

# Learning to Divide and Conquer

Dimitra Giannakopoulou

*RIACS, NASA Ames Research Center, N269-230, Moffett Field, CA 94035, USA*

Corina S. Păsăreanu

*Perot Systems, NASA Ames Research Center, N269-230, Moffett Field, CA 94035, USA*

Mihaela Gheorghiu Bobaru

*Department of Computer Science, University of Toronto, 10 King's College Road, Toronto,  
Ontario, CANADA M5S 3G4*

Jamieson M. Cobleigh

*The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760, USA*

Howard Barringer

*School of Computer Science, University of Manchester, Oxford Road, Manchester  
M13 9PL, UK*

---

## 1 Abstract

2 Assume-guarantee reasoning is a “divide-and-conquer” approach to the verification of large  
3 systems that makes use of *assumptions* about the environment of that system’s compo-  
4 nents. Developing appropriate assumptions used to be a difficult and manual process. Over  
5 the past five years, we have developed a framework for performing assume-guarantee  
6 verification of systems in an incremental and fully automated fashion. The framework  
7 uses an off-the-shelf learning algorithm to compute the assumptions. The assumptions are  
8 initially approximate and become more precise by means of counterexamples obtained  
9 by model checking components separately. The framework supports different assume-  
10 guarantee rules, both symmetric and non-symmetric. Moreover, we have recently intro-  
11 duced *alphabet refinement*, which extends the assumption learning process to also infer  
12 *assumption alphabets*. This refinement technique starts with assumption alphabets that are  
13 a subset of the minimal interface between a component and its environment, and adds ac-  
14 tions to it as necessary until a given property is shown to hold or to be violated in the  
15 system. We have applied the learning framework to a number of case studies that show that  
16 compositional verification by learning assumptions can be significantly more scalable than  
17 non-compositional verification.

18 *Key words:* Assume guarantee, model checking, labeled transition systems, learning,  
19 proof rules, compositional verification.

---

## 20 1 Introduction

21 Model checking is an effective technique for finding subtle errors in concurrent  
22 systems. Given a finite model of a system and a required property of that system,  
23 model checking determines automatically whether the property is satisfied by the  
24 system. The cost of model checking techniques may be exponential in the size of  
25 the system being verified, a problem known as state explosion [10]. This can make  
26 model checking intractable for systems of realistic size.

27 Compositional verification techniques address the state-explosion problem by us-  
28 ing a “divide-and-conquer” approach: properties of the system are decomposed  
29 into properties of its components and each component is then checked separately.  
30 In checking components individually, it is often necessary to incorporate some  
31 knowledge of the context in which each component is expected to operate correctly.

---

*Email addresses:* dimitra@email.arc.nasa.gov (Dimitra Giannakopoulou),  
pcorina@email.arc.nasa.gov (Corina S. Pasăreanu), mg@cs.toronto.edu  
(Mihaela Gheorghiu Bobaru), Jamieson.Cobleigh@mathworks.com (Jamieson  
M. Cobleigh), howard.barringer@manchester.ac.uk (Howard Barringer).

1 Assume-guarantee reasoning [20,25] addresses this issue by using *assumptions* that  
2 capture the expectations that a component makes about its environment. Assump-  
3 tions have traditionally been developed manually, which has limited the practical  
4 impact of assume-guarantee reasoning.

5 To address this problem, we have proposed a framework [13] that *fully automates*  
6 assume-guarantee model checking of safety properties for finite labeled transition  
7 systems. At the heart of this framework lies an off-the-shelf learning algorithm,  
8 namely  $L^*$  [4], that is used to compute the assumptions. In one instantiation of this  
9 framework, a safety property  $P$  is verified on a system consisting of components  
10  $M_1$  and  $M_2$  by learning an assumption under which  $M_1$  satisfies  $P$ . This assump-  
11 tion is then discharged by showing it is satisfied by  $M_2$ . In [5] we extended the  
12 learning framework to support a set of novel symmetric assume-guarantee rules  
13 that are sound and complete. In all cases, this learning based framework is guaran-  
14 teed to terminate, either stating that the property holds for the system, or returning  
15 a counterexample if the property is violated.

16 Compositional techniques have been shown particularly effective for well-  
17 structured systems that have small interfaces between components [7,17]. Inter-  
18 faces consist of *all* communication points through which components may influ-  
19 ence each other's behavior. In our initial presentations of the framework [13,5] the  
20 alphabets of the assumption automata included *all* the actions in the component  
21 interface. In a case study presented in [24], however, we observed that a smaller al-  
22 phabet can be sufficient to prove a property. This smaller alphabet was determined  
23 through manual inspection and with it, assume-guarantee reasoning achieves or-  
24 ders of magnitude improvement over monolithic (*i.e.*, non-compositional) model  
25 checking [24].

26 Motivated by the successful use of a smaller assumption alphabet in learning, we  
27 investigated in [16] whether the process of discovering a smaller alphabet that is  
28 sufficient for checking the desired properties can be automated. Smaller alphabets  
29 mean smaller interfaces among components, which may lead to smaller assump-  
30 tions, and hence to smaller verification problems. We developed an *alphabet re-*  
31 *finement* technique that extends the learning framework so that it starts with a small  
32 subset of the interface alphabet and adds actions into it as necessary until a required  
33 property is either shown to hold or shown to be violated in the system. Actions to  
34 be added are discovered by analysis of the counterexamples obtained from model  
35 checking the components.

36 The learning framework and the alphabet refinement have been implemented within  
37 the LTSA model checking tool [22] and they have been effective in verifying real-  
38 istic concurrent systems, such as the ones developed in NASA projects. This paper  
39 presents and expands the material presented in [13] (original learning framework  
40 for automated assume-guarantee reasoning with an asymmetric rule), [5] (learning  
41 for symmetric rules), and [16] (alphabet refinement for original framework). In ad-

1 dition we describe here a new extension with a circular rule, alphabet refinement  
2 for symmetric and circular rules, and present new experimental data.

3 The rest of the paper is organized as follows. Section 2 provides background on  
4 labeled transition systems, finite state machines, assume guarantee reasoning and  
5 the  $L^*$  algorithm. Section 3 follows with a presentation of the learning framework  
6 that automates assume guarantee reasoning for asymmetric and circular rules. Sec-  
7 tion 4 presents the extension of the framework with symmetric rules, followed by  
8 Section 5 which presents the algorithm for interface alphabet refinement. Section 6  
9 provides an experimental evaluation of the described techniques. Section 7 surveys  
10 related work and Section 8 concludes the paper.

## 11 2 Preliminaries

12 In this section we give background information for our work: we introduce labeled  
13 transition systems and finite state machines, together with their associated opera-  
14 tors, and also present how properties are expressed and checked in this context. We  
15 also introduce assume-guarantee reasoning and the notion of weakest assumption  
16 that is used in our learning framework. Moreover we provide a detailed descrip-  
17 tion of the learning algorithm that we use to automate assume-guarantee reasoning.  
18 The reader may wish to skip this section on the first reading.

### 19 2.1 Labeled Transition Systems (LTSs)

20 Let  $\mathcal{Act}$  be the universal set of observable actions and let  $\tau$  denote a local action *un-*  
21 *observable* to a component's environment. We use  $\pi$  to denote a special *error state*,  
22 which models the fact that a safety violation has occurred in the associated transi-  
23 tion system. We require that the error state has no outgoing transitions. Formally,  
24 an LTS  $M$  is a four tuple  $\langle Q, \alpha M, \delta, q_0 \rangle$  where:

- 25 •  $Q$  is a finite non-empty set of states
- 26 •  $\alpha M \subseteq \mathcal{Act}$  is a set of observable actions called the *alphabet* of  $M$
- 27 •  $\delta \subseteq Q \times \alpha M \cup \{\tau\} \times Q$  is a transition relation
- 28 •  $q_0 \in Q$  is the initial state

29 We use  $\Pi$  to denote the LTS  $\langle \{\pi\}, \mathcal{Act}, \emptyset, \pi \rangle$ . An LTS  $M = \langle Q, \alpha M, \delta, q_0 \rangle$  is *non-*  
30 *deterministic* if it contains  $\tau$ -transitions or if  $\exists (q, a, q'), (q, a, q'') \in \delta$  such that  
31  $q' \neq q''$ . Otherwise,  $M$  is *deterministic*.

32 Consider a simple communication channel that consists of two components whose  
33 LTSs are shown in Fig. 1. Note that the initial state of all LTSs in this paper is state  
34 0. The *Input* LTS receives an input when the action *input* occurs, and then sends it

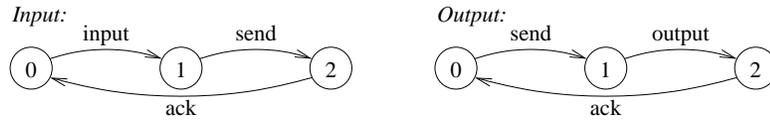


Fig. 1. Example LTSs

1 to the *Output* LTS with action *send*. After some data is sent to it, *Output* produces  
 2 output using the action *output* and acknowledges that it has finished, by using the  
 3 action *ack*. At this point, both LTSs return to their initial states so the process can  
 4 be repeated.

### 5 2.1.1 Traces

6 A *trace*  $t$  of an LTS  $M$  is a sequence of observable actions that  $M$  can perform  
 7 starting at its initial state. For example,  $\langle \text{input} \rangle$  and  $\langle \text{input}, \text{send} \rangle$  are both traces of  
 8 the *Input* LTS in Fig. 1. We sometimes abuse this notation and denote by  $t$  both  
 9 a trace and its trace LTS. For a trace  $t$  of length  $n$ , its trace LTS consists of  $n + 1$   
 10 states, where there is a transition between states  $m$  and  $m + 1$  on the  $m^{\text{th}}$  action in  
 11 the trace  $t$ . The set of all traces of an LTS  $M$  is the language of  $M$  and is denoted  
 12  $\mathcal{L}(M)$ .

13 For  $\Sigma \subseteq \mathcal{Act}$ , we use  $t \upharpoonright \Sigma$  to denote the trace obtained by removing from  $t$  all  
 14 occurrences of actions  $a \notin \Sigma$ . Similarly,  $M \upharpoonright \Sigma$  is defined to be an LTS over alpha-  
 15 bet  $\Sigma$  which is obtained from  $M$  by renaming to  $\tau$  all the transitions labeled with  
 16 actions that are not in  $\Sigma$ . Let  $t, t'$  be two traces. Let  $\Sigma, \Sigma'$  be the sets of actions  
 17 occurring in  $t, t'$ , respectively. By the *symmetric difference* of  $t$  and  $t'$  we mean the  
 18 symmetric difference of sets  $\Sigma$  and  $\Sigma'$ .

### 19 2.1.2 Parallel Composition

20 Let  $M = \langle Q, \alpha M, \delta, q_0 \rangle$  and  $M' = \langle Q', \alpha M', \delta', q'_0 \rangle$ . We say that  $M$  *transits* into  
 21  $M'$  with action  $a$ , denoted  $M \xrightarrow{a} M'$ , if and only if  $(q_0, a, q'_0) \in \delta$  and either  
 22  $Q = Q', \alpha M = \alpha M'$ , and  $\delta = \delta'$  for  $q'_0 \neq \pi$ , or, in the special case where  $q'_0 = \pi$ ,  
 23  $M' = \Pi$ .

24 The parallel composition operator  $\parallel$  is a commutative and associative operator that  
 25 combines the behavior of two components by synchronizing the actions common to  
 26 their alphabets and interleaving the remaining actions. For example, in the parallel  
 27 composition of the *Input* and *Output* components from Fig. 1, actions *send* and  
 28 *ack* will each be synchronized while *input* and *output* will be interleaved.

29 Formally, let  $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$  and  $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$  be two LTSs.  
 30 If  $M_1 = \Pi$  or  $M_2 = \Pi$ , then  $M_1 \parallel M_2 = \Pi$ . Otherwise,  $M_1 \parallel M_2$  is an LTS  
 31  $M = \langle Q, \alpha M, \delta, q_0 \rangle$ , where  $Q = Q^1 \times Q^2$ ,  $q_0 = (q_0^1, q_0^2)$ ,  $\alpha M = \alpha M_1 \cup \alpha M_2$ , and

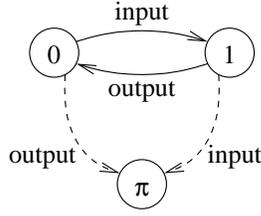


Fig. 2. *Order* Property

- 1  $\delta$  is defined as follows, where  $a$  is either an observable action or  $\tau$  (note that the  
 2 symmetric rules are implied by the fact that the operator is commutative):

$$3 \quad \frac{M_1 \xrightarrow{a} M'_1, a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M'_1 \parallel M_2} \quad \frac{M_1 \xrightarrow{a} M'_1, M_2 \xrightarrow{a} M'_2, a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M'_1 \parallel M'_2}$$

#### 4 2.1.3 Properties

- 5 We call a deterministic LTS that contains no  $\pi$  states a *safety LTS*. A *safety property*  
 6 is specified as a safety LTS  $P$ , whose language  $\mathcal{L}(P)$  defines the set of acceptable  
 7 behaviors over  $\alpha P$ . For LTS  $M$  and safety LTS  $P$  such that  $\alpha P \subseteq \alpha M$ ,  $M$  satisfies  
 8  $P$ , denoted as  $M \models P$ , if and only if  $\forall \sigma \in \mathcal{L}(M) : (\sigma \upharpoonright \alpha P) \in \mathcal{L}(P)$ .

When checking a property  $P$ , an *error LTS* denoted  $P_{err}$  is created, which traps possible violations with the  $\pi$  state. Formally, the error LTS of a property  $P = \langle Q, \alpha P, \delta, q_0 \rangle$  is  $P_{err} = \langle Q \cup \{\pi\}, \alpha P_{err}, \delta', q_0 \rangle$ , where  $\alpha P_{err} = \alpha P$  and

$$\delta' = \delta \cup \{(q, a, \pi) \mid q \in Q, a \in \alpha P, \text{ and } \nexists q' \in Q : (q, a, q') \in \delta\}$$

- 9 Note that the error LTS is *complete*, meaning each state other than the error state  
 10 has outgoing transitions for every action in the alphabet.

