Learning to divide and conquer: applying the L^* algorithm to automate assume-guarantee reasoning.

Puasuareanu, Corina and Giannakopoulou, Dimitra and Bobaru, Mihaela Gheorghiu and Cobleigh, Jamieson M and Barringer, Howard

2008

MIMS EPrint: **2010.56**

ISSN 1749-9097

Form Methods Syst Des (2008) 32: 175–205
DOI 10.1007/s10703-008-0049-6

Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning

Corina S. Păsăreanu · Dimitra Giannakopoulou · Mihaela Gheorghiu Bobaru · Jamieson M. Cobleigh · Howard Barringer

Published online: 25 January 2008

Abstract Assume-guarantee reasoning enables a "divide-and-conquer" approach to the verification of large systems that checks system components separately while using *assumptions* about each component's environment. Developing appropriate assumptions used to be a difficult and manual process. Over the past five years, we have developed a framework for performing assume-guarantee verification of systems in an incremental and fully automated fashion. The framework uses an off-the-shelf learning algorithm to compute the assumptions. The assumptions are initially approximate and become more precise by means of counterexamples obtained by model checking components separately. The framework supports different assume-guarantee rules, both symmetric and asymmetric. Moreover, we have recently introduced *alphabet refinement*, which extends the assumption learning process to also infer *assumption alphabets*. This refinement technique starts with assumption alpha-

J.M. Cobleigh currently employed by The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760, USA.

C.S. Păsăreanu (✉)
Perot Systems, NASA Ames Research Center, N269-230, Moffett Field, CA 94035, USA
e-mail: Corina.S.Pasareanu@nasa.gov

D. Giannakopoulou
RIACS, NASA Ames Research Center, N269-230, Moffett Field, CA 94035, USA
e-mail: Dimitra.Giannakopoulou@nasa.gov

M.G. Bobaru
Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4
e-mail: mg@cs.toronto.edu

J.M. Cobleigh
Department of Computer Science, University of Massachusetts, 140 Governor's Drive, Amherst, MA 01003, USA
e-mail: jcobleig@cs.umass.edu

H. Barringer
School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK
e-mail: howard.barringer@manchester.ac.uk

bets that are a subset of the minimal interface between a component and its environment, and adds actions to it as necessary until a given property is shown to hold or to be violated in the system. We have applied the learning framework to a number of case studies that show that compositional verification by learning assumptions can be significantly more scalable than non-compositional verification.

Keywords Assume-guarantee reasoning · Model checking · Labeled transition systems · Learning · Proof rules · Compositional verification · Safety properties

1 Introduction

Model checking is an effective technique for finding subtle errors in concurrent systems. Given a finite model of a system and a required property of that system, model checking determines automatically whether the property is satisfied by the system. The cost of model checking techniques may be exponential in the size of the system being verified, a problem known as state explosion [12]. This can make model checking intractable for systems of realistic size.

Compositional verification techniques address the state-explosion problem by using a "divide-and-conquer" approach: properties of the system are decomposed into properties of its components and each component is then checked separately. In checking components individually, it is often necessary to incorporate some knowledge of the context in which each component is expected to operate correctly. Assume-guarantee reasoning [22, 28] addresses this issue by using *assumptions* that capture the expectations that a component makes about its environment. Assumptions have traditionally been developed manually, which has limited the practical impact of assume-guarantee reasoning.

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

Compositional techniques have been shown particularly effective for well-structured systems that have small interfaces between components [8, 18]. Interfaces consist of *all* communication points through which components may influence each other's behavior. In our initial presentations of the framework [6, 13] the alphabets of the assumption automata included *all* the actions in the component interface. In a case study presented in [27], however, we observed that a smaller alphabet can be sufficient to prove a property. This smaller alphabet was determined through manual inspection and with it, assume-guarantee reasoning achieves orders of magnitude improvement over monolithic, *i.e.*, non-compositional, model checking [27].

Motivated by the successful use of a smaller assumption alphabet in learning, we investigated in [17] whether the process of discovering a smaller alphabet that is sufficient for checking the desired properties can be automated. Smaller alphabets mean smaller interfaces among components, which may lead to smaller assumptions, and hence to smaller verification problems. We developed an *alphabet refinement* technique that extends the learning framework so that it starts with a small subset of the interface alphabet and adds actions to it as necessary until a required property is either shown to hold or shown to be violated by the system. Actions to be added are discovered by analysis of the counterexamples obtained from model checking the components.

The learning framework and the alphabet refinement have been implemented within the LTSA model checking tool [24] and they have been effective in verifying realistic concurrent systems, such as the ones developed in NASA projects. This paper presents and expands the material presented in [13] (original learning framework for automated assume-guarantee reasoning with an asymmetric rule), [6] (learning for symmetric rules), and [17] (alphabet refinement for the original framework). In addition, we describe here a new extension that uses a circular rule, alphabet refinement for symmetric and circular rules, and present new experimental data.

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

2 Preliminaries

In this section we give background information for our work: we introduce labeled transition systems and finite-state machines, together with their associated operators, and also present how properties are expressed and checked in this context. We also introduce assume-guarantee reasoning and the notion of the weakest assumption. Moreover we provide a detailed description of the learning algorithm that we use to automate assume-guarantee reasoning. The reader may wish to skip this section on the first reading.

