $A' = (\eta'_A, \zeta'_A, \tau'_A)$, will be generated based on counterexample analysis. At the same time, the refinement has the following two properties: 1) $M_2 \preceq A'$, i.e. the updated assumption is still an over-approximation of M_2. 2) A' contains strictly less behaviors than A in the sense that the counterexample will be not feasible in $M_1 \parallel A'$. Then, the algorithm enters into the next iteration. The termination of the algorithm is guaranteed by the completeness of the inference rule (2).

Alphabet Selection. In those works on assume-guarantee reasoning of synchronous compositions with systems being modeled as LTSs, the alphabet η_A of the assumption can be a strict subset of η_2 and contains only those common variables between M_1 and M_2 and those variable necessary to prove the property, i.e. $\eta_A = (\eta_G \cup \eta_1) \cap \eta_2$. Even a smaller alphabet is used in [28] with the help of alphabet refinement techniques. However, when $M_1 \parallel A \models G$ does not hold and the assumption is refined based on the returned counterexample, the counterexample analysis in an asynchronous environment, in which systems are modeled as SVSs, requires that $\eta_A = \eta_2$. The rational behind the alphabet selection is the following.

Assume that $\eta_A \subset \eta_2$, i.e η_A is a strict subset of η_2. When a counterexample to $M_1 \parallel A \models G$ is returned, the assumption must be refined to make the counterexample infeasible in $M_1 \parallel A'$. In any way, some transition, assuming that $s \to s'$, that is enabled in A will be disabled in the refined assumption A'. The problem is that for every transition $t \cdot s \to t' \cdot s'$ of M_2, where $t, t' \in 2^{\eta_2 \setminus \eta_A}$, no transitions in A' are available to simulate it. In the case of LTSs, it is different because, if a transition labelled with an action is removed from A, another transition with the same label might still exist. For the same reason, the assumption generation approach given in [11], which uses the CDNF algorithm to learn an assumption represented as SVS, also assumes that $\eta_A = \eta_2$.

Predicate Abstraction. If $\eta_A = \eta_2$, the assumption A satisfying the premise $M_2 \preceq^{\mathcal{J}} A$ will be not "smaller" than M_2. To solve the problem, we use predicate abstraction to compress the states of the assumption. Let $\mathcal{AP}$ be a set of predicates over $\bar{\eta}_2 = \eta_2 \setminus \eta_I$ (recall that $\eta_I = \eta_1 \cap \eta_2$). Those variables in η_I are not included because they are used for interacting with the component M_1. The equivalence relation induced from $\mathcal{AP}$ over the set $2^{\bar{\eta}_2}$ is denoted as $\equiv_{\mathcal{AP}}$.

Given a set of predicates $\mathcal{AP}$ over $\bar{\eta}_2$, an existential abstraction of M_2, denoted as $M_2^{\mathcal{AP}} = (\eta_I \cup \bar{\eta}_2, \zeta_{\mathcal{P}}, \tau_{\mathcal{P}})$, is defined as the following:

- For $s_1 \in 2^{\bar{\eta}_2}$ and $s_2 \in 2^{\eta_I}$, $\zeta_{\mathcal{P}}(s_1 \cdot s_2)$, if there exists a state $t_1 \cdot s_2$ of M_2, where $t_1 \in 2^{\bar{\eta}_2}$, such that $t_1 \in [s_1]_{\equiv_{\mathcal{AP}}}$ and $\zeta_2(t_1 \cdot s_2)$.

– For $s_1, s_1' \in 2^{\bar{\eta}_2}$ and $s_2, s_2' \in 2^{\eta_I}$, $\tau_{\mathcal{P}}(s_1 \cdot s_2, s_1' \cdot s_2')$ if there exist $t_1, t_1' \in 2^{\bar{\eta}_2}$ such that $\tau_2(t_1 \cdot s_2, t_1' \cdot s_2')$, $t_1 \in [s_1]_{\equiv_{\mathcal{AP}}}$, and $t_2 \in [s_1']_{\equiv_{\mathcal{AP}}}$.

where $[s]_{\equiv_{\mathcal{AP}}}$ denotes an equivalence class of $\equiv_{\mathcal{AP}}$. By introducing predicate abstraction, we can reduce the size of M_2 from $2^{|\eta_2|}$ to $2^{|\mathcal{AP}|+|\eta_I|}$, where $|S|$ denotes the size of a set S. Normally, the size of η_I is small for a well designed system.