- 11 For example, the *Order* property shown in Fig. 2 captures a desired behavior of  
 12 the communication channel shown in Fig. 1. The property comprises states 0, 1 and  
 13 the transitions denoted by solid arrows. It expresses the fact that inputs and outputs  
 14 come in matched pairs, with the input always preceding the output. The dashed  
 15 arrows illustrate the transitions to the error state that are added to the property to  
 16 obtain its error LTS,  $Order_{err}$ .

- 17 To detect violations of property  $P$  by component  $M$ , the parallel composition  
 18  $M \parallel P_{err}$  is computed. It has been proved that  $M$  violates  $P$  if and only if the  
 19  $\pi$  state is reachable in  $M \parallel P_{err}$  [?]. For example, state  $\pi$  is not reachable in  
 20  $Input \parallel Output \parallel Order_{err}$ , so we conclude that  $Input \parallel Output \models Order$ .

## 1 2.2 LTSs and Finite-State Machines

2 As described in Section 4, some of the assume-guarantee rules require the use of the  
3 “complement” of an LTS. LTSs are not closed under complementation, so we need  
4 to define here a more general class of finite-state machines (FSMs) and associated  
5 operators for our framework.

6 An FSM  $M$  is a five tuple  $\langle Q, \alpha M, \delta, q_0, F \rangle$  where  $Q$ ,  $\alpha M$ ,  $\delta$ , and  $q_0$  are defined as  
7 for LTSs, and  $F \subseteq Q$  is a set of accepting states.

8 For an FSM  $M$  and a trace  $t$ , we use  $\hat{\delta}(q, t)$  to denote the set of states that  $M$  can  
9 reach after reading  $t$  starting at state  $q$ . A trace  $t$  is said to be *accepted* by an FSM  
10  $M = \langle Q, \alpha M, \delta, q_0, F \rangle$  if  $\hat{\delta}(q_0, t) \cap F \neq \emptyset$ . The *language accepted by  $M$* , denoted  
11  $\mathcal{L}(M)$  is the set  $\{t \mid \hat{\delta}(q_0, t) \cap F \neq \emptyset\}$ .

12 For an FSM  $M = \langle Q, \alpha M, \delta, q_0, F \rangle$ , we use  $\text{LTS}(M)$  to denote the LTS  
13  $\langle Q, \alpha M, \delta, q_0 \rangle$  defined by its first four fields. Note that this transformation does  
14 not preserve the language of the FSM, *i.e.*, in some cases  $\mathcal{L}(M) \neq \mathcal{L}(\text{LTS}(M))$ .  
15 On the other hand, an LTS is in fact a special instance of an FSM, since it can be  
16 viewed as an FSM for which all states are accepting. From now on, whenever we  
17 apply operators between FSMs and LTSs, it is implied that the LTS is treated as its  
18 corresponding FSM.

19 We call an FSM  $M$  *deterministic* if and only if  $\text{LTS}(M)$  is deterministic.

### 20 2.2.1 Parallel Composition of FSMs

21 Let  $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1, F^1 \rangle$  and  $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2, F^2 \rangle$  be two FSMs.  
22 Then  $M_1 \parallel M_2$  is an FSM  $M = \langle Q, \alpha M, \delta, q_0, F \rangle$ , where:

- 23 •  $\langle Q, \alpha M, \delta, q_0 \rangle = \text{LTS}(M_1) \parallel \text{LTS}(M_2)$ , and
- 24 •  $F = \{(s^1, s^2) \in Q^1 \times Q^2 \mid s^1 \in F^1 \text{ and } s^2 \in F^2\}$ .

#### 25 **Note 1**

26  $\mathcal{L}(M_1 \parallel M_2) = \{t \mid t \upharpoonright \alpha M_1 \in \mathcal{L}(M_1) \wedge t \upharpoonright \alpha M_2 \in \mathcal{L}(M_2) \wedge t \in (\alpha M_1 \cup \alpha M_2)^*\}$

### 27 2.2.2 Properties

For FSMs  $M$  and  $P$  where  $\alpha P \subseteq \alpha M$ ,  $M \models P$  if and only if

$$\forall t \in \mathcal{L}(M) : t \upharpoonright \alpha P \in \mathcal{L}(P)$$

### 1 2.2.3 Complementation

2 The complement of an FSM (or an LTS)  $M$ , denoted  $coM$ , is an FSM that accepts  
3 the complement of  $M$ 's language. It is constructed by first making  $M$  determinis-  
4 tic, subsequently completing it with respect to  $\alpha M$ , and finally turning all accepting  
5 states into non-accepting ones, and vice-versa. An automaton is complete with re-  
6 spect to some alphabet if every state has an outgoing transition for each action in  
7 the alphabet. Completion typically introduces a non-accepting state and appropriate  
8 transitions to that state.

## 9 2.3 Assume Guarantee Reasoning

### 10 2.3.1 Assume Guarantee Triples

11 In the assume-guarantee paradigm a formula is a triple  $\langle A \rangle M \langle P \rangle$ , where  $M$  is a  
12 component,  $P$  is a property, and  $A$  is an assumption about  $M$ 's environment. The  
13 formula is true if whenever  $M$  is part of a system satisfying  $A$ , then the system  
14 must also guarantee  $P$  [18,25], i.e.,  $\forall E, E \parallel M \models A$  implies  $E \parallel M \models P$ . For  
15 LTS  $M$  and safety LTSs  $A$  and  $P$ , checking  $\langle A \rangle M \langle P \rangle$  reduces to checking if state  
16  $\pi$  is reachable in  $A \parallel M \parallel P_{err}$ . Note that when  $\alpha P \subseteq \alpha A \cup \alpha M$ , this is equivalent  
17 to  $A \parallel M \models P$ .

18 **Theorem 1**  $\langle A \rangle M \langle P \rangle$  is true if and only if  $\pi$  is unreachable in  $A \parallel M \parallel P_{err}$ .

### 19 PROOF.

20 • **“if part”** Assume  $\langle A \rangle M \langle P \rangle$  is true. We need to show that  $\pi$  is unreachable  
21 in  $A \parallel M \parallel P_{err}$ . We prove this by contradiction. Assume  $\pi$  is reachable in  
22  $A \parallel M \parallel P_{err}$  by a trace  $t$ . As a result,  $t \upharpoonright \alpha A \in \mathcal{L}(A)$ ,  $t \upharpoonright \alpha M \in \mathcal{L}(M)$ , and  
23  $t \upharpoonright \alpha P \in \mathcal{L}(P_{err})$  (see Note 1 on page 7).

24 Let  $E$  be the trace LTS for the trace  $t \upharpoonright \alpha A$ , but augment its alphabet so  
25 that  $E \parallel M \models A$  and  $E \parallel M \models P$  are well defined, i.e.,  $\alpha A \subseteq (\alpha M \cup \alpha E)$   
26 and  $\alpha P \subseteq (\alpha M \cup \alpha E)$ . By construction,  $\mathcal{L}(E)$  consists of  $t \upharpoonright \alpha A$  and all of  
27 its prefixes. Since  $t \upharpoonright \alpha A \in \mathcal{L}(A)$ , we can conclude that  $E \models A$ . As a result,  
28  $E \parallel M \models A$ .

29 From our hypothesis that  $\langle A \rangle M \langle P \rangle$  is true, the fact that  $E \parallel M \models A$   
30 implies that  $E \parallel M \models P$ . However,  $t \upharpoonright \alpha E \in \mathcal{L}(E)$ ,  $t \upharpoonright \alpha M \in \mathcal{L}(M)$ , and  
31  $t \upharpoonright \alpha P \in \mathcal{L}(P_{err})$ . This means that  $\pi$  is reachable in  $E \parallel M \parallel P_{err}$  (on trace  $t$ ).  
32 As a result, we can conclude that  $E \parallel M \not\models P$ , which is a contradiction.

33 So, our original assumption that  $\pi$  is reachable in  $A \parallel M \parallel P_{err}$  is incorrect.  
34 Thus,  $\pi$  is not reachable in  $A \parallel M \parallel P_{err}$ , as desired.

35

1 • **“only if part”** Assume  $\pi$  is unreachable in  $A \parallel M \parallel P_{err}$ . We need to show that  
2  $\langle A \rangle M \langle P \rangle$ . We again prove by contradiction. Assume  $\langle A \rangle M \langle P \rangle$  is not true,  
3 *i.e.*, assume  $\exists E$  such that  $E \parallel M \models A$  but  $E \parallel M \not\models P$ . (Again, we assume that  
4  $\alpha E$  is constructed such that  $\models$  is well defined in the previous sentence.)  
5 Since  $E \parallel M \not\models P$  then  $\pi$  is reachable in  $E \parallel M \parallel P_{err}$  by some trace  
6  $t$ . As a result,  $t \upharpoonright \alpha E \in \mathcal{L}(E)$ ,  $t \upharpoonright \alpha M \in \mathcal{L}(M)$ , and  $t \upharpoonright \alpha P \in \mathcal{L}(P_{err})$ . Since  
7  $E \parallel M \models A$  and  $\alpha A \subseteq \alpha E$ , it follows that  $t \upharpoonright \alpha A \in \mathcal{L}(A)$ . As a result,  $\pi$  is  
8 reachable in  $A \parallel M \parallel P_{err}$  by  $t \upharpoonright (\alpha A \cup \alpha M \cup \alpha P)$ , which is a contradiction.  
9 Thus,  $\langle A \rangle M \langle P \rangle$  is true, as desired.  $\square$

### 10 2.3.2 Weakest Assumption

11 A central notion of our work is that of the *weakest assumption* [17], defined for-  
12 mally here.

13 **Definition 2 (Weakest Assumption for  $\Sigma$ )** Let  $M_1$  be an LTS for a component,  
14  $P$  be a safety LTS for a property required of  $M_1$ , and  $\Sigma$  be the interface of the  
15 component to the environment. The weakest assumption  $A_{w,\Sigma}$  of  $M_1$  for  $\Sigma$  and  
16 for property  $P$  is a deterministic LTS such that: 1)  $\alpha A_{w,\Sigma} = \Sigma$ , and 2) for any  
17 component  $M_2$  such that  $\Sigma \subseteq \alpha M_2$ ,  $M_1 \parallel (M_2 \upharpoonright \Sigma) \models P$  iff  $M_2 \models A_{w,\Sigma}$

18 The notion of a weakest assumption depends on the interface between the compo-  
19 nent and its environment. Accordingly, projection of  $M_2$  to  $\Sigma$  forces  $M_2$  to com-  
20 municate with our module only through  $\Sigma$  (second condition above). In [17] we  
21 showed that weakest assumptions exist for components expressed as LTSs, safety  
22 properties expressed as safety LTSs, and provided an algorithm for computing these  
23 assumptions.

24 The definition above refers to *any* environment component  $M_2$  that interacts with  
25 component  $M_1$  via an alphabet  $\Sigma$ . When  $M_2$  is given, there is a natural notion of  
26 the complete *interface* between  $M_1$  and its environment  $M_2$ , when property  $P$  is  
27 checked.

28 **Definition 3 (Interface Alphabet)** Let  $M_1$  and  $M_2$  be component LTSs, and  
29  $P$  be a safety LTS. The interface alphabet  $\Sigma_I$  of  $M_1$  is defined as:  
30  $\Sigma_I = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$ .

31 **Definition 4 (Weakest Assumption)** Given  $M_1$ ,  $M_2$  and  $P$  as above, the weakest  
32 assumption  $A_w$  is defined as  $A_{w,\Sigma_I}$ .

33 Note that, to deal with any system-level property, we allow properties in Defini-  
34 tion 3 to include actions that are not in  $\alpha M_1$  but are in  $\alpha M_2$ . These actions need  
35 to be in the interface since they are controllable by  $M_2$ . Moreover from the above  
36 definitions, it follows that  $M_1 \parallel M_2 \models P$  iff  $M_2 \models A_w$ .

```

(1) let  $S = E = \{\lambda\}$ 
    loop {
(2)   Update  $T$  using queries
      while  $(S, E, T)$  is not closed {
(3)     Add  $sa$  to  $S$  to make  $S$  closed where  $s \in S$  and  $a \in \Sigma$ 
(4)     Update  $T$  using queries
      }
(5)   Construct candidate DFSM  $C$  from  $(S, E, T)$ 
(6)   Make the conjecture  $C$ 
(7)   if  $C$  is correct return  $C$ 
      else
(8)     Add  $e \in \Sigma^*$  that witnesses the counterexample to  $E$ 
    }

```

Fig. 3. The  $L^*$  Algorithm

## 1 2.4 The $L^*$ Learning Algorithm

2  $L^*$  was developed by Angluin [4] and later improved by Rivest and Schapire [26].  
3  $L^*$  learns an unknown regular language  $U$  over alphabet  $\Sigma$  and produces a deter-  
4 ministic finite state machine (DFSM) that accepts it.  $L^*$  interacts with a *Minimally*  
5 *Adequate Teacher*, henceforth referred to as a *Teacher*, that answers two types of  
6 questions from  $L^*$ . The first type of question is a *membership query*, in which  $L^*$   
7 asks whether a string  $s \in \Sigma^*$  is in  $U$ . The second type of question is a *conjec-*  
8 *ture*, in which  $L^*$  asks whether a conjectured DFSM  $C$  is one where  $\mathcal{L}(C) = U$ .  
9 If  $\mathcal{L}(C) \neq U$  the Teacher returns a counterexample, which is a string  $s$  in the  
10 symmetric difference of  $\mathcal{L}(C)$  and  $U$ .

11 At a higher level,  $L^*$  creates a table where it incrementally records whether strings  
12 in  $\Sigma^*$  belong to  $U$ . It does this by making membership queries to the Teacher. At  
13 various stages  $L^*$  decides to make a conjecture. It constructs a candidate automaton  
14  $C$  based on the information contained in the table and asks the Teacher whether  
15 the conjecture is correct. If it is, the algorithm terminates. Otherwise,  $L^*$  uses the  
16 counterexample returned by the Teacher to extend the table with strings that witness  
17 differences between  $\mathcal{L}(C)$  and  $U$ .