2.1 Labeled transition systems (LTSs)

Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment. We use π to denote a special *error state*, which models the fact that a safety violation has occurred in the associated transition system. We require that the error state have no outgoing transitions. Formally, an LTS M is a four-tuple $\langle Q, \alpha M, \delta, q_0\rangle$ where:

- Q is a finite non-empty set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a set of observable actions called the *alphabet* of M
- $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is a transition relation
- $q_0 \in Q$ is the initial state

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

As an example, consider a simple communication channel that consists of two components whose LTSs are shown in Fig. 1. Note that the initial state of all LTSs in this paper is state 0. The *Input* LTS receives an input when the action *input* occurs, and then sends it to the *Output* LTS with action *send*. After being sent some data, *Output* produces some output using the action *output* and acknowledges that it has finished, by using the action *ack*. At this point, both LTSs return to their initial states so the process can be repeated.

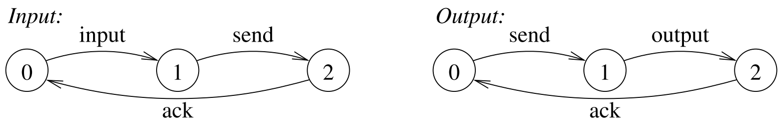


Fig. 1 Example LTSs

2.1.1 Traces

A *trace* t of an LTS M is a finite sequence of observable actions that label the transitions that M can perform starting at its initial state (ignoring the τ-transitions). For example, $\langle \text{input} \rangle$ and $\langle \text{input}, \text{send} \rangle$ are both traces of the *Input* LTS in Fig. 1. We sometimes abuse this notation and denote by t both a trace and its trace LTS. For a trace t of length n, its trace LTS consists of $n+1$ states, where there is a transition between states m and $m+1$ on the mth action in the trace t. The set of all traces of an LTS M is the language of M and is denoted $\mathcal{L}(M)$. We denote as *errTr*(M) the set of traces that lead to π, which are called the *error traces* of M.

For $\Sigma \subseteq \mathcal{A}ct$, we use $t\lceil\Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. Similarly, $M\lceil\Sigma$ is defined to be an LTS over alphabet Σ which is obtained from M by renaming to τ all the transitions labeled with actions that are not in Σ. Let t, t' be two traces. Let Σ, Σ' be the sets of actions occurring in t, t', respectively. By the *symmetric difference* of t and t' we mean the symmetric difference of the sets Σ and Σ'.

2.1.2 Parallel composition

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 M' with action a, denoted $M \xrightarrow{a} M'$, if and only if $(q_0, a, q_0') \in \delta$ and either $Q = Q'$, $\alpha M = \alpha M'$, and $\delta = \delta'$ for $q_0' \neq \pi$, or, in the special case where $q_0' = \pi$, $M' = \Pi$.

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

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. If $M_1 = \Pi$ or $M_2 = \Pi$, then $M_1 \parallel M_2 = \Pi$. Otherwise, $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, 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 δ is defined as follows, where a is either an observable action or τ:

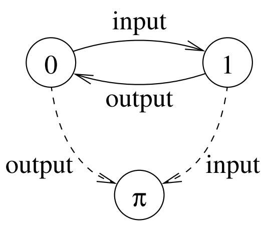
$$\frac{M_1 \xrightarrow{a} M_1', a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2}, \qquad \frac{M_2 \xrightarrow{a} M_2', a \notin \alpha M_1}{M_1 \parallel M_2 \xrightarrow{a} M_1 \parallel M_2'},$$

$$\frac{M_1 \xrightarrow{a} M_1', M_2 \xrightarrow{a} M_2', a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'}.$$

2.1.3 Properties

We call a deterministic LTS that contains no π states a *safety LTS*. A *safety property* is specified as a safety LTS P, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors

Fig. 2 *Order* property



over αP. For an LTS M and a safety LTS P such that $\alpha P \subseteq \alpha M$, we say that M satisfies P, denoted $M \models P$, if and only if $\forall t \in \mathcal{L}(M) : (t \restriction \alpha P) \in \mathcal{L}(P)$.

When checking a property P, an *error LTS* denoted P_{err} is created, which traps possible violations with the π 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

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

Note that the error LTS is *complete*, meaning each state other than the error state has outgoing transitions for every action in its alphabet. Also note that the error traces of P_{err} define the language of P's complement (see Sect. 2.2.3 below).

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

To detect violations of a property P by a component M, the parallel composition $M \parallel P_{err}$ is computed. It has been proved that M violates P if and only if the π state is reachable in $M \parallel P_{err}$ [8]. For example, state π is not reachable in $Input \parallel Output \parallel Order_{err}$, so we conclude that $Input \parallel Output \models Order$.

2.2 LTSs and finite-state machines

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

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

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

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

We call an FSM M *deterministic* if and only if $LTS(M)$ is deterministic.

2.2.1 Parallel composition of FSMs

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. Then $M_1 \parallel M_2$ is an FSM $M = \langle Q, \alpha M, \delta, q_0, F\rangle$, where:

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

Note 1

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

2.2.2 Properties

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

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

2.2.3 Complementation

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

2.3 Assume-guarantee reasoning

2.3.1 Assume-guarantee triples

In the assume-guarantee paradigm a formula is a triple $\langle A\rangle M\langle P\rangle$, where M is a component, P is a property, and A is an assumption about M's environment. The formula is true if whenever M is part of a system satisfying A, then the system must also guarantee P [19, 28], *i.e.*, $\forall E$, $E \parallel M \models A$ implies $E \parallel M \models P$. For LTS M and safety LTSs A and P, checking $\langle A\rangle M\langle P\rangle$ reduces to checking if state π is reachable in $A \parallel M \parallel P_{err}$. Note that when $\alpha P \subseteq \alpha A \cup \alpha M$, this is equivalent to $A \parallel M \models P$. Also note that we assume that M contains no π states.