Lemma 4. *For a set of predicates $\mathcal{AP}$ over $\bar{\eta}_2$ and the induced existential abstraction $M_2^{\mathcal{AP}}$ of M_2 from it, $M_2 \preceq_{\eta_I}^{\mathcal{J}} M_2^{\mathcal{AP}}$.*

Proof. Let $H = \{(s,t) | s \in 2^{\eta_2}, t \in 2^{\eta_2}, s_{\eta_I} = t_{\restriction \eta_I}, s_{\restriction \bar{\eta}_2} \in [t_{\restriction \bar{\eta}_2}]_{\equiv_{\mathcal{AP}}}\}$.

Assumption Initialization. The set $\mathcal{AP}$ will be *initialized to the empty set.* The abstraction of M_2 induced from the empty set will be used as our coarsest over-approximation.

Assumption Refinement. Let $p = v_0, v_1, \cdots, v_k$ be a path of $M_1 \parallel M_2$, where v_i is a valuation over $\eta_1 \cup \eta_2$. Some transitions are executed as a result of the executions of M_1, while others as a result of M_2. The projection of p on M_1 (M_2) is obtained by removing details about those transitions and states not related to any transitions of M_1 (M_2) and projecting those remained states onto η_1 (η_2). It should be noted that a projection is not necessarily a path of a component. For example, $p' = \langle 0,0\rangle \xrightarrow{P_1} \langle 1,0\rangle \xrightarrow{P_2} \langle 0,0\rangle \xrightarrow{P_1} \langle 0,1\rangle$ is a path of the program given in Fig. 1, where the labels on transitions indicate the SVS, P_1 or P_2, to be executed. The projection of p' on P_1 is $p'' = \langle 0,0\rangle \xrightarrow{P_1} \langle 1,0\rangle \rightsquigarrow \langle 0,0\rangle \xrightarrow{P_1} \langle 0,1\rangle$, where $\rightsquigarrow$ denotes a *jump*. In the jump, the shared variable x_1 is changed by P_2 while the local variable x_2 is left untouched. It is easy to see the jump is not feasible in M_1.

Definition 3 (η-$\mathcal{J}$-Path). *A projection including transitions and jumps, in which the variables contained in η are left untouched, is called an η-$\mathcal{J}$-path.*

If $M_1 \parallel A \models G$ does not hold, a counterexample will be returned by a model checker. Let $ce = v_0, v_1, \cdots, v_k$ be the counterexample's projection on A. As we use the equivalence classes of $\equiv_{\mathcal{AP}}$ to represent the set of states of $2^{\bar{\eta}_2}$, rather than enumerating them explicitly, every state of ce belongs to $2^{AP \cup \bar{\eta}_2}$. From previous discussions, we know that ce is an η_{J_1}-$\mathcal{J}$-path, where $\eta_{J_1} = \eta_A \setminus \eta_1$. Then, we will have to decide if ce is feasible in M_2 by checking if there exists a η_{J_2}-$\mathcal{J}$-path $p = s_0, s_1, \cdots, s_k$ in M_2, where $\eta_{J_2} = \eta_2 \setminus \eta_1$, such that:

– $\zeta_2(s_0)$, $[s_{0\restriction \bar{\eta}_2}]_{\equiv_{\mathcal{AP}}} = v_{0\restriction_{\mathcal{AP}}}$, and $s_{0\restriction_{\eta_I}} = v_{0\restriction_{\eta_I}}$.
– If $v_i \rightarrow v_{i+1}$ is a *transition* or an η_{J_1}-jump of ce, then $[s_{i\restriction \bar{\eta}_2}]_{\equiv_{\mathcal{AP}}} = v_{i\restriction_{\mathcal{AP}}}$, $s_{i\restriction_{\eta_I}} = v_{i\restriction_{\eta_I}}$, $[s_{i+1\restriction \bar{\eta}_2}]_{\equiv_{\mathcal{AP}}} = v_{i+1\restriction_{\mathcal{AP}}}$, $s_{i+1\restriction_{\eta_I}} = v_{i+1\restriction_{\eta_I}}$, and $\tau_2(s_i, s_{i+1})$ if $v_i \rightarrow v_{i+1}$ is a real transition.

If such an η_{J_2}-$\mathcal{J}$-path is found, the counterexample to $M_1 \parallel A \models G$ is guaranteed to be feasible in $M_1 \parallel M_2$, as $\eta_1 \cap \eta_2 \subseteq \eta_A$ ensures that all common variables

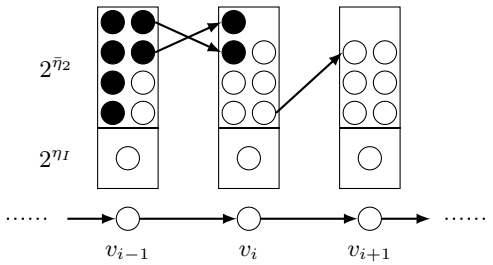

Fig. 4. Counterexample Analysis

between M_1 and M_2 are included in η_A. Thus, $M_1 \parallel M_2 \not\models G$ and the algorithm will terminate. Otherwise, the assumption A has to be refined to exclude the counterexample.

In reverse to the existential abstraction given before, a concretization function γ will be defined as the following:

$$\gamma(v) = \{s \cdot t | s \in 2^{\bar{\eta}_2}, t \in 2^{\eta_I}, [s]_{\equiv_{\mathcal{AP}}} = v_{\restriction_{\mathcal{AP}}}, \text{ and } t = v_{\restriction_{\eta_I}}\} \tag{3}$$

In Fig. 4, the path ce is shown at the bottom, while the set of concrete states $\gamma(v_i)$ corresponding to every state v_i is shown above. All concrete states have the same values over η_I, but different over $\bar{\eta}_2$. Let $R_0 = \{s | s \in 2^{\eta_2}, s \in \gamma(v_0), \text{ and } \zeta_2(s)\}$. Thus, R_0 denotes the set concrete states corresponding to the initial abstract state. Then, R_i for $i \geq 1$ is recursively defined as follows:

- if $v_{i-1} \rightarrow v_i$ is a transition of ce, then $R_i = \{s | s \in 2^{\eta_2}, s \in \gamma(v_i), \exists t \in R_{i-1} : \tau_2(t, s)\}$.
- if $v_{i-1} \rightarrow v_i$ is a jump of ce, then $R_i = \{s | s \in 2^{\eta_2}, s \in \gamma(v_i), \exists t \in R_{i-1} : s_{\restriction_{\bar{\eta}_2}} = t_{\restriction_{\bar{\eta}_2}}\}$.

Every set R_i actually defines those concrete states that can be reachable along the counterexample. If the counterexample ce is not feasible in M_2, there's no corresponding transition in M_2 for some *transition* $v_i \rightarrow v_{i+1}$ of ce. Then, we will have:

$$R_i \cap (\tau_2^{-1}(\gamma(v_{i+1})) \cap \gamma(v_i)) = \emptyset \tag{4}$$

where τ_2^{-1} denotes the pre-image calculation. As shown in Fig. 4, there is at least one sate $s \in \gamma(v_{i+1})$ which is reachable from some states of $\gamma(v_i)$. However, none of those states of $\gamma(v_i)$ are included in R_i.