### 18 2.4.1 Details of $L^*$

In the following more detailed presentation of the algorithm, line numbers refer to  $L^*$ 's illustration in Fig. 3.  $L^*$  builds an observation table  $(S, E, T)$  where  $S$  and  $E$  are a set of prefixes and suffixes, respectively, both over  $\Sigma$ . In addition,  $T$  is a function mapping  $(S \cup S \cdot \Sigma) \cdot E$  to  $\{\text{true}, \text{false}\}$ , where the operator “ $\cdot$ ” is defined as follows. Given two sets of sequences of actions  $P$  and  $Q$ ,  $P \cdot Q = \{pq \mid p \in P \text{ and } q \in Q\}$ , where  $pq$  represents the concatenation of the se-

quences  $p$  and  $q$ . Initially,  $L^*$  sets  $S$  and  $E$  to  $\{\lambda\}$  (line 1), where  $\lambda$  represents the empty string. Subsequently, it updates the function  $T$  by making membership queries so that it has a mapping for every string in  $(S \cup S \cdot \Sigma) \cdot E$  (line 2). It then checks whether the observation table is *closed*, *i.e.*, whether

$$\forall s \in S, \forall a \in \Sigma, \exists s' \in S, \forall e \in E : T(sae) = T(s'e)$$

1 If  $(S, E, T)$  is not closed, then  $sa$  is added to  $S$  where  $s \in S$  and  $a \in \Sigma$  are the  
 2 elements for which there is no  $s' \in S$  (line 3). Once  $sa$  has been added to  $S$ ,  $T$   
 3 needs to be updated (line 4). Lines 3 and 4 are repeated until  $(S, E, T)$  is closed.

4 Once the observation table is closed, a candidate DFSM  $C = \langle Q, \alpha C, \delta, q_0, F \rangle$  is  
 5 constructed (line 5), with states  $Q = S$ , initial state  $q_0 = \lambda$ , and alphabet  $\alpha C = \Sigma$ ,  
 6 where  $\Sigma$  is the alphabet of the unknown language  $U$ . The set  $F$  consists of the states  
 7  $s \in S$  such that  $T(s) = \text{true}$ . The transition relation  $\delta$  is defined as  $\delta(s, a) = s'$   
 8 where  $\forall e \in E : T(sae) = T(s'e)$ . Such an  $s'$  is guaranteed to exist when  $(S, E, T)$   
 9 is closed. The DFSM  $C$  is presented as a conjecture to the Teacher (line 6). If the  
 10 conjecture is correct, *i.e.*, if  $\mathcal{L}(C) = U$ ,  $L^*$  returns  $C$  as correct (line 7), otherwise  
 11 it receives a counterexample  $c \in \Sigma^*$  from the Teacher.

12 The counterexample  $c$  is analyzed using a process described below to find a suffix  
 13  $e$  of  $c$  that witnesses a difference between  $\mathcal{L}(C)$  and  $U$  (line 8).  $e$  must be such that  
 14 adding it to  $E$  will cause the next conjectured automaton to reflect this difference.  
 15 Once  $e$  has been added to  $E$ ,  $L^*$  iterates the entire process by looping around to  
 16 line 2.

17 As stated previously, in line 8  $L^*$  must analyze the counterexample  $c$  to find a suffix  
 18  $e$  of  $c$  that witnesses a difference between  $\mathcal{L}(C)$  and  $U$ . This is done by finding the  
 19 earliest point in  $c$  at which the conjectured automaton and the automaton that would  
 20 recognize the language  $U$  diverge in behavior. This point is found by determining  
 21 where  $\zeta_i \neq \zeta_{i+1}$ , where  $\zeta_i$  is computed as follows:

- 22 (1) Let  $p$  be the sequence of actions made up of the first  $i$  actions in  $c$ . Let  $r$  be  
 23 the sequence made up of the actions after the first  $i$  actions in  $c$ . Thus,  $c = pr$ .
- 24 (2) Run  $C$  on  $p$ . This moves  $C$  into some state  $q$ . By construction, this state  $q$   
 25 corresponds to a row  $s \in S$  of the observation table.
- 26 (3) Perform a query on the actions sequence  $sr$ .
- 27 (4) Return the result of the membership query as  $\zeta_i$ .

28 By using binary search, the point where  $\zeta_i \neq \zeta_{i+1}$  can be found in  $\mathcal{O}(\log |c|)$   
 29 queries, where  $|c|$  is the length of  $c$ .

## 1 2.4.2 Characteristics of $L^*$

2  $L^*$  is guaranteed to terminate with a minimal automaton  $M$  for the unknown lan-  
3 guage  $U$ . Moreover, for each closed observation table  $(S, E, T)$ , the candidate  
4 DFSM  $C$  that  $L^*$  constructs is smallest, in the sense that any other DFSM con-  
5 sistent<sup>1</sup> with the function  $T$  has at least as many states as  $C$ . This characteristic  
6 of  $L^*$  makes it particularly attractive for our framework. The conjectures made by  
7  $L^*$  strictly increase in size; each conjecture is smaller than the next one, and all  
8 incorrect conjectures are smaller than  $M$ . Therefore, if  $M$  has  $n$  states,  $L^*$  makes  
9 at most  $n - 1$  incorrect conjectures. The number of membership queries made by  
10  $L^*$  is  $\mathcal{O}(kn^2 + n \log m)$ , where  $k$  is the size of the alphabet of  $U$ ,  $n$  is the number  
11 of states in the minimal DFSM for  $U$ , and  $m$  is the length of the longest counterex-  
12 ample returned when a conjecture is made.

## 13 3 Learning for Assume-Guarantee Reasoning

14 In this section we introduce a simple, asymmetric assume-guarantee rule and we  
15 describe a framework which uses  $L^*$  to learn assumptions that automate reason-  
16 ing about two components based on this rule. We also discuss how the framework  
17 has been extended to reason about  $n$  components and to use circular rules. **[JMC:**  
18 **Should this section focus on just rule ASYM? It goes into a lot of detail about**  
19 **this rule and learning for this rule, including a long example. I think the struc-**  
20 **ture of the paper would be better if the extensions to ASYM rules were in their**  
21 **own Section.]**

### 22 3.1 Assume-Guarantee Rule ASYM

23 As mentioned, our framework incorporates a number of symmetric and asymmetric  
24 rules for assume-guarantee reasoning. The simplest assume-guarantee proof is for  
25 checking a property  $P$  on a system with two components  $M_1$  and  $M_2$  and is as  
26 follows [18]:

#### Rule ASYM

$$\frac{1 : \langle A \rangle M_1 \langle P \rangle \quad 2 : \langle true \rangle M_2 \langle A \rangle}{\langle true \rangle M_1 \parallel M_2 \langle P \rangle}$$

<sup>1</sup> A DFSM  $C$  is consistent with function  $T$  if, for every  $\sigma$  in  $(S \cup S \cdot \Sigma) \cdot E$ ,  $\sigma \in \mathcal{L}(C)$  if and only if  $T(\sigma) = \text{true}$ .

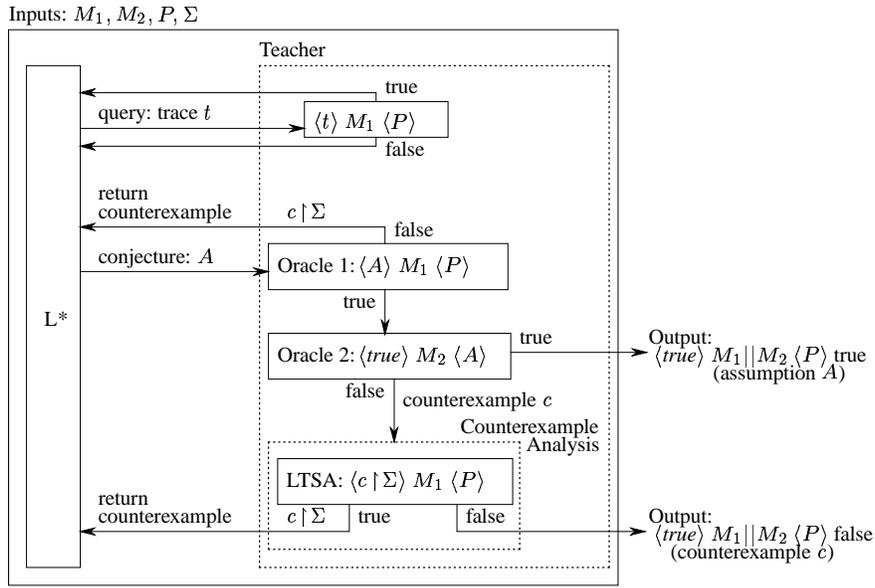


Fig. 4. Learning framework for rule ASYM

- 1 In this rule,  $A$  denotes an assumption about the environment in which  $M_1$  is placed.
- 2 Note that this rule is not symmetric in its use of the two components, and does not
- 3 support circularity. Despite its simplicity, our experience with applying compo-
- 4 sitional verification to several applications has shown it to be most useful in the
- 5 context of checking safety properties.
- 6 For the use of the rule ASYM to be justifi ed, the assumption must be more abstract
- 7 than  $M_2$ , but still reflect  $M_2$ 's behavior. Additionally, an appropriate assumption
- 8 for the rule needs to be strong enough for  $M_1$  to satisfy  $P$  in premise 1. Developing
- 9 such an assumption is diffi cult to do manually. In the following, we describe a
- 10 framework that uses  $L^*$  to learn assumptions automatically.

### 11 3.2 Learning Framework for Rule ASYM

12 To use  $L^*$  to learn assumptions, it needs to be supplied with a Teacher capable of  
 13 answering queries and conjectures. We use the LTSA model checker to answer both  
 14 of these questions. The learning framework for rule ASYM is shown in Fig. 4. For  
 15 this framework, the alphabet of the learned assumption is  $\Sigma = \Sigma_I$ . As a result, the  
 16 sequence of automata conjectured by  $L^*$  converges on the weakest assumption,  $A_w$ .

#### 17 3.2.1 The Teacher

18 **Answering Queries** Recall that  $L^*$  makes a query by asking whether a trace  $t$  is  
 19 in the language being learned. The teacher must return true if  $t$  is in the language  
 20 being learned and false otherwise. To answer a query, the Teacher uses LTSA to

1 check  $\langle t \rangle M_1 \langle P \rangle$ . If this is false, then the assumption needed to make  $\langle A \rangle M_1 \langle P \rangle$   
 2 true should not allow this behavior, and false will be return to  $L^*$ . Otherwise, the  
 3 behavior is allowed and true will be returned to  $L^*$ .

4 **Answering Conjectures** A conjecture consists of an FSM that  $L^*$  believes will  
 5 recognize the language being learned. The Teacher must return true if the conjec-  
 6 ture is correct. Otherwise, the teacher must return false and a counterexample that  
 7 witnesses an error in the conjectured FSM, *i.e.*, a trace in the symmetric difference  
 8 of the language being learned and the conjectured automaton. In our framework,  
 9 the conjectured FSM is an assumption that is being used to complete an assume-  
 10 guarantee proof. Since LTSA takes LTSs and not FSMs, we treat the conjectured  
 11 FSM as an LTS, as described in Section 2.2, which we denote as the LTS  $A$ . To  
 12 answer a conjecture, the Teacher uses two oracles:

- 13 • *Oracle 1* guides  $L^*$  towards a conjecture that makes premise 1 of rule ASYM  
 14 true. It checks  $\langle A \rangle M_1 \langle P \rangle$  and if the result is false, then a counterexample  $c$  is  
 15 produced. Since the premise was false, the conjectured assumption is incorrect  
 16 because it is allowing a behavior (represented by  $c$ ) that should be forbidden.  
 17  $c \upharpoonright \Sigma$  is returned to  $L^*$  to answer the conjecture. If the triple is true, then the  
 18 Teacher moves on to Oracle 2.
- 19 • *Oracle 2* is invoked to check premise 2 of rule ASYM, *i.e.*, to discharge  $A$  on  $M_2$   
 20 by verifying that  $\langle true \rangle M_2 \langle A \rangle$  is true. This triple is checked and if it is true,  
 21 then the assumption makes both premises true and thus, the assume-guarantee  
 22 rule guarantees that  $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$  is true. The Teacher then returns true  
 23 and the computed assumption  $A$ . Note that  $A$  is not necessarily  $A_w$ , it can be  
 24 *stronger* than  $A_w$ , *i.e.*,  $\mathcal{L}(A) \subseteq \mathcal{L}(A_w)$ , but the computed assumption is suffi-  
 25 cient to prove that the property holds. If the triple is not true, then a counterex-  
 26 ample  $c$  is produced. In this case further analysis is needed to determine if either  
 27  $P$  is indeed violated by  $M_1 \parallel M_2$  or if  $A$  is not precise enough, in which case  $A$   
 28 needs to be modified.

29 **Counterexample analysis** The counterexample  $c$  must be analyzed to determine  
 30 if it is a real counterexample, *i.e.*, if it causes  $M_1 \parallel M_2$  to violate  $P$ . To do this,  
 31 the Teacher performs a query on  $c \upharpoonright \Sigma$ , in other words it uses LTSA to check  
 32  $\langle c \upharpoonright \Sigma \rangle M_1 \langle P \rangle$ . If this is false, then  $c$  is a behavior that occurs in  $M_2$  that will result  
 33 in a violation of  $P$  when  $M_2$  interact with  $M_1$ . Thus,  $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$  is false  
 34 and  $c$  is returned as a witness to this. If the triple  $\langle c \upharpoonright \Sigma \rangle M_1 \langle P \rangle$  is true, then  $c$  is a  
 35 behavior that occurs in  $M_2$  that will not result in a violation of  $P$  when  $M_2$  interact  
 36 with  $M_1$ , so the assumption  $A$  is restricting the behavior of  $M_2$  unnecessarily. Thus,  
 37  $c \upharpoonright \Sigma$  is returned to  $L^*$  to answer its conjecture.

Table 1  
Mapping  $T_1$

		$E_1$
		$\lambda$
$S_1$	$T_1$	
	$\lambda$	true
	output	false
$S_1 \cdot \Sigma$	ack	true
	output	false
	send	true
	output, ack	false
	output, output	false
	output, send	false

Table 2  
Mapping  $T_2$

		$E_2$	
		$\lambda$	ack
$S_2$	$T_2$		
	$\lambda$	true	true
	output	false	false
	send	true	false
$S_2 \cdot \Sigma$	ack	true	true
	output	false	false
	send	true	false
	output, ack	false	false
	output, output	false	false
	output, send	false	false
	send, ack	false	false
	send, output	true	true
	send, send	true	true

1 **Remarks** A characteristic of  $L^*$  that makes it particularly attractive for our  
2 framework is its monotonicity. This means that the intermediate candidate assump-  
3 tions that are generated increase in size; each assumption is smaller than the next  
4 one. We should note, however, that there is no monotonicity at the semantic level.  
5 So while  $|A_i| < |A_{i+1}|$ , it is not necessarily the case that  $\mathcal{L}(A_i) \subset \mathcal{L}(A_{i+1})$ .