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

Proof

- "$\Rightarrow$": Assume $\langle A\rangle M\langle P\rangle$ is true. We show that π is unreachable in $A \parallel M \parallel P_{err}$ by contradiction. Assume π is reachable in $A \parallel M \parallel P_{err}$ by a trace t. As a result, $t{\upharpoonright}\alpha A \in \mathcal{L}(A)$, $t{\upharpoonright}\alpha M \in \mathcal{L}(M)$, and $t{\upharpoonright}\alpha P \in errTr(P_{err})$ (see Note 1).

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

From our hypothesis that $\langle A\rangle M\langle P\rangle$ is true, it follows that $E \parallel M \models A$ implies $E \parallel M \models P$. However, $t{\upharpoonright}\alpha E \in \mathcal{L}(E)$, $t{\upharpoonright}\alpha M \in \mathcal{L}(M)$, and $t{\upharpoonright}\alpha P \in \mathit{errTr}(P_{err})$. Moreover t's actions belong to $\alpha E \cup \alpha M \cup \alpha P$. Therefore π is reachable in $E \parallel M \parallel P_{err}$ on trace t. As a result, we can conclude that $E \parallel M \not\models P$, which is a contradiction. Thus, π is not reachable in $A \parallel M \parallel P_{err}$, as desired.

- "$\Leftarrow$": Assume π is unreachable in $A \parallel M \parallel P_{err}$. We show that $\langle A\rangle M\langle P\rangle$ by contradiction. Assume $\langle A\rangle M\langle P\rangle$ is not true, *i.e.*, assume $\exists E$ such that $E \parallel M \models A$ but $E \parallel M \not\models P$. (Again, we assume that αE is such that $\models$ is well defined in the previous sentence.)

  Since $E \parallel M \not\models P$ then π is reachable in $E \parallel M \parallel P_{err}$ by some trace t. As a result, $t{\upharpoonright}\alpha E \in \mathcal{L}(E)$, $t{\upharpoonright}\alpha M \in \mathcal{L}(M)$, and $t{\upharpoonright}\alpha P \in \mathit{errTr}(P_{err})$. Since $E \parallel M \models A$ and $\alpha A \subseteq \alpha E \cup \alpha M$, it follows that $t{\upharpoonright}\alpha A \in \mathcal{L}(A)$. As a result, π is reachable in $A \parallel M \parallel P_{err}$ by $t{\upharpoonright}(\alpha A \cup \alpha M \cup \alpha P)$, which is a contradiction. Thus, $\langle A\rangle M\langle P\rangle$ is true, as desired. □

2.3.2 *Weakest assumption*

A central notion of our work is that of the *weakest assumption* [18], defined formally here.

Definition 2 (Weakest Assumption for Σ) Let M_1 be an LTS for a component, P be a safety LTS for a property required of M_1, and Σ be the interface of the component to the environment. The weakest assumption $A_{w,\Sigma}$ of M_1 for Σ and for property P is a deterministic LTS such that: (1) $\alpha A_{w,\Sigma} = \Sigma$, and (2) for any component M_2, $\langle \mathit{true}\rangle M_1 \parallel (M_2{\upharpoonright}\Sigma)\langle P\rangle$ if and only if $\langle \mathit{true}\rangle M_2\langle A_{w,\Sigma}\rangle$.

The notion of a weakest assumption depends on the interface between the component and its environment. Accordingly, in the second condition above, projecting M_2 onto Σ forces M_2 to communicate with M_1 only through actions in Σ. In [18] we showed that weakest assumptions exist for components expressed as LTSs and properties expressed as safety LTSs. Additionally, we provided an algorithm for computing weakest assumptions.

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

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

Definition 4 (Weakest Assumption) Given M_1, M_2, and P as above, the weakest assumption A_w is defined as A_{w,Σ_I}.

Note that from the above definitions, it follows that $\langle \mathit{true}\rangle M_1 \parallel M_2\langle P\rangle$ if and only if $\langle \mathit{true}\rangle M_2\langle A_w\rangle$. The following lemma will be used later in the paper.

Lemma 5 *Given M_1, P, and Σ as above, then $\langle A_{w,\Sigma}\rangle M_1\langle P\rangle$ holds.*