Because two states respectively from R_i and $\tau_2^{-1}(\gamma(v_{i+1})) \cap \gamma(v_i)$ agree on their projections on η_I, the equation (4) implies that the intersection between $R_{i\restriction_{\bar{\eta}_2}}$ and $(\tau_2^{-1}(\gamma(v_{i+1})) \cap \gamma(v_i))_{\restriction_{\bar{\eta}_2}}$ is also empty. As we can symbolically compute the two sets, let f_1 and f_2 be the Boolean formulae representing R_i and $\tau_2^{-1}(\gamma(v_{i+1})) \cap \gamma(v_i)$, respectively. Let $\mathcal{F}_1 = \exists \eta_I : f_1$ and $\mathcal{F}_2 = \exists \eta_I : f_2$. We know that $\mathcal{F}_1 \wedge \mathcal{F}_2$ is unsatisfiable. Then, let $\mathcal{I}$ be the interpolant of $(\mathcal{F}_1, \mathcal{F}_2)$, which is defined over $\bar{\eta}_2$. We will add the interpolant I into $\mathcal{AP}$ as a new predicate and refine the assumption according to the augmented $\mathcal{AP}$.

Algorithm AAG(M_1, M_2, G)
Let $\mathcal{AP} = \emptyset$ be a set of predicates over $\bar{\eta}_2$.
while $TRUE$ **do**
Let $A = M_2^{\mathcal{AP}}$.
Check if $M_1 \parallel A \models G$.
if $M_1 \parallel A \not\models G$ **do**
Let ce be the counterexample's projection on A
Check if ce is feasible in M_2.
if ce is feasible **do**
$M_1 \parallel M_2 \not\models G$ and terminate.
else
Calculate interpolant based on formula (4).
Add the interpolant to $\mathcal{AP}$.
else
$M_1 \parallel M_2 \models G$ and terminate.

Fig. 5. Interpolant-based Compositional Model Checking

Lemma 5. *The counterexample ce will be not feasible in the new assumption.*

The whole algorithm, named AAG (asynchronous assume-guarantee), is presented in Fig. 5. For the algorithm, we have the following theorem:

Theorem 2. *The algorithm will terminate. When the algorithm terminates, if $M_1 \parallel M_2 \not\models G$, a real counterexample will be returned and, otherwise, an assumption A such that $M_2 \preceq^{\mathcal{J}} A$ and $M_1 \parallel A \models G$ will be found.*

Proof. Termination. If a real counterexample or an assumption such that $M_2 \preceq^{\mathcal{J}} A$ and $M_1 \parallel A \models G$ is found, the algorithm will terminate. Otherwise, the assumption A converges to M_2 in at most $|\bar{\eta}_2|$ iterations and the algorithm terminates.

When the algorithm terminates, $M_1 \parallel M_2 \not\models G$ if a real counterexample is returned. Otherwise, an assumption satisfying the two premises of rule (2) will be reached. The correctness of $M_1 \parallel M_2 \models G$ is guaranteed by theorem 1. □

5 Experimental Results

Our algorithm AAG, presented in Fig. 5, has been implemented in the C language. We use NuSMV [29] to check the premise $M_1 \parallel A \models G$ and MathSAT [23] to, given two inconsistent clauses, calculate an interpolant which will be added to the set of existing predicates. To make a comparison with learning-based algorithms, we also adapted the CDNF-based approach proposed in [11] to learn the Boolean initial and transition predicates of an assumption. The work is selected to be compared with because it also uses shared variable structures, rather than LTSs, as system models although it is purely focused on synchronous

Table 1. Experimental Results

Problems	Truth	CDNF					AAG			
		MQs	EQs	$\|\zeta_A\|$	$\|\tau_A\|$	Time(s)	ARs	$\|\zeta_A\|$	$\|\tau_A\|$	Time(s)
Inverter-1-2	T	221	45	2	9	0.49	1	2	9	0.03
Inverter-2-2	F	99	20	2	9	0.18	1	1	7	0.02
Inverter-3-2	T	211	45	2	9	0.53	1	2	9	0.04
Inverter-4-2	F	99	20	2	9	0.22	1	1	7	0.01
Inverter-1-4	T	1393	184	4	15	7.13	4	5	17	0.10
Inverter-2-4	F	749	98	2	7	2.19	1	1	9	0.03
Exclusive-3-1	T	348	53	4	11	0.17	1	2	7	0.15
Exclusive-2-2	T	5263	396	4	13	10.67	5	6	17	0.2
Exclusive-3-3	T	×	×	×	×	×	8	8	22	0.66

parallel compositions. The authors also showed that their approach outperforms interpolation-based monolithic model checking [30].