### 6 3.2.2 Example

7 Given components *Input* and *Output* as shown in Fig. 1 and the property *Order*  
8 shown in Fig. 2 (see Section 2), we will check  $\langle true \rangle Input \parallel Output \langle Order \rangle$   
9 using rule ASYM. To do this, we set  $M_1 = Input$ ,  $M_2 = Output$ ,  
10 and  $P = Order$ . The alphabet of the interface for this example is  
11  $\Sigma = ((\alpha Input \cup \alpha Order) \cap \alpha Output) = \{\text{send, output, ack}\}$ .

12 As described, at each iteration  $L^*$  updates its observation table and produces a  
13 candidate assumption whenever the table becomes closed. The first closed ta-  
14 ble obtained is shown in Table 1 and its associated assumption,  $A_1$ , is shown in  
15 Fig. 5. The Teacher answers conjecture  $A_1$  by first invoking Oracle 1, which checks  
16  $\langle A_1 \rangle Input \langle Order \rangle$ . Oracle 1 returns false, with counterexample  $c = \langle \text{input, send,}$   
17  $\text{ack, input} \rangle$ , which describes a trace in  $A_1 \parallel Input \parallel Order_{err}$  that leads to state  $\pi$ .

18 The Teacher therefore returns counterexample  $c \upharpoonright \Sigma = \langle \text{send, ack} \rangle$  to  $L^*$ , which  
19 uses queries to again update its observation table until it is closed. From this ta-  
20 ble, shown in Table 2, the assumption  $A_2$ , shown in Fig. 6, is constructed and  
21 conjectured to the Teacher. This time, Oracle 1 reports that  $\langle A_2 \rangle Input \langle Order \rangle$   
22 is true, meaning the assumption is not too weak. The Teacher then calls Oracle 2 to



Fig. 5.  $A_1$

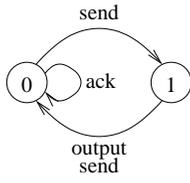


Fig. 6.  $A_2$

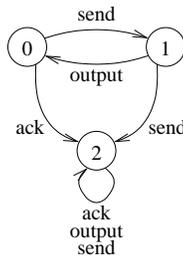


Fig. 7.  $A_3$

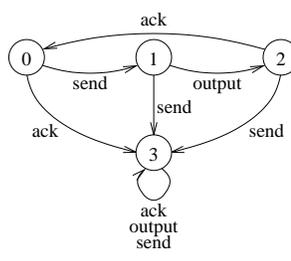


Fig. 8.  $A_4$

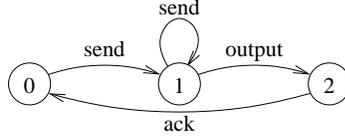


Fig. 9. LTS for  $Output'$

1 determine if  $\langle true \rangle Output \langle A_2 \rangle$ . This is also true, so the framework reports that  
 2  $\langle true \rangle Input \parallel Output \langle Order \rangle$  is true.

3 This example did not involve weakening of the assumptions produced by  $L^*$ , since  
 4 the assumption  $A_2$  was sufficient for the compositional proof. This will not always  
 5 be the case. Consider  $Output'$ , shown in Fig. 9, which allows multiple send actions  
 6 to occur before producing output. If  $Output$  were replaced by  $Output'$ , then the  
 7 verification process would be identical to the previous case, until Oracle 2 is in-  
 8 voked by the Teacher for conjecture  $A_2$ . Oracle 2 returns that  $\langle true \rangle Output' \langle A_2 \rangle$   
 9 is false, with counterexample  $\langle send, send, output \rangle$ . The Teacher analyzes this coun-  
 10 terexample and determines that in the context of this trace,  $Input$  does not violate  
 11  $Order$ . This trace (projected onto  $\Sigma$ ) is returned to  $L^*$ , which will weaken the  
 12 conjectured assumption. The process involves two more iterations, during which  
 13 assumptions  $A_3$  (Fig. 7) and  $A_4$  (Fig. 8), are produced. Using  $A_4$ , which is the  
 14 weakest assumption  $A_w$ , both Oracles report true, so it can be concluded that  
 15  $\langle true \rangle Input \parallel Output' \langle Order \rangle$  also holds.

### 16 3.2.3 Correctness and Termination

17 **Theorem 5** Given components  $M_1$  and  $M_2$ , and property  $P$ , the algorithm im-  
 18 plemented by our framework for rule ASYM terminates and correctly reports on  
 19 whether  $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$  holds or not.

20 **PROOF.** To prove the theorem we will first argue correctness of our approach, and  
 21 then the fact that it terminates.

22 **Correctness:** The Teacher in our framework uses the two premises of the assume-  
 23 guarantee rule to answer conjectures. It only reports that  $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$  is

1 true when both premises are true, and therefore correctness is guaranteed by the  
 2 compositional rule. Our framework reports an error when it detects a trace  $c$  of  $M_2$   
 3 which, when simulated on  $M_1$ , violates the property, which implies that  $M_1 \parallel M_2$   
 4 violates  $P$ .

5 Termination: At any iteration [**JMC: After an assumption is conjectured (?),**  
 6 **our algorithm reports on whether  $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$  is true]** and terminates,  
 7 or continues by providing a counterexample to  $L^*$ . By correctness of  $L^*$ , we are  
 8 guaranteed that if it keeps receiving counterexamples, it will eventually, at some  
 9 iteration  $i$ , produce  $A_w$ . During this iteration, Step 1 will return true by defi nition  
 10 of  $A_w$ . The Teacher will therefore apply Step 2, which will return either true and  
 11 terminate, or a counterexample. This counterexample represents a trace of  $M_2$  that  
 12 is not contained in  $\mathcal{L}(A_w)$ . Since, as discussed before,  $A_w$  is both necessary and  
 13 suffi cient, analysis of the counterexample will return false, and the algorithm will  
 14 terminate.  $\square$

15 It is interesting to note that the algorithm implemented by our framework may ter-  
 16 minate before the weakest assumption is constructed via the iterative learning and  
 17 refi nement process. It terminates as soon as an assumption has been constructed  
 18 that is strong enough to discharge the fi rst premise but weak enough for the second  
 19 premise to produce conclusive results, *i.e.*, to prove the property or produce a real  
 20 counterexample; this assumption may have fewer states than the weakest assump-  
 21 tion.

### 22 3.3 Generalization to $n$ Components

23 We presented our approach so far to the case of two components. Assume now that  
 24 a system consists of  $n$  components. To check if system  $M_1 \parallel M_2 \parallel \dots \parallel M_n$  sat-  
 25 isfi es  $P$ , we decompose it into:  $M_1$  and  $M'_2 = M_2 \parallel M_3 \parallel \dots \parallel M_n$  and the learn-  
 26 ing framework is applied recursively to check the second premise of the assume-  
 27 guarantee rule.

28 At each recursive invocation for  $M_j$  and  $M'_j = M_{j+1} \parallel M_{j+2} \parallel \dots \parallel M_n$ , we solve  
 29 the following problem: fi nd assumption  $A_j$  such that the following are both true:

- 30 •  $\langle A_j \rangle M_j \langle A_{j-1} \rangle$  and
- 31 •  $\langle true \rangle M_{j+1} \parallel M_{j+2} \parallel \dots \parallel M_n \langle A_j \rangle$ .

32 Here  $A_{j-1}$  is the assumption for  $M_{j-1}$  and plays the role of the property for the  
 33 current recursive call. Correctness and termination for this extension follows by  
 34 induction on  $n$  from Theorem 5.

### 1 3.4 Extension with a Circular Rule

2 Our framework can accommodate a variety of assume-guarantee rules that are  
 3 sound. Completeness of rules is required to guarantee termination. We investigate  
 4 here another rule, that is similar to ASYM but it involves circular reasoning. This  
 5 rule appeared originally in [18] (for reasoning about two components). The rule  
 6 can be extended easily to reasoning about  $n \geq 2$  components. [JMC: Should we  
 7 reference Maier’s paper “Compositional Circular Assume-Guarantee Rules  
 8 Cannot Be Sound and Complete” here? Corina: i think we should ref it in  
 9 related work]

#### Rule CIRC-N

$$\begin{array}{l}
 1 : \quad \langle A_1 \rangle M_1 \langle P \rangle \\
 2 : \quad \langle A_2 \rangle M_2 \langle A_1 \rangle \\
 \vdots \\
 n : \quad \langle A_n \rangle M_n \langle A_{n-1} \rangle \\
 n + 1 : \langle true \rangle M_1 \langle A_n \rangle \\
 \hline
 \langle true \rangle M_1 \parallel M_2 \parallel \dots \parallel M_n \langle P \rangle
 \end{array}$$

10 Note that this rule is similar to the rule ASYM applied recursively for  $n + 1$  com-  
 11 ponents, where the first and the last component coincide. Therefore, learning based  
 12 assume-guarantee reasoning proceeds as described in Section 3.3.

## 13 4 Learning with Symmetric Rules

14 Although sound and complete, the rules presented in the previous section are not  
 15 always satisfactory since they are not symmetric in the use of the components.  
 16 In [5] we proposed a set of symmetric rules that are sound and complete and we  
 17 also described their automation using learning. They are symmetric in the sense that  
 18 they are based on establishing and discharging assumptions for each component at  
 19 the same time. Here we present one of the rules that we found particularly effective  
 20 in practice and describe its integration in the learning framework.

### 21 4.1 Symmetric Assume Guarantee Rules

22 Here is an example of a symmetric rule that can be used for reasoning about a sys-  
 23 tem composed of  $n \geq 2$  components:  $M_1 \parallel M_2 \parallel \dots \parallel M_n$ . We require  $\alpha P \subseteq$   
 24  $\alpha M_1 \cup \alpha M_2 \cup \dots \cup \alpha M_n$  and that for  $i \in \{1, 2, \dots, n\}$   $\alpha A_i \subseteq (\alpha M_1 \cap \alpha M_2 \cap \dots \cap \alpha M_n) \cup \alpha P$ .  
 25 Informally, each assumption  $A_i$  is a postulated environment assumptions for the

- 1 component  $M_i$  to achieve to satisfy property  $P$ . The co-assumption for  $M_i$  is de-  
 2 noted  $coA_i$  and is the complement of  $A_i$ .

**Rule SYM-N**

$$\begin{array}{l}
 1 : \quad \langle A_1 \rangle M_1 \langle P \rangle \\
 2 : \quad \langle A_2 \rangle M_2 \langle P \rangle \\
 \vdots \\
 n : \quad \langle A_n \rangle M_n \langle P \rangle \\
 n + 1 : \quad \mathcal{L}(coA_1 \parallel coA_2 \parallel \cdots \parallel coA_n) \subseteq \mathcal{L}(P) \\
 \hline
 \langle true \rangle M_1 \parallel M_2 \parallel \cdots \parallel M_n \langle P \rangle
 \end{array}$$

- 3 **Theorem 6** *Rule SYM-N is sound and complete.*

4 **PROOF.** To establish soundness, we show that the premises together with the  
 5 negated conclusion lead to a contradiction. Consider a trace  $t$  for which the conclu-  
 6 sion fails, *i.e.*,  $t$  is a trace of  $M_1 \parallel M_2 \parallel \cdots \parallel M_n$  that violates property  $P$ , in other  
 7 words  $t$  is not accepted by  $P$ . By the definition of parallel composition,  $t \upharpoonright \alpha M_1$  is  
 8 accepted by  $M_1$ . Hence, by premise 1, the trace  $t \upharpoonright \alpha A_1$  can not be accepted by  $A_1$ ,  
 9 *i.e.*,  $t \upharpoonright \alpha A_1$  is accepted by  $coA_1$ . Similarly, by premise  $i = 2 \dots n$ , the trace  $t \upharpoonright \alpha A_i$   
 10 is accepted by  $coA_i$ . By the definition of parallel composition and the fact that an  
 11 FSM and its complement have the same alphabet,  $t \upharpoonright (\alpha A_1 \cup \alpha A_2 \cup \cdots \cup \alpha A_n)$  is ac-  
 12 cepted by  $coA_1 \parallel coA_2 \parallel \cdots \parallel coA_n$  and it violates  $P$ . But premise  $n + 1$  states  
 13 that the common traces in the co-sets belong to the language of  $P$ . Hence we have  
 14 a contradiction.