Proof $A_{w,\Sigma}{\upharpoonright}\Sigma = A_{w,\Sigma}$. If in Definition 2 we substitute $A_{w,\Sigma}$ for M_2, we obtain that: $\langle \mathit{true}\rangle M_1 \parallel A_{w,\Sigma}\langle P\rangle$ if and only if $\langle \mathit{true}\rangle A_{w,\Sigma}\langle A_{w,\Sigma}\rangle$. But the latter holds trivially, so we conclude that $\langle \mathit{true}\rangle M_1 \parallel A_{w,\Sigma}\langle P\rangle$, which is equivalent to $\langle A_{w,\Sigma}\rangle M_1\langle P\rangle$, always holds. □

```
(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

2.4 The L* learning algorithm

The learning algorithm L* was developed by Angluin [4] and later improved by Rivest and Schapire [29]. L* learns an unknown regular language U over alphabet Σ and produces a deterministic finite-state machine (DFSM) that accepts it. L* interacts with a *Minimally Adequate Teacher*, henceforth referred to as the *Teacher*, that answers two types of questions. The first type is a membership *query*, in which L* asks whether a string $s \in \Sigma^*$ is in U. The second type is a *conjecture*, in which L* asks whether a conjectured DFSM C is such that $\mathcal{L}(C) = U$. If $\mathcal{L}(C) \neq U$ the Teacher returns a counterexample, which is a string s in the symmetric difference of $\mathcal{L}(C)$ and U.

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

*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 the observation table (S, E, T) where S and E are a set of prefixes and suffixes, respectively, both over Σ^*. In addition, T is a function mapping $(S \cup S \cdot \Sigma) \cdot E$ to {true, 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 sequences p and q. Initially, L* sets S and E to $\{\lambda\}$ (line 1), where λ 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

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

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

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

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

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

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

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

2.4.2 *Characteristics of L**

L* is guaranteed to terminate with a minimal automaton M for the unknown language U. Moreover, for each closed observation table (S, E, T), the candidate DFSM C that L* constructs is smallest, in the sense that any other DFSM consistent[1] with the function T has at least as many states as C. This characteristic of L* makes it particularly attractive for our framework. The conjectures made by L* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than M. Therefore, if M has n states, L* makes at most $(n - 1)$ incorrect conjectures. The number of membership queries made by L* is $\mathcal{O}(kn^2 + n \log m)$, where k is the size of the alphabet of U, n is the number of states in the minimal DFSM for U, and m is the length of the longest counterexample returned when a conjecture is made.

3 Learning for assume-guarantee reasoning

In this section we introduce a simple, asymmetric assume-guarantee rule and we describe a framework which uses L* to learn assumptions that automate reasoning about two compo-

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

nents based on this rule. We also discuss how the framework has been extended to reason about n components and to use circular rules.

3.1 Assume-guarantee rule ASYM

Our framework incorporates a number of symmetric and asymmetric rules for assume-guarantee reasoning. The simplest assume-guarantee proof is for checking a property P on a system with two components M_1 and M_2 and is as follows [19]:

Rule ASYM

$$\frac{\begin{array}{l}1: \langle A\rangle M_1 \langle P\rangle \\ 2: \langle \mathit{true}\rangle M_2 \langle A\rangle\end{array}}{\langle \mathit{true}\rangle M_1 \parallel M_2 \langle P\rangle}$$

In this rule, A denotes an assumption about the environment in which M_1 is placed. Soundness of the rule follows from $\langle \mathit{true}\rangle M_2 \langle A\rangle$ implies $\langle \mathit{true}\rangle M_1 \parallel M_2 \langle A\rangle$ and from the definition of assume-guarantee triples. Completeness holds trivially, by substituting M_2 for A.

Note that the rule is not symmetric in its use of the two components, and does not support circularity. Despite its simplicity, our experience with applying compositional verification to several applications has shown it to be most useful in the context of checking safety properties.

For the use of rule ASYM to be justified, the assumption must be more abstract than M_2, but still reflect M_2's behavior. Additionally, an appropriate assumption for the rule needs to be strong enough for M_1 to satisfy P in premise 1. Developing such an assumption is difficult to do manually. In the following, we describe a framework that uses L* to learn assumptions automatically.

3.2 Learning framework for rule ASYM

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

3.2.1 The Teacher

To explain how the teacher answers queries and conjectures we use the following lemma.

Lemma 6 *Let $t \in \Sigma^*$. Then $t \in \mathcal{L}(A_w)$ if and only if $\langle t\rangle M_1 \langle P\rangle$ holds. In the assume-guarantee triple, we treat t as its corresponding trace LTS with the alphabet set to Σ.*