The examples we consider are systems consisting of multiple threads, i.e. $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$ for some finite n. We arbitrarily divide such a system into two sub-systems, say by composing the first i threads ($M_1 \parallel \cdots \parallel M_i$) and composing the rest of the threads ($M_{i+1} \parallel \cdots \parallel M_n$) for some $1 \leq i \leq n$. These two composed models serve as M_1 and M_2 of the inference rule (2). Several auxiliary variables are introduced in M_2 to ensure that the executions are fair in the sense that every enabled thread of $M_{i+1} \parallel \cdots \parallel M_n$ will be executed infinitely often. We use the notation *XXXX-a-b* to denote the above described partition of a verification problem, where *XXXX* is the name of the problem, a and b denote the number of threads in M_1 and M_2, respectively. The experimental systems used here are the asynchronous version of a *ring* of inverters and the semaphore-based exclusive access, which are distributed with NuSMV. The property verified for the first example states that any inverter will infinitely often output data and receive data for its neighbors, while the property for the second one states that no two processes are in the critical section at the same time.

The experimental results are summarized in Table 1. The CDNF algorithm needs to ask membership queries (MQs) to a teacher on whether the initial predicate or transition predicate evaluates to true for a given valuation to Boolean variables. By asking an equivalence query (EQ), the learning algorithm can get an affirmative answer, i.e. the submitted conjecture is an assumption satisfying the premises of the inference rule, or a counterexample. In our AAG algorithm, only abstraction refinements (ARs) are necessary. The size of generated assumptions are measured in the number of Boolean variables of ζ_A and τ_A, denoted as $|\zeta_A|$ and $|\tau_A|$, respectively. The execution time is measured in seconds. The columns labelled with crosses indicate that the CDNF-based algorithm does not terminate in 30 minutes. The experiments were run on a Macbook Pro laptop with a 2.2 GHz Intel Core i7 CPU and 4GB of memory running Mac OS X.

The experiment results confirm the results presented in [10], showing that a learning algorithm takes normally 90 percent of the time. The learning algorithms

CDNF asks a huge amount of queries to learn a formula. Normally, answering a membership query needs to solve a SAT problem or do a simulation check, while equivalence queries are more specific to the application domain and tend to be even more expensive. On the contrary, the AAG algorithm just "mechanically" calculates an abstraction and then checks if the second premise of the inference rule is satisfied. As discussed above, it is required that $\eta_A = \eta_2$ in the CDNF-based assume-guarantee learning. It is possible to obtain an assumption smaller than M_2 only when the verified property does not hold. As our algorithm introduces an abstraction over not shared variables of M_2, the generated assumption can be smaller than M_2 even if a property holds, as shown in some cases. It's also possible that the generated assumption is greater than M_2 when some *redundant predicates* are produced. A predicate is redundant if it is implied by the conjunction of some predicates that are produced later. In general, our approach outperforms the CDNF-based assume-guarantee reasoning.

6 Conclusion

As a promising technique to tackle the state explosion problem, compositional verification of concurrent systems based on assume-guarantee reasoning has been studied extensively. Inference rules play a key role in assume-guarantee reasoning as they tell how to verify a system by checking its constituents. However, the most widely used inference rule used in the literature has only been proved for synchronous systems. Based on a new simulation relation introduced in this paper, we prove that the rule holds for asynchronous systems as long as the alphabets of the components satisfy certain constraints. Then, an automating assumption generation approach is proposed based on counterexample-guided abstraction refinement, rather than using learning algorithms. Our approach is compared with the CDNF-based assume-guarantee reasoning algorithm.