15 Our argument for the completeness of Rule SYM-N relies on weakest assumptions.  
 16 To establish completeness, we assume the conclusion of the rule and show that we  
 17 can construct assumptions that will satisfy the premises of the rule. We construct  
 18 the weakest assumptions  $A_{w1}, A_{w2}, \dots, A_{wn}$  for  $M_1, M_2, \dots, M_n$ , respectively, to  
 19 achieve  $P$  and substitute them for  $A_1, A_2, \dots, A_n$ . Premises 1 through  $n$  are sat-  
 20 isfied. It remains to show that premise  $n + 1$  holds. Again we proceed toward a  
 21 contradiction. Suppose there is a trace  $t$  in  $\mathcal{L}(coA_{w1} \parallel coA_{w2} \parallel \cdots \parallel coA_{wn})$  that  
 22 violates  $P$ ; more precisely  $t \upharpoonright \alpha P$  violates property  $P$ . By definition of parallel com-  
 23 position,  $t$  is accepted by all  $coA_{w1}, coA_{w2}, \dots, coA_{wn}$ . Furthermore, there will  
 24 exist  $t_1 \in \mathcal{L}(M_1 \parallel coP)$  such that  $t_1 \upharpoonright \alpha t = t$ , where  $\alpha t$  is the alphabet of the as-  
 25 sumptions. Similarly for  $i = 2 \dots n$ ,  $t_i \in \mathcal{L}(M_i \parallel coP)$ .  $t_1, t_2, \dots, t_n$  can then be  
 26 combined into a trace  $t'$  of  $M_1 \parallel M_2 \parallel \cdots \parallel M_n$  such that  $t' \upharpoonright \alpha t = t$ . But if that is  
 27 so, this contradicts the assumed conclusion that  $M_1 \parallel M_2 \parallel \cdots \parallel M_n$  satisfies  $P$ ,  
 28 since  $t$  violates  $P$ . Therefore, there can not be such a common trace  $t$ , and premise  
 29  $n + 1$  holds.  $\square$

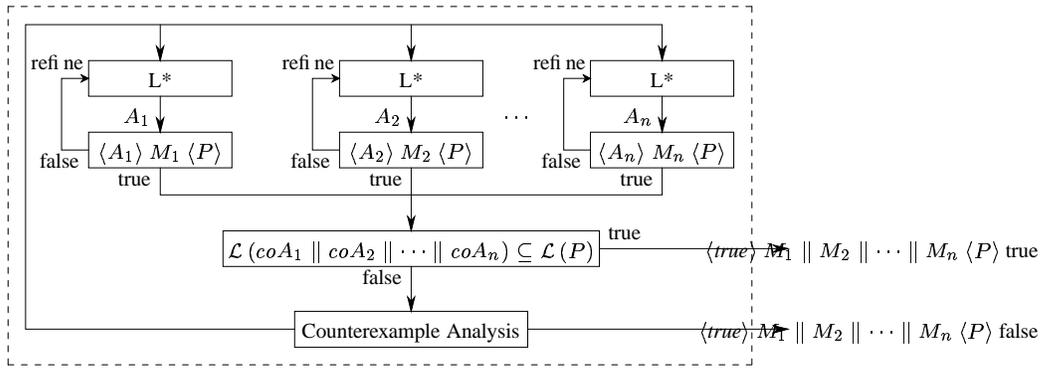


Fig. 10. Learning framework for rule SYM-N to be fixed to have triples in conclusion

## 4.2 Learning Framework for Rule SYM-N

Since rule SYM is just a special case of rule SYM-N, we present here directly the framework for rule SYM-N, as illustrated in Fig. 10. To obtain appropriate assumptions, the framework applies the compositional rule in an iterative fashion.  $L^*$  is used to generate incrementally an assumption for each component, each of which being strong enough to establish the property  $P$ , *i.e.*, to discharge premises 1 through  $n$  of Rule SYM-N. We use separate instances of the  $L^*$  algorithm to iteratively learn the traces of  $A_{w1}, A_{w2}, \dots, A_{wn}$ .

### 4.2.1 The Teacher

As before, we use model checking to implement the Teacher needed by  $L^*$ . Approximate assumptions are built by *querying* the system and using the results of the previous iteration.

The conjectures returned by  $L^*$  are intermediate assumptions  $A_1, A_2, \dots, A_n$ . The Teacher implements  $n + 1$  oracles, one for each premise in the assume-guarantee rule:

- *Oracles* 1, 2,  $\dots$ ,  $n$  guide the corresponding  $L^*$  instances towards conjectures that make the corresponding premise of rule SYM-N true. Once this is accomplished,
- *Oracle*  $n + 1$  is invoked to check the last premise of the rule, *i.e.*,

$$\mathcal{L}(coA_1 \parallel coA_2 \parallel \dots \parallel coA_n) \subseteq \mathcal{L}(P)$$

If this is true, rule SYM-N guarantees that  $M_1 \parallel M_2 \parallel \dots \parallel M_n$  satisfies  $P$ .

If the result of *Oracle*  $n + 1$  is false (with counterexample trace  $t$ ), by counterexample analysis we identify whether  $P$  is indeed violated in  $M_1 \parallel M_2 \parallel \dots \parallel M_n$  or some of the candidate assumptions need to be modified. If an assumption needs

1 to be refined then behaviors must be added, in the next iteration. The result will be  
 2 that at least the behavior that the counterexample represents will be allowed by that  
 3 assumption at the next iteration. The new assumption may of course be too abstract,  
 4 and therefore the entire process must be repeated.

5 **Counterexample analysis** Counterexample  $t$  is analyzed in a way similar to the  
 6 analysis for rule ASYM, *i.e.*, we analyze  $t$  to determine whether it indeed corre-  
 7 sponds to a violation in  $M_1 \parallel M_2 \parallel \dots \parallel M_n$ . This is checked by simulating  $t$  on  
 8  $M_i \parallel coP$ , for all  $i = 1 \dots n$ . The following cases arise:

- 9 • If  $t$  is a violating trace of all components  $M_1, M_2, \dots, M_n$ , then  
 10  $M_1 \parallel M_2 \parallel \dots \parallel M_n$  indeed violates  $P$ , which is reported to the user.
- 11 • If  $t$  is not a violating trace of at least one component  $M_i$ , then we use  $t$  to weaken  
 12 the corresponding assumption(s).

#### 13 4.2.2 Correctness and Termination

14 **Theorem 7** *Given components  $M_1, M_2, \dots, M_n$  and property  $P$ , the algorithm im-*  
 15 *plemented by our framework for rule SYM-N terminates and correctly reports on*  
 16 *whether  $P$  holds on  $M_1 \parallel M_2 \parallel \dots \parallel M_n$ .*

17 **PROOF.** Correctness: The Teacher returns true only if the premises of rule SYM-N  
 18 hold, and therefore correctness is guaranteed by the soundness of the rule. The  
 19 Teacher reports a counterexample only when it finds a trace that is violating in all  
 20 components, which implies that  $M_1 \parallel M_2 \parallel \dots \parallel M_n$  also violates  $P$ .

21 Termination: At any iteration, the teacher reports on whether or not  $P$  holds on  
 22  $M_1 \parallel M_2 \parallel \dots \parallel M_n$  and terminates, or continues by providing a counterexam-  
 23 ple to  $L^*$ . By the correctness of  $L^*$ , we are guaranteed that if it keeps receiving  
 24 counterexamples, it will eventually, produce  $A_{w1}, A_{w2}, \dots, A_{wn}$  respectively.

25 During this last iteration, premises 1 through  $n$  will hold by definition of the weak-  
 26 est assumptions. The Teacher will therefore check premise  $n + 1$ , which will return  
 27 either true and terminate, or a counterexample. Since the weakest assumptions are  
 28 used, by the completeness of the rule, we know that the counterexample analysis  
 29 will reveal a true error, and hence the process will terminate.  $\square$

## 30 5 Learning with Alphabet Refinement

31 In this section, we present a technique that extends the learning based assume-  
 32 guarantee reasoning framework with alphabet refinement. We first illustrate the

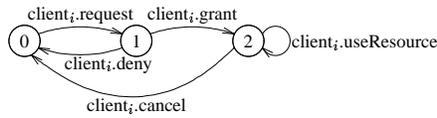


Fig. 11. Example LTS for a client

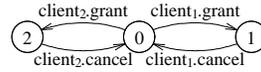


Fig. 12. Mutual exclusion property

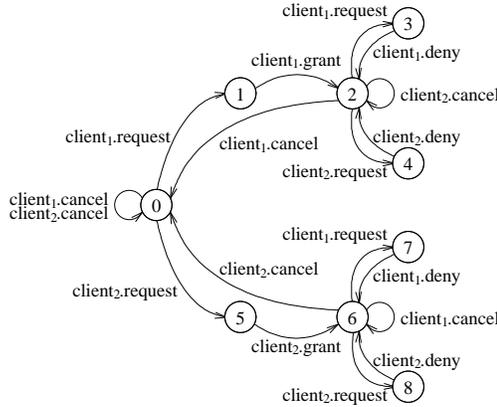


Fig. 13. Example LTS for a server

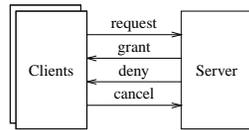


Fig. 14. Complete interface for the client-server example

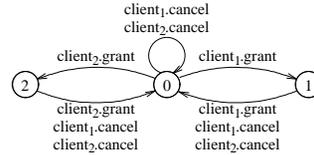


Fig. 15. Assumption learned with an alphabet smaller than the complete interface alphabet

1 benefits of smaller interface alphabets for assume-guarantee reasoning through a  
 2 simple client-server example from [24]. Then, we explain the effect of smaller in-  
 3 terface alphabets on learning assumptions. We then describe the alphabet refi-  
 4 nement algorithm, give its properties, and discuss how it extends to reasoning about  
 5  $n$  components as well as to circular and symmetric rules.

### 6 5.1 Example

7 Consider a system consisting of a *server* component and two identical *client* com-  
 8 ponents that communicate through shared actions. Each client sends *requests* for  
 9 reservations to use a common resource, waits for the server to *grant* the reserva-  
 10 tion, uses the resource, and then *cancel*s the reservation. For example, the LTS of a  
 11 client is shown in Fig. 11, where  $i = 1, 2$ . The server, shown in Fig. 13 can *grant* or  
 12 *deny* a request, ensuring that the resource is used only by one client at a time. We

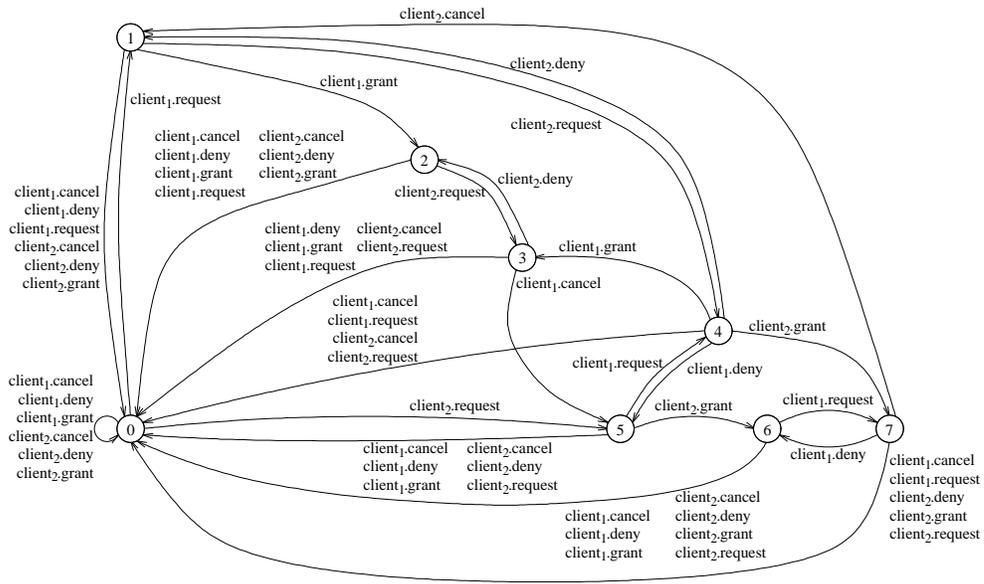


Fig. 16. Client-Server Example: assumption obtained with the complete interface alphabet

1 are interested in checking the mutual exclusion property illustrated in Fig. 12, that  
 2 captures the desired behavior of the client-server application.

3 To check the property in a compositional way, assume that we break up the  
 4 system into:  $M_1 = Client_1 \parallel Client_2$  and  $M_2 = Server$ . The *complete* alpha-  
 5 bet of the interface between  $M_1 \parallel P$  and  $M_2$  (see Fig. 14) is:  $\{client_1.cancel,$   
 6  $client_1.grant, client_1.deny, client_1.request, client_2.cancel, client_2.grant, client_2.deny,$   
 7  $client_2.request\}$ .

8 Using this alphabet and the learning method of [13], an assumption with eight  
 9 states is learned, shown in Fig. 16. However, a (much) smaller assumption is suf-  
 10 ficient for proving the mutual exclusion property. With an assumption alphabet  
 11 of  $\{client_1.cancel, client_1.grant, client_2.cancel, client_2.grant\}$ , a strict subset of the  
 12 complete interface alphabet (and, in fact, the alphabet of the property), a three state  
 13 assumption, shown in Fig. 15, is learned. This smaller assumption enables more  
 14 efficient verification than the eight state assumption obtained with the complete al-  
 15 phabet. In the following section, we present an extension of the learning framework  
 16 that is able to automatically infer smaller interface alphabets (and the correspond-  
 17 ing assumptions).

## 18 5.2 Learning Based Assume-guarantee Reasoning and Small Interface Alphabets

19 Before describing the alphabet refinement algorithm, let us first consider the effect  
 20 of smaller interface alphabets in the context of the learning framework. Let  $M_1$  and  
 21  $M_2$  be components,  $P$  be a property,  $\Sigma_I$  be the interface alphabet, and  $\Sigma$  be an al-  
 22 phabet such that  $\Sigma \subset \Sigma_I$ . Assume that we use the learning framework of Section 3

1 but we now set this smaller  $\Sigma$  to be the alphabet of the assumption that the frame-  
 2 work learns. From the correctness of the assume-guarantee rule, if the framework  
 3 reports true,  $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$ . When it reports false, it is because it fi nds a trace  
 4  $t$  in  $M_2$  that falsifi es  $\langle t \upharpoonright \Sigma \rangle M_1 \langle P \rangle$ . This, however, does not necessarily mean that  
 5  $M_1 \parallel M_2$  violates  $P$ . Real violations are discovered by our original framework only  
 6 when the alphabet is  $\Sigma_I$ , and are traces  $t'$  of  $M_2$  that falsify  $\langle t' \upharpoonright \Sigma_I \rangle M_1 \langle P \rangle$ .<sup>2</sup>

7 Consider again the client-server example. Assume  $\Sigma = \{\text{client}_1.\text{cancel}, \text{client}_1.\text{grant},$   
 8  $\text{client}_2.\text{grant}\}$ , which is smaller than  $\Sigma_I = \{\text{client}_1.\text{cancel}, \text{client}_1.\text{grant}, \text{client}_1.\text{deny},$   
 9  $\text{client}_1.\text{request}, \text{client}_2.\text{cancel}, \text{client}_2.\text{grant}, \text{client}_2.\text{deny}, \text{client}_2.\text{request}\}$ . Learn-  
 10 ing with  $\Sigma$  produces trace:  $t = \langle \text{client}_2.\text{request}, \text{client}_2.\text{grant}, \text{client}_2.\text{cancel},$   
 11  $\text{client}_1.\text{request}, \text{client}_1.\text{grant} \rangle$ . [JMC: How is this trace produced? Is it a trace  
 12 that “Counterexample Analysis” believes is a real counterexample with the al-  
 13 phabet  $\Sigma$ ? If so, please say this. If not, please say where this counterexample  
 14 comes from.] Projected to  $\Sigma$ , this becomes  $t \upharpoonright \Sigma = \langle \text{client}_2.\text{grant}, \text{client}_1.\text{grant} \rangle$ . In  
 15 the context of  $t \upharpoonright \Sigma$ ,  $M_1$  violates the property since  $Client_1 \parallel Client_2 \parallel P_{err}$   
 16 contains the following behavior (see Fig. 14): [JMC: Why does Figure 14 show this?]  
 17

$$\begin{array}{ccccc}
 (0, 0, 0) & \xrightarrow{\text{client}_1.\text{request}} & (1, 0, 0) & \xrightarrow{\text{client}_2.\text{request}} & (1, 1, 0) \\
 & & \xrightarrow{\text{client}_2.\text{grant}} & (1, 2, 2) & \xrightarrow{\text{client}_1.\text{grant}} & (2, 2, \text{error})
 \end{array}$$

18  
 19 Learning therefore reports *false*. This behavior is not feasible, however, in the  
 20 context of  $t \upharpoonright \Sigma_I = \langle \text{client}_2.\text{request}, \text{client}_2.\text{grant}, \text{client}_2.\text{cancel}, \text{client}_1.\text{request},$   
 21  $\text{client}_1.\text{grant} \rangle$ . This trace requires a  $\text{client}_2.\text{cancel}$  to occur before the  $\text{client}_1.\text{grant}$ .  
 22 Thus, in the context of  $\Sigma_I$  the above violating behavior would be infeasible. We  
 23 conclude that when applying the learning framework with alphabets smaller than  
 24  $\Sigma_I$ , if *true* is reported then the property holds in the system, but violations reported  
 25 may be spurious.