Proof By Theorem 1, $\langle t\rangle M_1 \langle P\rangle$ holds if and only if π is unreachable in $t \parallel M_1 \parallel P_{err}$, which is equivalent to checking $\langle \mathit{true}\rangle M_1 \parallel t \langle P\rangle$. By Definition 2, this is the same as checking $\langle \mathit{true}\rangle t \langle A_w\rangle$, which is equivalent to checking $t \in \mathcal{L}(A_w)$. □

Answering queries Recall that L* makes a query by asking whether a trace t is in the language being learned, which is $\mathcal{L}(A_w)$. The Teacher must return true if t is in $\mathcal{L}(A_w)$ and false otherwise. To answer a query, the Teacher uses LTSA to check $\langle t\rangle M_1 \langle P\rangle$ (here t is treated as a trace LTS and its alphabet is Σ). From Lemma 6 it follows if this check is false, then $t \notin \mathcal{L}(A_w)$ and false is returned to L*. Otherwise, $t \in \mathcal{L}(A_w)$ and true is returned to L*.

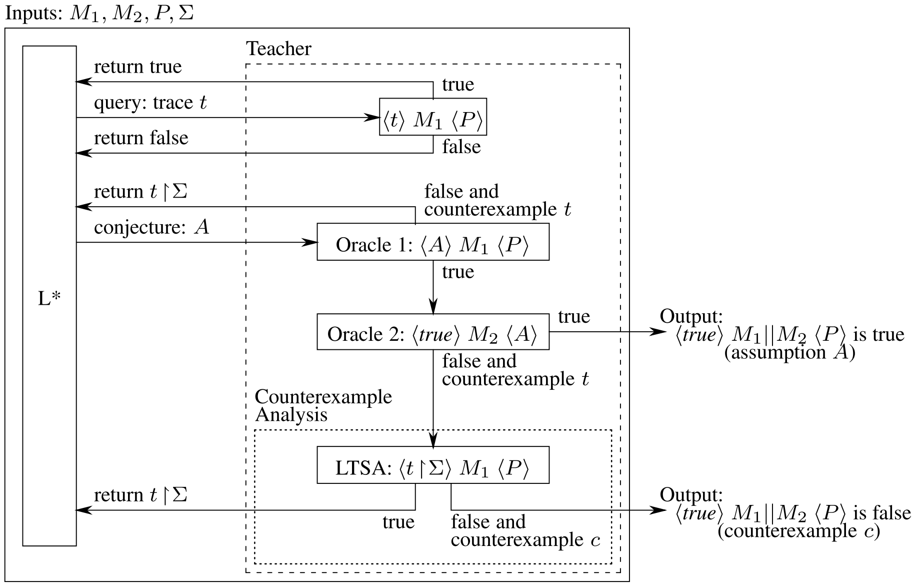

Inputs: M_1, M_2, P, Σ
Teacher
return true
true
query: trace t
$\langle t \rangle\ M_1\ \langle P \rangle$
return false
false
return $t\restriction\Sigma$
false and counterexample t
conjecture: A
Oracle 1: $\langle A \rangle\ M_1\ \langle P \rangle$
true
L*
Oracle 2: $\langle true \rangle\ M_2\ \langle A \rangle$
true
Output: $\langle true \rangle\ M_1 || M_2\ \langle P \rangle$ is true (assumption A)
false and counterexample t
Counterexample Analysis
LTSA: $\langle t\restriction\Sigma \rangle\ M_1\ \langle P \rangle$
return $t\restriction\Sigma$
true
false and counterexample c
Output: $\langle true \rangle\ M_1 || M_2\ \langle P \rangle$ is false (counterexample c)

Fig. 4 Learning framework for rule ASYM

Answering conjectures A conjecture consists of an FSM that L* believes will recognize the language being learned. The Teacher must return true if the conjecture is correct. Otherwise, the Teacher must return false and a counterexample that witnesses an error in the conjectured FSM, *i.e.*, a trace in the symmetric difference of the language being learned and that of the conjectured automaton. In our framework, the conjectured FSM is an assumption that is being used to complete an assume-guarantee proof. We treat the conjectured FSM as an LTS, as described in Sect. 2.2, which we denote as the LTS A. To answer the conjecture, the Teacher uses two oracles:

- *Oracle 1* guides L* towards a conjecture that makes premise 1 of rule ASYM true. It checks $\langle A \rangle M_1 \langle P \rangle$ and if the result is false, then a counterexample t is produced. Since the $\langle A \rangle M_1 \langle P \rangle$ is false, we know that $t\restriction\Sigma \in \mathcal{L}(A)$. But, since π is reachable in $t\restriction\Sigma \parallel M_1 \parallel P_{err}$, by Lemma 6 we know that $t\restriction\Sigma \notin \mathcal{L}(A_w)$. Thus, $t\restriction\Sigma$ witnesses a difference between A and A_w so it is returned to L* to answer the conjecture. If the triple is true, then the Teacher moves on to Oracle 2.
- *Oracle 2* is invoked to check premise 2 of rule ASYM, *i.e.*, to discharge A on M_2 by verifying that $\langle true \rangle M_2 \langle A \rangle$ is true. This triple is checked and if it is true, then the assumption makes both premises true and thus, the assume-guarantee rule guarantees that $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$ is true. The Teacher then returns true and the computed assumption A. Note that A is not necessarily A_w, it can be *stronger* than A_w, *i.e.*, $\mathcal{L}(A) \subseteq \mathcal{L}(A_w)$, but the computed assumption is sufficient to prove that the property holds. If the triple is not true, then a counterexample t is produced. In this case further analysis is needed to determine if either P is indeed violated by $M_1 \parallel M_2$ or if A is not precise enough, in which case A needs to be modified.

Counterexample analysis The counterexample t from Oracle 2 must be analyzed to determine if it is a real counterexample, *i.e.*, if it causes $M_1 \parallel M_2$ to violate P. To do this,

Table 1 Mapping T_1

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

the Teacher performs a query on $t\lceil\Sigma$, in other words it uses LTSA to check $\langle t\lceil\Sigma\rangle M_1\langle P\rangle$ (here again $t\lceil\Sigma$ is treated as a trace LTS and its alphabet is Σ). If this triple is true, then by Lemma 6 we know that $t\lceil\Sigma \in \mathcal{L}(A_w)$. Since this trace caused $\langle true\rangle M_2\langle A\rangle$ to be false, we also know that $t\lceil\Sigma \notin \mathcal{L}(A)$, thus $t\lceil\Sigma$ witnesses a difference between A and A_w. Therefore, $t\lceil\Sigma$ is returned to L* to answer its conjecture.

If the triple $\langle t\lceil\Sigma\rangle M_1\langle P\rangle$ is false, then the model checker returns a (new) counterexample c that witnesses the violation of P on M_1 in the context of $t\lceil\Sigma$. With $\Sigma = \Sigma_I$, c is guaranteed to be a real error trace in $M_1 \parallel M_2 \parallel P_{err}$ (we will see in Sect. 5 that when Σ is only a subset of Σ_I, this is no longer the case). Thus, $\langle true\rangle M_1 \parallel M_2\langle P\rangle$ is false and c is returned to the user as a counterexample.

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

3.2.2 Example