Although only safety properties are considered in Section 4, the inference rule (2) allows liveness properties. The techniques given in [20] for identifying spurious loop counterexamples can be used for refining abstractions when liveness properties are taken into account. In addition, some lazy or approximate abstraction strategies might replace the exact existential abstraction used in our current implementation, which is normally very expensive.

Acknowledgements. The work was partially supported by the National Natural Science Foundation of China under grant Nos. 60903051, 61003028 and the Knowledge Innovation Program of the Chinese Academy of Sciences under grant No. ISCAS2009-DR09.

The first author's academic visit to the Computer Science Department of Carnegie Mellon University is supported by the Foundation for Selected Young Scientists Studying Abroad, Chinese Academy of Sciences. Thanks are also due to the Carnegie Mellon University for providing the infrastructure during the first author's visit.

References

1. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems, pp. 123–144. Springer-Verlag New York, Inc., New York (1985)
2. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Trans. Program. Lang. Syst. 16, 843–871 (1994)
3. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning Assumptions for Compositional Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
4. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Assumption generation for software component verification. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE 2002, p. 3. IEEE Computer Society, Washington, DC (2002)
5. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
6. Barringer, H., Giannakopoulou, D.: Proof rules for automated compositional verification through learning. In: Proc. SAVCBS Workshop, pp. 14–21 (2003)
7. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Assumption generation for software component verification. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE 2002, pp. 3–12. IEEE Computer Society, Washington, DC (2002)
8. Bobaru, M.G., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: [31], pp. 135–148
9. Păsăreanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Form. Methods Syst. Des. 32, 175–205 (2008)
10. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. ACM Trans. Softw. Eng. Methodol. 17(2), 7:1–7:52 (2008)
11. Chen, Y.F., Clarke, E.M., Farzan, A., Tsai, M.H., Tsay, Y.K., Wang, B.Y.: Automated Assume-Guarantee Reasoning through Implicit Learning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 511–526. Springer, Heidelberg (2010)
12. Bshouty, N.H.: Exact learning boolean functions via the monotone theory. Inf. Comput. 123, 146–153 (1995)
13. Chaki, S., Gurfinkel, A.: Automated assume-guarantee reasoning for omega-regular systems and specifications. Innov. Syst. Softw. Eng. 7, 131–139 (2011)
14. Chaki, S., Strichman, O.: Optimized L*-Based Assume-Guarantee Reasoning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 276–291. Springer, Heidelberg (2007)
15. Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated Assume-Guarantee Reasoning by Abstraction Refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
16. Alur, R., Madhusudan, P., Nam, W.: Symbolic Compositional Verification by Learning Assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005)
17. Sinha, N., Clarke, E.: SAT-based compositional verification using lazy learning. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 39–54. Springer, Heidelberg (2007)

18. Gupta, A., Mcmillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Form. Methods Syst. Des. 32(3), 285–301 (2008)
19. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning Minimal Separating DFA's for Compositional Verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
20. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
21. Bobaru, M.G., Pasareanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: [31], pp. 135–148
22. Komuravelli, A., Păsăreanu, C.S., Clarke, E.M.: Assume-Guarantee Abstraction Refinement for Probabilistic Systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 310–326. Springer, Heidelberg (2012)
23. MathSAT, `http://mathsat.fbk.eu/`
24. iZ3, `http://research.microsoft.com/en-us/um/redmond/projects/z3/iz3.html`
25. Bonet, M.L., Pitassi, T., Raz, R.: Lower bounds for cutting planes proofs with small coefficients. J. Symb. Log. 62(3), 708–728 (1997)
26. Milner, R.: An algebraic definition of simulation between programs. Technical report, Stanford, CA, USA (1971)
27. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
28. Gheorghiu, M., Giannakopoulou, D., Păsăreanu, C.S.: Refining Interface Alphabets for Compositional Verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007)
29. NuSMV, `http://nusmv.fbk.eu/`
30. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
31. Gupta, A., Malik, S. (eds.): CAV 2008. LNCS, vol. 5123. Springer, Heidelberg (2008)