### 26 5.3 Algorithm for Alphabet Refinement

27 *Alphabet refinement* extends the learning framework from [13] to deal with smaller  
 28 alphabets than  $\Sigma_I$  while avoiding spurious counterexamples. The steps of the algo-  
 29 rithm are as follows (see Fig. 17 (a)):

- 30 (1) **Initialize**  $\Sigma$  to a set  $S$  such that  $S \subseteq \Sigma_I$ .
- 31 (2) Use the classic learning framework for  $\Sigma$ . If the framework returns *true*, then  
 32 report *true* and go to step 4 (END). If the framework returns false with coun-  
 33 terexamples  $c$  and  $t$ , go to the next step.

<sup>2</sup> In the assume-guarantee triples:  $t \upharpoonright \Sigma$  and  $t' \upharpoonright \Sigma_I$  are trace LTSs with alphabets  $\Sigma$  and  $\Sigma_I$ , respectively.

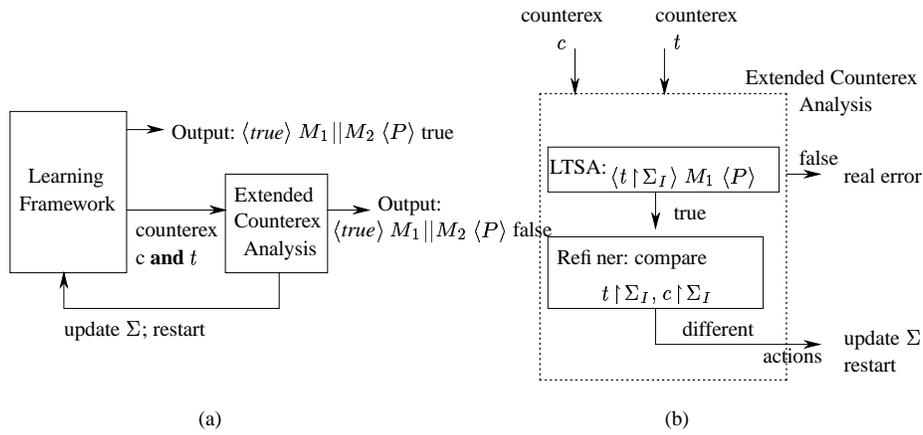


Fig. 17. Learning with alphabet refinement (a) and additional counterexample analysis (b)

[JMC: But Fig. 4 shows the learning algorithm only returning  $c$ . Should we modify Fig. 4 to show it returning  $c$  and  $t$ . Or, should we have it just have it return  $c$  and define  $t$  to be  $c \upharpoonright \Sigma$ . Actually this whole next section is confusing because Fig. 4 shows  $c$  coming out of Oracle 2, which seems to be the opposite of how it is used in this section, i.e., in this section  $t$  is used as the output of Oracle 2. Can someone take a look at the  $c$ 's and  $t$ 's in this section and see if they are used consistently with Fig. 4?]

- (3) Perform **extended counterexample analysis** for  $c$ . If  $c$  is a real counterexample, then report *false* and go to step 4 (END). If  $c$  is spurious, then **refine**  $\Sigma$ , which consists of adding actions to  $\Sigma$  from  $\Sigma_I$ . Go to step 2.
- (4) END of algorithm.

When spurious counterexamples are detected, the refiner extends the alphabet with actions in the alphabet of the weakest assumption [JMC: **maximal interface alphabet (?)**] and the learning of assumptions is restarted. In the worst case,  $\Sigma_I$  is reached, and as proved in our previous work, learning would then only reports real counterexamples. In the above high-level algorithm, the highlighted steps are further specified in the following.

**Alphabet initialization** The correctness of our algorithm is insensitive to the initial alphabet. We set the initial alphabet to those actions in the alphabet of the property that are also in  $\Sigma_I$ , i.e.,  $\alpha P \cap \Sigma_I$ . The intuition is that these interface actions are likely to be significant in proving the property, since they are involved in its definition. A good initial guess of the alphabet may achieve big savings in terms of time since it results in fewer refinement iterations.

**Extended counterexample analysis** An additional counterexample analysis is appended to the original learning framework as illustrated in Fig. 17(a). The steps of this analysis are shown in Fig. 17(b). The extension takes as inputs both the counterexample  $t$  returned by Oracle 2, and the counterexample  $c$  that is returned by the original counterexample analysis. We modified the “classic” learning framework

1 (Fig. 4) to return both  $c$  **and**  $t$  to be used in alphabet refinement (as explained be-  
 2 low). As discussed,  $c$  is obtained because  $\langle t \upharpoonright \Sigma \rangle M_1 \langle P \rangle$  does not hold. The next  
 3 step is to check whether in fact  $t$  uncovers a real violation in the system. As illus-  
 4 trated by the client-server example, the results of checking  $M_1 \parallel P_{err}$  in the context  
 5 of  $t$  projected to different alphabets may be different. The correct (non-spurious)  
 6 results are obtained by projecting  $t$  on the alphabet  $\Sigma_I$  of the weakest assumption.  
 7 Counterexample analysis therefore calls LTSA to check  $\langle t \upharpoonright \Sigma_I \rangle M_1 \langle P \rangle$ . If LTSA  
 8 finds an error, the resulting counterexample  $c$  is real. If error is not reached, then  
 9 the counterexample is spurious and the alphabet  $\Sigma$  needs to be refined. Refinement  
 10 proceeds as described next.

11 **Alphabet refinement** When spurious counterexamples are detected, we need to  
 12 enrich the current alphabet  $\Sigma$  so that these counterexamples are eventually elim-  
 13 inated. A counterexample  $c$  is spurious if in the context of  $t \upharpoonright \Sigma_I$  it would not be  
 14 obtained. Our refinement heuristics are therefore based on comparing  $c$  and  $t \upharpoonright \Sigma_I$   
 15 to discover actions in  $\Sigma_I$  to be added to the learning alphabet (for this reason  $c$  is  
 16 also projected on  $\Sigma_I$  in the refinement process). We have currently implemented  
 17 the following heuristics:

18 **AllDiff:** adds all the actions in the symmetric difference of  $t \upharpoonright \Sigma_I$  and  $c \upharpoonright \Sigma_I$ . A  
 19 potential problem of this heuristic is that it may add too many actions too soon. If  
 20 it happens to add useful actions, however, it may terminate after a small number  
 21 of iterations.

22 **Forward:** scans the traces [**JMC:**  $t \upharpoonright \Sigma_I$  **and**  $c \upharpoonright \Sigma_I$  (?)] in parallel from beginning  
 23 to end looking for the first index  $i$  where they disagree; if such an  $i$  is found,  
 24 both actions  $t \upharpoonright \Sigma_I(i)$ ,  $c \upharpoonright \Sigma_I(i)$  are added to the alphabet. By adding fewer actions  
 25 during each iteration, the algorithm may end up with a smaller alphabet. But, it  
 26 may take more iterations before it does not produce a spurious result.

27 **Backward:** as similar to Forward, but scans from the end of the traces to the be-  
 28 ginning.

### 29 5.3.1 Correctness and Termination

30 For correctness and termination of learning with alphabet refinement, we first show  
 31 progress of refinement, meaning that at each refinement stage, new actions are dis-  
 32 covered to be added to  $\Sigma$ .

33 **Proposition 8 (Progress of alphabet refinement)** *Let  $\Sigma_I = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$   
 34 and  $\Sigma \subset \Sigma_I$  be the alphabet of the weakest assumption and that of the assumption  
 35 at the current alphabet refinement stage, respectively. Let  $t$  be a trace of  $M_2 \parallel A_{err}$   
 36 such that  $t \upharpoonright \Sigma$  leads to error on  $M_1 \parallel P_{err}$  by an error trace  $c$ , but  $t \upharpoonright \Sigma_I$  does not  
 37 lead to error on  $M_1 \parallel P_{err}$ . Then  $t \upharpoonright \Sigma_I \neq c \upharpoonright \Sigma_I$  and there exists an action in their  
 38 symmetric difference that is not in  $\Sigma$ .*

1 **PROOF.** We prove by contradiction that  $t \upharpoonright_{\Sigma_I} \neq c \upharpoonright_{\Sigma_I}$ . Suppose  $t \upharpoonright_{\Sigma_I} = c \upharpoonright_{\Sigma_I}$ .  
2 We know that  $c$  is an error trace on  $M_1 \parallel P_{err}$ . Since actions of  $c$  that are not in  $\Sigma_I$   
3 are internal to  $M_1 \parallel P$ , then  $c \upharpoonright_{\Sigma_I}$  also leads to error on  $M_1 \parallel P_{err}$ . But then  $t \upharpoonright_{\Sigma_I}$   
4 leads to error on  $M_1 \parallel P_{err}$ , which is a contradiction.

5 We now show that there exists an action in the symmetric difference between  $t \upharpoonright_{\Sigma_I}$   
6 and  $c \upharpoonright_{\Sigma_I}$  that is not in  $\Sigma$  (this action will be added to  $\Sigma$  by alphabet refinement).  
7 Trace  $t \upharpoonright_{\Sigma_I}$  is  $t \upharpoonright_{\Sigma}$ , with some interleaved actions from  $\Sigma_I \setminus \Sigma$ . Similarly,  $c \upharpoonright_{\Sigma_I}$   
8 is  $c \upharpoonright_{\Sigma}$  with some interleaved actions from  $\Sigma_I \setminus \Sigma$ , since  $c$  is obtained by compos-  
9 posing the trace LTS  $t \upharpoonright_{\Sigma}$  with  $M_1 \parallel P_{err}$ . Thus  $t \upharpoonright_{\Sigma} = c \upharpoonright_{\Sigma}$ . We again proceed  
10 by contradiction. If all the actions in the symmetric difference between  $t \upharpoonright_{\Sigma_I}$  and  
11  $c \upharpoonright_{\Sigma_I}$  were in  $\Sigma$ , we would have  $t \upharpoonright_{\Sigma_I} = t \upharpoonright_{\Sigma} = c \upharpoonright_{\Sigma} = c \upharpoonright_{\Sigma_I}$ , which contradicts  
12  $t \upharpoonright_{\Sigma_I} \neq c \upharpoonright_{\Sigma_I}$ .  $\square$

13 We also use the following lemma.

14 **Lemma 9** *For any component  $M_1$ , property  $P$ , and interface alphabet  $\Sigma$ ,*  
15  *$\langle A_{w,\Sigma} \rangle M_1 \langle P \rangle$  holds.*

16 **PROOF.**  $A_{w,\Sigma} \upharpoonright_{\Sigma} = A_{w,\Sigma}$ . If in Definition 2 we substitute  $A_{w,\Sigma}$  for  $M_2$ , we obtain  
17 that:  $M_1 \parallel A_{w,\Sigma_1} \models P$  if and only if  $A_{w,\Sigma_1} \models A_{w,\Sigma}$ . But the latter holds trivially, so  
18 we conclude that  $M_1 \parallel A_{w,\Sigma_1} \models P$ , which is equivalent to  $\langle A_{w,\Sigma} \rangle M_1 \langle P \rangle$ , always  
19 holds.  $\square$

20 Correctness follows from the assume guarantee rule and the extended counterex-  
21 ample analysis. Termination follows from termination of the original framework,  
22 from the progress property and also from the finiteness of  $\Sigma_I$ . Moreover, from the  
23 progress property it follows that the refinement algorithm for two components has  
24 at most  $|\Sigma_I|$  iterations.

25 **Theorem 10** *Given components  $M_1$  and  $M_2$ , and property  $P$ ,  $L^*$  with alphabet*  
26 *refinement terminates and returns true if  $M_1 \parallel M_2$  satisfies  $P$  and false otherwise.*

27 **PROOF.** Correctness: When the teacher returns true, then correctness is guaran-  
28 teed by the assume-guarantee compositional rule. If the teacher returns false, the  
29 extended counterexample analysis reports an error for a trace  $t$  of  $M_2$ , such that  
30  $t \upharpoonright_{\Sigma_I}$  in the context of  $M_1$  violates the property (the same test is used in the algo-  
31 rithm from [13]) hence  $M_1 \parallel M_2$  violates the property.