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

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

The Teacher therefore returns counterexample $t\lceil\Sigma = \langle\text{send, ack}\rangle$ to L*, which uses queries to again update its observation table until it is closed. From this table, shown in Table 2, the assumption A_2, shown in Fig. 6, is constructed and conjectured to the Teacher. This time, Oracle 1 reports that $\langle A_2\rangle Input\langle Order\rangle$ is true, meaning the assumption is not too weak. The Teacher then calls Oracle 2 to determine if $\langle true\rangle Output\langle A_2\rangle$. This is also true, so the framework reports that $\langle true\rangle Input \parallel Output\langle Order\rangle$ is true.

Table 2 Mapping T_2

		E_2	
	T_2	λ	ack
S_2	λ	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

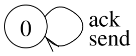


Fig. 5 A_1

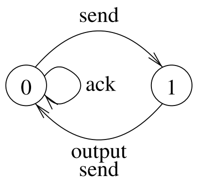


Fig. 6 A_2

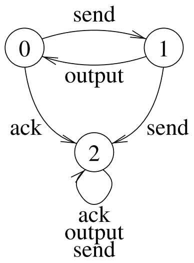


Fig. 7 A_3

This example did not involve weakening of the assumptions produced by L*, since the assumption A_2 was sufficient for the compositional proof. This will not always be the case. Consider *Output*$'$, shown in Fig. 9, which allows multiple send actions to occur before producing output. If *Output* were replaced by *Output*$'$, then the verification process would be identical to the previous case, until Oracle 2 is invoked by the Teacher for conjecture A_2. Oracle 2 returns that $\langle true\rangle Output' \langle A_2\rangle$ is false, with counterexample $\langle$send, send, output$\rangle$. The Teacher analyzes this counterexample and determines that in the context of this trace, *Input* does not violate *Order*. This trace (projected onto Σ) is returned to L*, which will weaken the conjectured assumption. The process involves two more iterations, during which assumptions A_3 (Fig. 7) and A_4 (Fig. 8), are produced. Using A_4, which is the weakest assump-

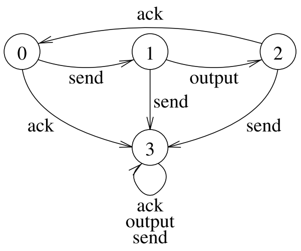


Fig. 8 A_4

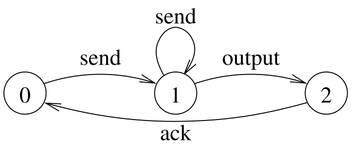


Fig. 9 LTS for *Output*$'$

tion A_w, both Oracles report true, so it can be concluded that $\langle true\rangle Input \parallel Output' \langle Order\rangle$ also holds.

3.2.3 Correctness and termination

Theorem 7 *Given components M_1 and M_2, and property P, the algorithm implemented by our framework for rule* ASYM *terminates and correctly reports on whether $\langle true\rangle M_1 \parallel M_2\langle P\rangle$ holds.*

Proof To prove the theorem, we first argue the correctness, and then the termination of our algorithm.

Correctness: The Teacher in our framework uses the two premises of the assume-guarantee rule to answer conjectures. It only reports that $\langle true\rangle M_1 \parallel M_2\langle P\rangle$ is true when both premises are true, and therefore correctness is guaranteed by the compositional rule. Our framework reports an error when it detects a trace t of M_2 which, when simulated on M_1, violates the property, which implies that $M_1 \parallel M_2$ violates P.

Termination: At any iteration, after an assumption is conjectured, our algorithm reports on whether $\langle true\rangle M_1 \parallel M_2\langle P\rangle$ is true and terminates, or continues by providing a counterexample to L*. By correctness of L*, we are guaranteed that if it keeps receiving counterexamples to conjectures, it will eventually, at some iteration i, produce A_w. During this iteration, Oracle 1 will return true by definition of A_w. The Teacher will therefore apply Oracle 2, which will return either true and terminate, or will return a counterexample. This counterexample represents a trace of M_2 that is not contained in $\mathcal{L}(A_w)$. Since, as discussed before, A_w is both necessary and sufficient, analysis of the counterexample will report that this is a real counterexample, and the algorithm will terminate. □

3.3 Generalization to n components

We presented our approach so far to the case of two components. Assume now that a system consists of $n \geq 2$ components. To check if system $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ satisfies P, we decompose it into: M_1 and $M_2' = M_2 \parallel M_3 \parallel \cdots \parallel M_n$ and the learning framework is applied recursively to check the second premise of the assume-guarantee rule.

At each recursive invocation for M_j and $M_j' = M_{j+1} \parallel M_{j+2} \parallel \cdots \parallel M_n$, we solve the following problem: find assumption A_j such that the following are both true:

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

Here A_{j-1} is the assumption for M_{j-1} and plays the role of the property for the current recursive call. Correctness and termination for this extension follows by induction on n from Theorem 7.

3.4 Extension with a circular rule

Our framework can accommodate a variety of assume-guarantee rules that are sound. Completeness of rules is required to guarantee termination. We investigate here another rule, that is similar to ASYM but it involves some form of circular reasoning. This rule appeared originally in [19] (for reasoning about two components). The rule can be extended easily to reasoning about $n \geq 2$ components.

Rule CIRC-N

$$\frac{\begin{array}{ll} 1: & \langle A_1\rangle M_1\langle P\rangle \\ 2: & \langle A_2\rangle M_2\langle A_1\rangle \\ \vdots & \\ n: & \langle A_n\rangle M_n\langle A_{n-1}\rangle \\ n+1: & \langle true\rangle M_1\langle A_n\rangle \end{array}}{\langle true\rangle M_1 \parallel M_2 \parallel \cdots \parallel M_n\langle P\rangle}$$

Soundness and completeness of this rule follow from [19]. Note that this rule is similar to the rule ASYM applied recursively for $n+1$ components, where the first and the last component coincide (hence the term "circular"). Learning based assume-guarantee reasoning proceeds as described in Sect. 3.3.

4 Learning with symmetric rules

Although sound and complete, the rules presented in the previous section are not always satisfactory since they are not symmetric in the use of the components. In [6] we proposed a set of symmetric rules that are sound and complete and we also described their automation using learning. They are symmetric in the sense that they are based on establishing and discharging assumptions for each component at the same time.

4.1 Symmetric assume-guarantee rules

Here we present one of the rules that we found particularly effective in practice. The rule may be used for reasoning about a system composed of $n \geq 2$ components: $M_1 \parallel M_2 \parallel \cdots \parallel M_n$.

Rule SYM-N

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

We require $\alpha P \subseteq \alpha M_1 \cup \alpha M_2 \cup \cdots \cup \alpha M_n$ and that for $i \in \{1, 2, \ldots n\}$

$$\alpha A_i \subseteq (\alpha M_1 \cap \alpha M_2 \cap \cdots \cap \alpha M_n) \cup \alpha P.$$

Informally, each A_i is a postulated environment assumption for the component M_i to achieve to satisfy property P. Recall that coA_i is the complement of A_i.

Theorem 8 *Rule* SYM-N *is sound and complete.*

Proof To establish soundness, we show that the premises together with the negated conclusion lead to a contradiction. Consider a trace t for which the conclusion fails, *i.e.*, t is a trace of $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ that violates property P, in other words t is not accepted by P. By the definition of parallel composition, $t{\upharpoonright}\alpha M_1$ is accepted by M_1. Hence, by premise 1, the trace $t{\upharpoonright}\alpha A_1$ can not be accepted by A_1, *i.e.*, $t{\upharpoonright}\alpha A_1$ is accepted by coA_1. Similarly, by premise $i = 2 \ldots n$, the trace $t{\upharpoonright}\alpha A_i$ is accepted by coA_i. By the definition of parallel composition and the fact that an FSM and its complement have the same alphabet, $t{\upharpoonright}(\alpha A_1 \cup A_2 \cup \cdots \cup A_n)$ is accepted by $coA_1 \parallel coA_2 \parallel \cdots \parallel coA_n$ and it violates P. But premise $n + 1$ states that the common traces in the complements of the assumptions belong to the language of P. Hence we have a contradiction.

Our argument for the completeness of Rule SYM-N relies on weakest assumptions. To establish completeness, we assume the conclusion of the rule and show that we can construct assumptions that will satisfy the premises of the rule. We construct the weakest assumptions A_{w1}, $A_{w2}, \ldots A_{wn}$ for M_1, $M_2, \ldots M_n$, respectively, to achieve P and substitute them for A_1, $A_2, \ldots A_n$. Premises 1 through n are satisfied. It remains to show that premise $n + 1$ holds. Again we proceed by contradiction. Suppose there is a trace t in $\mathcal{L}(coA_{w1} \parallel coA_{w2} \parallel \cdots \parallel coA_{wn})$ that violates P; more precisely $t{\upharpoonright}\alpha P \in \mathcal{L}(coP)$. By definition of parallel composition, t is accepted by all $coA_{w1}, coA_{w2}, \ldots coA_{wn}$. Furthermore, there will exist $t_1 \in \mathcal{L}(M_1 \parallel coP)$ such that $t_1{\upharpoonright}\alpha t = t$, where αt is the alphabet of the assumptions. Similarly for $i = 2 \ldots n$, $t_i \in \mathcal{L}(M_i \parallel coP)$. t_1, $t_2, \ldots t_n$ can be combined into trace t' of $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ such that $t'{\upharpoonright}\alpha t = t$. This contradicts the assumed conclusion that $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ satisfies P, since t violates P. Therefore, there can not be such a common trace t, and premise $n + 1$ holds. □

4.2 Learning framework for rule SYM-N

The framework for rule SYM-N is illustrated in Fig. 10. To obtain appropriate assumptions, the framework applies the compositional rule in an iterative fashion. At each iteration L* is used to generate appropriate assumptions for each component, based on querying the system and on the results of the previous iteration. Each assumption is then checked to establish the premises of Rule SYM-N. We use separate instances of L* to iteratively learn $A_{w1}, A_{w2}, \ldots A_{wn}$.

4.2.1 The Teacher

As before, we use model checking to implement the Teacher needed by L*. The conjectures returned by L* are the intermediate assumptions $A_1, A_2, \ldots, A_n$. The Teacher implements $n + 1$ oracles, one for each premise in the SYM-N rule:

- *Oracles* $1, 2, \ldots n$ guide the corresponding L* instances towards conjectures that make the corresponding premise of rule SYM-N true. Once this is accomplished,

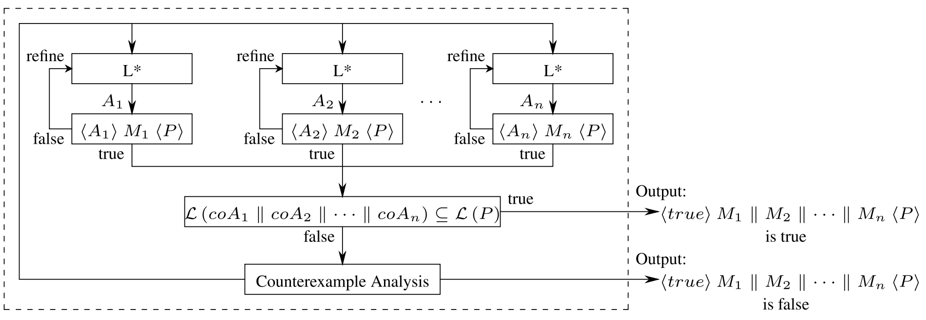

refine
L*
A_1
$\langle A_1\rangle\ M_1\ \langle P\rangle$
false
true