32 Termination: From the correctness of  $L^*$ , we know that at each refinement stage  
33 (with alphabet  $\Sigma$ ), if  $L^*$  keeps receiving counterexamples, it is guaranteed to gen-  
34 erate  $A_{w,\Sigma}$ . At that point, Oracle 1 will return true (from Lemma 9). Therefore,

1 Oracle 2 will be applied, which will return either true, and terminate, or a coun-  
2 terexample  $t$ . This counterexample is a trace that is not in  $\mathcal{L}(A_{w,\Sigma})$ . It is either a  
3 real counter example (in which case the algorithm terminates) or it is a trace  $t$  such  
4 that  $t \upharpoonright \Sigma$  leads to error on  $M_1 \parallel P_{err}$  by an error trace  $c$ , but  $t \upharpoonright \Sigma_I$  does not lead  
5 to error on  $M_1 \parallel P_{err}$ . Then from Theorem 8, we know that  $t \upharpoonright \Sigma_I \neq c \upharpoonright \Sigma_I$  and  
6 there exists an action in their symmetric difference that is not in  $\Sigma$ . The refiner will  
7 add this action (or more actions depending on the refinement strategy) to  $\Sigma$  and  
8 the learning algorithm is repeated for this new alphabet. Since  $\Sigma_I$  is finite, in the  
9 worst case,  $\Sigma$  grows into  $\Sigma_I$ , for which termination and correctness follow from  
10 Theorem ??  $\square$

11 We also note a property of weakest assumptions, which states that by adding ac-  
12 tions to an alphabet  $\Sigma$ , the corresponding weakest assumption becomes *weaker*  
13 (*i.e.*, contains more behaviors) than the previous one.

14 **Proposition 11** Assume components  $M_1$  and  $M_2$ , property  $P$  and the correspond-  
15 ing interface alphabet  $\Sigma_I$ . Let  $\Sigma, \Sigma'$  be sets of actions such that:  $\Sigma \subset \Sigma' \subset \Sigma_I$ .  
16 Then:  $\mathcal{L}(A_{w,\Sigma}) \subseteq \mathcal{L}(A_{w,\Sigma'}) \subseteq \mathcal{L}(A_{w,\Sigma_I})$ .

17 **PROOF.** Since  $\Sigma \subseteq \Sigma'$ , we know that  $A_{w,\Sigma} \upharpoonright \Sigma' = A_{w,\Sigma}$ . By substituting, in  
18 Definition 2,  $A_{w,\Sigma}$  for  $M_2$ , we obtain that:  $\langle true \rangle M_1 \parallel (A_{w,\Sigma}) \langle P \rangle$  if and only if  
19  $\langle true \rangle A_{w,\Sigma} \langle A_{w,\Sigma'} \rangle$ . From Proposition 9 [JMC: Should this be Lemma 9?] we  
20 know that  $\langle true \rangle M_1 \parallel (A_{w,\Sigma}) \langle P \rangle$ . Therefore,  $\langle true \rangle A_{w,\Sigma} \langle A_{w,\Sigma'} \rangle$  holds, which  
21 implies that  $\mathcal{L}(A_{w,\Sigma}) \subseteq \mathcal{L}(A_{w,\Sigma'})$ . Similarly,  $\mathcal{L}(A_{w,\Sigma'}) \subseteq \mathcal{L}(A_{w,\Sigma_I})$ .  $\square$

22 With alphabet refinement, our framework adds actions to the alphabet, which trans-  
23 lates into adding more behaviors to the weakest assumption that  $L^*$  tries to prove.  
24 This means that at each refinement stage  $i$ , when the learner is started with a new  
25 alphabet  $\Sigma_i$  such that  $\Sigma_{i-1} \subset \Sigma_i$ , it will try to learn a weaker assumption  $A_{w,\Sigma_i}$   
26 than  $A_{w,\Sigma_{i-1}}$ , which was its goal in the previous stage. Moreover, all these assump-  
27 tions are *under-approximations* of the weakest assumption  $A_{w,\Sigma_I}$  that is necessary  
28 and sufficient to prove the desired property. Note that at each refinement stage the  
29 learner might stop earlier, *i.e.*, before computing the corresponding weakest as-  
30 sumption. The above property allows re-use of learning results across refinement  
31 stages (see Section 8).

## 32 5.4 Generalization to $n$ Components

33 Alphabet refinement can also be used when reasoning about more than  
34 two components using rule ASYM. Recall from Section 3 that to check  
35 if system  $M_1 \parallel M_2 \parallel \dots \parallel M_n$  satisfies  $P$ , we decompose it into:  $M_1$  and

1  $M'_2 = M_2 \parallel M_3 \parallel \dots \parallel M_n$  and the learning algorithm (without refinement) is in-  
 2 voked recursively for checking the second premise of the assume-guarantee rule.

3 Learning with alphabet refinement follows this recursion. At each recursive invo-  
 4 cation for  $M_j$  and  $M'_j = M_{j+1} \parallel M_{j+2} \dots \parallel M_n$ , we solve the following problem:  
 5 find assumption  $A_j$  and alphabet  $\Sigma_{A_j}$  such that the rule premises hold, *i.e.*

6 Oracle 1:  $\langle true \rangle M_j \parallel A_j \langle A_{j-1} \rangle$  and

7 Oracle 2:  $\langle true \rangle M_{j+1} \parallel M_{j+2} \parallel \dots \parallel M_n \langle A_j \rangle$ .

8 Here  $A_{j-1}$  is the assumption for  $M_{j-1}$  and plays the role of the property for the  
 9 current recursive call. Thus, the alphabet of the weakest assumption for this re-  
 10 cursive invocation is  $\Sigma_I^j = (\alpha M_j \cup \alpha A_{j-1}) \cap (\alpha M_{j+1} \cup \alpha M_{j+2} \cup \dots \cup \alpha M_n)$ . If

11 Oracle 2 returns a counterexample, then the counterexample analysis and alphabet  
 12 refinement proceed exactly as in the 2-component case. Note that at a new recursive  
 13 call for  $M_j$  with a new  $A_{j-1}$ , the alphabet of the weakest assumption is recomputed.

14 Correctness and termination of this extension follow from Theorem 10 (and from  
 15 finiteness of  $n$ ). The proof proceeds by induction on  $n$ .

## 16 5.5 Extension with Circular and Symmetric Rules

17 Alphabet refinement also applies to the rules CIRC-N and SYM-N. As mentioned,  
 18 CIRC-N is a special case of the recursive application of rule ASYM for  $n+1$  compo-  
 19 nents, where the first and last component coincide. Therefore alphabet refinement  
 20 applies to CIRC-N as we described here.

21 For rule SYM-N, the counterexample analysis for the error trace  $t$  obtained from  
 22 checking premise  $n+1$  is extended for each component  $M_i$ , for  $i = 1 \dots n$ . The  
 23 extension works similarly to that for ASYM discussed earlier in this section. The  
 24 error trace  $t$  is simulated on each  $M_i \parallel coP$  with the current assumption alphabet.

- 25 • If  $t$  is violating for some  $i$ , then we check whether  $t$ , with the entire alphabet of  
 26 the weakest assumption for  $i$  is still violating. If it is, then  $t$  is a real error trace  
 27 for  $M_i$ . If it is not, the alphabet of the current assumption for  $i$  is refined with  
 28 actions from the alphabet of the corresponding weakest assumption.
- 29 • If  $t$  is a real error trace for all  $i$ , then it is reported as a real violation of the  
 30 property on the entire system.

31 If alphabet refinement takes place for some  $i$ , the learning of the assumption for  
 32 this  $i$  is restarted with the refined alphabet, and premise  $n+1$  is re-checked with  
 33 the new learned assumption for  $i$ .

1 **needs to be fixed; it is unclear what c is here; see Fig 17**

## 2 **6 Experiments**

3 **this is biased somewhat towards alpha refinement: maybe we should change**  
4 **that tables should be made smaller**

5 We implemented learning with rules ASYM, SYM-N, CIRC-N, with and without al-  
6 phabet refinement in LTSA and evaluated the implementations for checking safety  
7 properties of various concurrent systems that we briefly describe below. The goal  
8 of the evaluation was to assess the performance of learning, the effect of alpha-  
9 bet refinement on learning, to compare the effect of the different rules, and to also  
10 compare the scalability of compositional verification by learning to that of non-  
11 compositional verification.

12 **Models and properties** We used the following case studies. *Gas Station* [12] de-  
13 scribes a self-serve gas station consisting of  $k$  customers, two pumps, and an op-  
14 erator. For  $k = 3, 4, 5$ , we checked that the operator correctly gives change to a  
15 customer for the pump that he/she used. *Chiron* [12] models a graphical user inter-  
16 face consisting of  $k$  artists, a wrapper, a manager, a client initialization module, a  
17 dispatcher, and two event dispatchers. For  $k = 2 \dots 5$ , we checked two properties:  
18 “the dispatcher notifies artists of an event before receiving a next event”, and “the  
19 dispatcher only notifies artists of an event after it receives that event”. *MER* [24]  
20 models the flight software component for JPL’s Mars Exploration Rovers. It con-  
21 tains  $k$  users competing for resources managed by an arbiter. For  $k = 2 \dots 6$ , we  
22 checked that communication and driving cannot happen at the same time as they  
23 share common resources. *Rover Executive* [13] models a subsystem of the Ames  
24 K9 Rover. The models consists of a main ‘Executive’ and an ‘ExecCondChecker’  
25 component responsible for monitoring state conditions. We checked that for a spe-  
26 cific shared variable, if the Executive reads its value, then the ExecCondChecker  
27 should not read it before the Executive clears it.

28 Note that the Gas Station and Chiron were analyzed before, in [12], using learning  
29 based assume guarantee reasoning (with ASYM and no alphabet refinement). Four  
30 properties of Gas Station and nine properties of Chiron were checked to study how  
31 various 2-way model decompositions (i.e. grouping the modules of each analyzed  
32 system into two “super-components”) affect the performance of learning. For most  
33 of these properties, learning performs better than non-compositional verification  
34 and produces small (one-state) assumptions. For some other properties, learning  
35 does not perform that well, and produces much larger assumptions. To stress-test  
36 our approach, we selected the latter, more challenging properties for our study here.

Table 3

Comparison of learning for 2-way decompositions with ASYM, with and without alphabet refinement.

Case	$k$	No refinement			Refinement + bwd			Refinement + fwd			Refinement + allDiff		
		$ A $	Mem.	Time	$ A $	Mem.	Time	$ A $	Mem.	Time	$ A $	Mem.	Time
Gas Station	3	177	4.34	–	8	3.29	2.70	37	6.47	36.52	18	4.58	7.76
	4	195	100.21	–	8	24.06	19.58	37	46.95	256.82	18	36.06	52.72
	5	53	263.38	–	8	248.17	183.70	20	414.19	–	18	360.04	530.71
Chiron, Property 1	2	9	1.30	1.23	8	1.22	3.53	8	1.22	1.86	8	1.22	1.90
	3	21	5.70	5.71	20	6.10	23.82	20	6.06	7.40	20	6.06	7.77
	4	39	27.10	28.00	38	44.20	154.00	38	44.20	33.13	38	44.20	35.32
	5	111	569.24	607.72	110	–	300	110	–	300	110	–	300
Chiron, Property 2	2	9	1.16	1.10	3	1.05	0.73	3	1.05	0.73	3	1.05	0.74
	3	25	4.45	6.39	3	2.20	0.93	3	2.20	0.92	3	2.20	0.92
	4	45	25.49	32.18	3	8.13	1.69	3	8.13	1.67	3	8.13	1.67
	5	122	131.49	246.84	3	163.85	18.08	3	163.85	18.05	3	163.85	17.99
MER	2	40	6.57	7.84	6	1.78	1.01	6	1.78	1.02	6	1.78	1.01
	3	377	158.97	–	8	10.56	11.86	8	10.56	11.86	8	10.56	11.85
	4	38	391.24	–	10	514.41	1193.53	10	514.41	1225.95	10	514.41	1226.80
Rover Exec.	2	11	2.65	1.82	4	2.37	2.53	11	2.67	4.17	11	2.54	2.88

1 **Experimental set-up and results** We performed several sets of experiments. First,  
2 we compared learning *with* different alphabet refinement heuristics to learning  
3 *without* alphabet refinement for 2-way decompositions with rule ASYM. Then, we  
4 compared the recursive implementation of the refinement algorithm with ASYM to  
5 monolithic (non-compositional) verification, for increasing number of components.  
6 All the experiments were performed on a Dell PC with a 2.8 GHz Intel Pentium 4  
7 CPU and a 1.0 GB RAM, running Linux Fedora Core 4 and using Sun’s Java SDK  
8 version 1.5. For the first set of experiments, for Gas Station and Chiron we used the  
9 best 2-way decompositions described in [12]. For Gas Station, the operator and the  
10 first pump are one component, and the rest of the modules are the other. For Chiron,  
11 the event dispatchers are one component, and the rest of the modules are the other.  
12 For MER, half of the users are in one component, and the other half with the ar-  
13 biter in the other. For the Rover we used the two components described in [13]. For  
14 the second set of experiments, we used an additional heuristic to compute the *or-*  
15 *dering* of the modules in the sequence  $M_1, \dots, M_n$  for the recursive learning with  
16 refinement so as to minimize the sizes of the interface alphabets  $\Sigma_I^1, \dots, \Sigma_I^n$ . We  
17 generated offline all possible orders with their associated interface alphabets and  
18 then chose the order that minimizes the sum  $\sum_{j=1..n} |\Sigma_I^j|$ .

19 The experimental results shown in Tables 3 and 4 are for running the learning  
20 framework with ‘No refinement’, and for refinement with backward (‘+bwd’), for-  
21 ward (‘+fwd’) and ‘+allDiff’ heuristics. For each run, we report  $|A|$  (the *maximum*  
22 assumption size reached during learning), ‘Mem.’ (the *maximum* memory used by  
23 LTSA to check assume-guarantee triples, measured in MB) and ‘Time’ (total CPU  
24 running time, measured in seconds). Column ‘Monolithic’ reports the memory and  
25 run-time of non-compositional model checking. We set a limit of 30 minutes for  
26 each run. The sign ‘–’ indicates that the limit of 1GB of memory or the time limit  
27 has been exceeded. For these cases, the data is reported as it was when the limit  
28 was reached.

Table 4

Comparison of learning for ASYM rule with and without alphabet refinement, and monolithic verification.

Case	$k$	ASYM			ASYM+ref			Monolithic	
		$ A $	Mem.	Time	$ A $	Mem.	Time	Mem.	Time
Gas Station	3	473	109.97	–	25	2.41	13.29	1.41	0.034
	4	287	203.05	–	25	3.42	22.50	2.29	0.13
	5	268	283.18	–	25	5.34	46.90	6.33	0.78
Chiron, Property 1	2	352	343.62	–	4	0.93	2.38	0.88	0.041
	3	182	114.57	–	4	1.18	2.77	1.53	0.062
	4	182	116.66	–	4	2.13	3.53	2.75	0.147
	5	182	115.07	–	4	7.82	6.56	13.39	1.202
Chiron, Property 2	2	190	107.45	–	11	1.68	40.11	1.21	0.035
	3	245	68.15	–	114	28	–	1.63	0.072
	4	245	70.26	–	103	23.81	–	2.89	0.173
	5	245	76.10	–	76	32.03	–	15.70	1.53
MER	2	40	8.65	21.90	6	1.23	1.60	1.04	0.024
	3	501	240.06	–	8	3.54	4.76	4.05	0.111
	4	273	101.59	–	10	9.61	13.68	14.29	1.46
	5	200	78.10	–	12	19.03	35.23	14.24	27.73
	6	162	84.95	–	14	47.09	91.82	–	600

1 Finally, we compared learning with and without alphabet refinement for rules SYM-  
2 N and CIRC-N under the same conditions as in the previous experiments. The re-  
3 sults are in Tables 5 and 6.

4 **Discussion** The results overall show that without alphabet refinement learning has  
5 limited performance, but alphabet refinement improves it significantly. Table 3  
6 shows that alphabet refinement improved the assumption size in all cases, and in a  
7 few, up to two orders of magnitude (see Gas Station with  $k = 2, 3$ , Chiron, Property  
8 3, with  $k = 5$ , MER with  $k = 3$ ). It improved memory consumption in 10 out of  
9 15 cases, and also improved running time, as for Gas Station and for MER with  
10  $k = 3, 4$  learning without refinement did not finish within the time limit, whereas  
11 with refinement it did. The benefit of alphabet refinement is even more obvious  
12 in Table 4: ‘No refinement’ exceeded the time limit in all but one case, whereas  
13 refinement completed in 14 of 16 cases, producing smaller assumptions and using  
14 less memory in all the cases, and up to two orders of magnitude in a few. Table 3  
15 also indicates that the performance of the ‘bwd’ strategy is (slightly) better than  
16 the other refinement strategies. Therefore we used this strategy for the experiments  
17 reported in Table 4.

18 Table 4 indicates that learning with refinement scales better than without refine-  
19 ment for increasing number of components. As  $k$  increases, the memory and time  
20 consumption for ‘Refinement’ grows slower than that of ‘Monolithic’. For Gas Sta-  
21 tion, Chiron (Property 1), and MER, for small values of  $k$ , ‘Refinement’ consumes

Table 5

Comparison of learning for SYM rule with and without alphabet refinement.

Case	$k$	SYM			SYM+ref		
		$ A $	Mem.	Time	$ A $	Mem.	Time
Gas Station	3	7	1.34	–	83	31.94	874.39
	4	7	2.05	–	139	38.98	–
	5	7	2.77	–	157	52.10	–
Chiron, Property 1	2	19	2.21	–	21	4.56	52.14
	3	19	2.65	–	21	4.99	65.50
	4	19	4.70	–	21	6.74	70.40
	5	19	17.65	–	21	28.38	249.3
Chiron, Property 2	2	7	1.16	–	8	0.93	6.35
	3	7	1.36	–	16	1.43	9.40
	4	7	2.29	–	32	3.51	16.00
	5	7	8.20	–	64	20.90	57.94
MER	2	40	6.56	9.00	6	1.69	1.64
	3	64	11.90	25.95	8	3.12	4.03
	4	88	1.82	83.18	10	9.61	9.72
	5	112	27.87	239.05	12	19.03	22.74
	6	136	47.01	608.44	14	47.01	47.90

1 more memory than ‘Monolithic’, but as  $k$  increases the gap is narrowing, and for  
2 the largest  $k$  ‘Refinement’ becomes better than ‘Monolithic’. This leads to cases  
3 such as MER with  $k = 6$  where, for a large enough parameter value, ‘Monolithic’  
4 runs out of memory, whereas ‘Refinement’ succeeds.

5 Tables 5 and 6 indicate that generally rules SYM-N and CIRC-N do not improve the  
6 performance of learning or the effect of alphabet refinement, but they can some-  
7 times handle cases which were challenging for ASYM, as is the case of SYM-N for  
8 Chiron, property 2. Thus there is some benefit in using these rules complementary  
9 to each other.

## 10 7 Related work

11 Several frameworks have been proposed to support assume-guarantee reason-  
12 ing [20,25,11,18]. For example, the Calvin tool [15] uses assume-guarantee rea-  
13 soning for the analysis of Java programs, while Mocha [2] supports modular ver-  
14 ification of components with requirements specified based in the Alternating-time  
15 Temporal logic. The practical impact of these previous approaches has been limited  
16 because they require non-trivial human input in defining appropriate assumptions.

17 Previous work [17,13] proposed to use  $L^*$  to automate assume-guarantee reason-  
18 ing. Since then, several other frameworks that use  $L^*$  for learning assumptions have

Table 6

Comparison of learning for CIRC rule with and without alphabet refinement.

Case	$k$	CIRC			CIRC+ref		
		$ A $	Mem.	Time	$ A $	Mem.	Time
Gas Station	3	205	108.96	–	25	2.43	15.10
	4	205	107.00	–	25	3.66	25.90
	5	199	105.89	–	25	5.77	58.74
Chiron, Property 1	2	259	78.03	–	4	0.96	2.71
	3	253	77.26	–	4	1.20	3.11
	4	253	77.90	–	4	2.21	3.88
	5	253	81.43	–	4	7.77	7.14
Chiron, Property 2	2	67	100.91	–	327	44.17	–
	3	245	75.76	–	114	26.61	–
	4	245	77.93	–	103	23.93	–
	5	245	81.33	–	76	32.07	–
MER	2	148	597.30	–	6	1.89	1.51
	3	281	292.01	–	8	3.53	4.00
	4	239	237.22	–	10	9.60	10.64
	5	221	115.37	–	12	19.03	27.56
	6	200	88.00	–	14	47.09	79.17

1 been developed – [3] presents a symbolic BDD implementation using NuSMV. This  
 2 symbolic version was extended in [23] with algorithms that decompose models  
 3 using hypergraph partitioning, to optimize the performance of learning on result-  
 4 ing decompositions. Different decompositions are also studied in [12] where the  
 5 best two-way decompositions are computed for model-checking with the LTSA  
 6 and FLAVERS tools. L\* has also been used in [1] to synthesize interfaces for Java  
 7 classes, and in [27] to check component compatibility after component updates.

8 Our approach for alphabet refinement is similar in spirit to counterexample-guided  
 9 abstraction refinement (CEGAR) [9]. CEGAR computes and analyzes abstractions  
 10 of programs (usually using a set of abstraction predicates) and refines them based  
 11 on spurious counter-examples. However, there are some important differences be-  
 12 tween CEGAR and our algorithm. Alphabet refinement works on actions rather  
 13 than predicates, it is applied compositionally in an assume-guarantee style and  
 14 it computes under-approximations (of assumptions) rather than behavioral over-  
 15 approximations (as it happens in CEGAR). In the future, we plan to investigate  
 16 more the relationship between CEGAR and our algorithm. The work of [19] pro-  
 17 poses a CEGAR approach to interface synthesis for C libraries. This work does  
 18 not use learning, nor does it address the use of the resulting interfaces in assume-  
 19 guarantee verification.

20 A similar idea to our alphabet refinement for L\* in the context of assume guarantee  
 21 verification has been developed independently in [6]. In that work, L\* is started  
 22 with an empty alphabet, and, similarly to ours, the assumption alphabet is refined

1 when a spurious counterexample is obtained. At each refinement stage, a new minimal  
2 alphabet is computed that eliminates all spurious counterexamples seen so far.  
3 The computation of such a minimal alphabet is shown to be NP-hard. In contrast,  
4 we use much cheaper heuristics, but do not guarantee that the computed alphabet is  
5 minimal. The approach presented in [28] improves upon assume guarantee learning  
6 for systems that communicate based on shared memory, by using SAT based model  
7 checking and alphabet clustering.

8 **Corina: add Maier’s circular paper???** **i am not sure about it since we actually**  
9 **that circular rules can be sound and complete???**

## 10 **8 Conclusions and Future Work**

11 We have introduced a framework that uses a learning algorithm to synthesize as-  
12 sumptions that automate assume guarantee reasoning for finite state machines and  
13 safety safety. The framework incorporates symmetric, asymmetric and circular as-  
14 sume guarantee rules and also uses alphabet refinement to compute small assump-  
15 tions alphabet that are sufficient for verification. The framework has been applied  
16 to a variety of systems where it showed its effectiveness.

17 In future work we plan to look beyond checking safety properties and to address fur-  
18 ther algorithmic optimizations, e.g. reuse of query results and learning tables across  
19 alphabet refinement stages. Moreover, we plan to explore techniques alternative to  
20 learning for computing assumption, e.g. we are investigating CEGAR-like tech-  
21 niques for computing assumptions incrementally as abstractions of environments.  
22 Finally we plan to perform more experiments to fully evaluate our technique.

## 23 **References**

- 24 [1] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. “Synthesis of interface specifications  
25 for Java classes”. In *Proceedings of POPL’05*, pages 98–109, 2005.
- 26 [2] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. “MOCHA:  
27 Modularity in Model Checking”. In *Proceedings of CAV’98*, volume 1427 of *LNCS*,  
28 pages 521–525, 1998.
- 29 [3] R. Alur, P. Madhusudan, and Wonhong Nam. “Symbolic Compositional Verification  
30 by Learning Assumptions”. In *Proceedings of CAV05*, pages 548–562, 2005.
- 31 [4] D. Angluin. “Learning regular sets from queries and counterexamples”. *Information*  
32 *and Computation*, 75(2):87–106, November 1987.

- 1 [5] H. Barringer, D. Giannakopoulou, and C. S. Pasareanu. “Proof Rules for Automated  
2 Compositional Verification through Learning”. In *Proceedings of SAVCBS’03*, pages  
3 14–21, 2003.
- 4 [6] S. Chaki and O. Strichman. “Optimized L\*-based Assume-guarantee Reasoning”. In  
5 *Proceedings of TACAS’07 (to appear)*, 2007.
- 6 [7] S.C. Cheung and J. Kramer. Checking safety properties using compositional  
7 reachability analysis. *ACM Transactions on Software Engineering and Methodology*,  
8 8(1):49–78, 1999.
- 9 [8] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE*  
10 *Transactions on Software Engineering*, SE-4(3):178–187, May 1978.
- 11 [9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. “Counterexample-Guided  
12 Abstraction Refinement”. In *Proceedings of CAV’00*, volume 1855 of *LNCS*, pages  
13 154–169, 2000.
- 14 [10] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
- 15 [11] E. M. Clarke, D. E. Long, and K. L. McMillan. “Compositional Model Checking”. In  
16 *Proceedings of LICS’89*, pages 353–362, 1989.
- 17 [12] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. “Breaking Up is Hard to Do: An  
18 Investigation of Decomposition for Assume-Guarantee Reasoning”. In *Proceedings*  
19 *of ISSTA’06*, pages 97–108. ACM Press, 2006.
- 20 [13] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. “Learning Assumptions for  
21 Compositional Verification”. In *Proceedings of TACAS’03*, volume 2619 of *LNCS*,  
22 pages 331–346, 2003.
- 23 [14] T. Dean and M. S. Boddy. An analysis of time-dependent planning. In *Proc. of the 7th*  
24 *National Conf. on Artificial Intelligence*, pages 49–54, Aug. 1988.
- 25 [15] C. Flanagan, S. N. Freund, and S. Qadeer. “Thread-Modular Verification for Shared-  
26 Memory Programs”. In *Proceedings of ESOP’02*, pages 262–277, 2002.
- 27 [16] M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. “Refining Interface  
28 Alphabets for Compositional Verification”. In *Proceedings of TACAS’07*, volume 4424  
29 of *LNCS*, pages 292–307, 2007.
- 30 [17] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. “Assumption Generation for  
31 Software Component Verification”. In *Proceedings of ASE’02*, pages 3–12. IEEE  
32 Computer Society, 2002.
- 33 [18] O. Grumberg and D. E. Long. “Model Checking and Modular Verification”. In  
34 *Proceedings of CONCUR’91*, pages 250–265, 1991.
- 35 [19] T. A. Henzinger, R. Jhala, and R. Majumdar. “Permissive Interfaces”. In *Proceedings*  
36 *of ESEC/SIGSOFT FSE’05*, pages 31–40, 2005.
- 37 [20] C. B. Jones. “Specification and Design of (Parallel) Programs”. In *Information*  
38 *Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP:  
39 North Holland, 1983.

- 1 [21] J.-P. Krimm and L. Mounier. “Compositional State Space Generation from Lotos  
2 Programs”. In *Proceedings of TACAS’97*, pages 239–258, 1997.
- 3 [22] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley &  
4 Sons, 1999.
- 5 [23] W. Nam and R. Alur. “Learning-Based Symbolic Assume-Guarantee Reasoning with  
6 Automatic Decomposition”. In *Proceedings of ATVA’06*, volume 4218 of *LNCS*, 2006.
- 7 [24] C. S. Pasareanu and D. Giannakopoulou. “Towards a Compositional SPIN”. In  
8 *Proceedings of SPIN’06*, volume 3925 of *LNCS*, pages 234–251, 2006.
- 9 [25] A. Pnueli. “In Transition from Global to Modular Temporal Reasoning about  
10 Programs”. In *Logic and Models of Concurrent Systems*, volume 13, pages 123–144,  
11 1984.
- 12 [26] R. L. Rivest and R. E. Shapire. “Inference of finite automata using homing sequences”.  
13 *Information and Computation*, 103(2):299–347, April 1993.
- 14 [27] N. Sharygina, S. Chaki, E. Clarke, and N. Sinha. “Dynamic Component  
15 Substitutability Analysis”. In *Proceedings of FM’05*, volume 3582 of *LNCS*, pages  
16 512–528, 2005.
- 17 [28] N. Sinha and E. Clarke. “SAT-Based Compositional Verification Using Lazy  
18 Learning”. In *Proceedings of CAV*, 2007.