refine
L*
A_2
$\langle A_2\rangle\ M_2\ \langle P\rangle$
false
true
refine
L*
A_n
$\langle A_n\rangle\ M_n\ \langle P\rangle$
false
true
$\mathcal{L}\,(coA_1 \parallel coA_2 \parallel \cdots \parallel coA_n) \subseteq \mathcal{L}\,(P)$
true
false
Counterexample Analysis
Output: $\langle true\rangle\ M_1 \parallel M_2 \parallel \cdots \parallel M_n\ \langle P\rangle$ is true
Output: $\langle true\rangle\ M_1 \parallel M_2 \parallel \cdots \parallel M_n\ \langle P\rangle$ is false

Fig. 10 Learning framework for rule SYM-N

- *Oracle* $n+1$ is invoked to check the last premise of the rule, *i.e.*,

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

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

If the result of *Oracle* $n+1$ is false (with counterexample trace t), by counterexample analysis we identify either that P is indeed violated in $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ or that some of the candidate assumptions need to be modified. If (some of the) assumptions need to be refined in the next iteration, then behaviors must be added to those assumptions. The result will be that at least the behavior that the counterexample represents will be allowed by those assumptions during the next iteration. The new assumptions may of course be too abstract, and therefore the entire process must be repeated.

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

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

4.2.2 Correctness and termination

Theorem 9 *Given components $M_1, M_2, \ldots M_n$ and property P, the algorithm implemented by our framework for rule* SYM-N *terminates and correctly reports on whether P holds on $M_1 \parallel M_2 \parallel \cdots \parallel M_n$.*

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

Termination: At any iteration, the Teacher reports on whether or not P holds on $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ and terminates, or continues by providing a counterexample to L*. By the correctness of L*, we are guaranteed that if it keeps receiving counterexamples, it eventually produces $A_{w1}, A_{w2}, \ldots A_{wn}$, respectively.

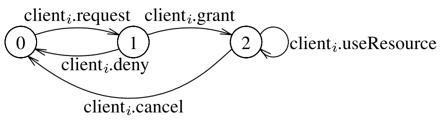


Fig. 11 Example LTS for a client

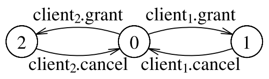


Fig. 12 Mutual exclusion property

During this last iteration, premises 1 through n will hold by definition of the weakest assumptions. The Teacher therefore checks premise $n+1$, which either returns true and terminates, or returns a counterexample. Since the weakest assumptions are used, by the completeness of the rule, we know that the counterexample analysis reveals a real error, and hence the process terminates. □

5 Learning with alphabet refinement

In this section, we present a technique that extends the learning based assume-guarantee reasoning framework with alphabet refinement. We first illustrate the benefits of smaller interface alphabets for assume-guarantee reasoning through a simple client-server example from [27]. Then, we explain the effect of smaller interface alphabets on learning assumptions. We then describe the alphabet refinement algorithm, give its properties, and discuss how it extends to reasoning about n components as well as to circular and symmetric rules.

5.1 Example

Consider a system consisting of a *server* component and two identical *client* components that communicate through shared actions. Each client sends *requests* for reservations to use a common resource, waits for the server to *grant* the reservation, uses the resource, and then *cancels* the reservation. For example, the LTS of a client is shown in Fig. 11, where $i = 1, 2$. The server, shown in Fig. 13 can *grant* or *deny* a request, ensuring that the resource is used only by one client at a time. We are interested in checking the mutual exclusion property illustrated in Fig. 12, that captures a desired behavior of the client-server application.

To check the property compositionally, assume that we decompose the system as: $M_1 = \mathit{Client}_1 \parallel \mathit{Client}_2$ and $M_2 = \mathit{Server}$. The *complete* alphabet of the interface between $M_1 \parallel P$ and M_2 (see Fig. 14) is: $\Sigma_I = \{$client$_1$.cancel, client$_1$.grant, client$_1$.deny, client$_1$.request, client$_2$.cancel, client$_2$.grant, client$_2$.deny, client$_2$.request$\}$.

Using this alphabet and the learning framework in Sect. 3, an assumption with eight states is learned, shown in Fig. 16. However, a (much) smaller assumption is sufficient for proving the mutual exclusion property. With the assumption alphabet $\Sigma = \{$client$_1$.cancel, client$_1$.grant, client$_2$.cancel, client$_2$.grant$\}$, which is a strict subset of Σ_I (and, in fact, the alphabet of the property), a three-state assumption is learned, shown in Fig. 15. This smaller assumption enables more efficient verification than the eight state assumption obtained with the complete alphabet. In the following section, we present an extension of the learning framework that infers automatically smaller interface alphabets (and the corresponding assumptions).

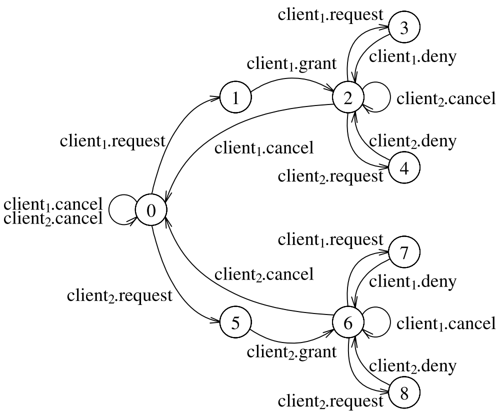


Fig. 13 Example LTS for a server

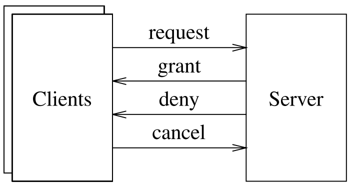


Fig. 14 Complete interface for the client-server example

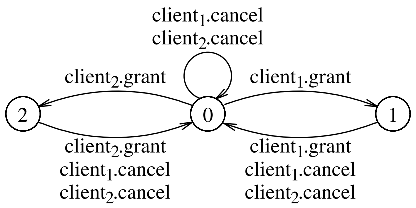
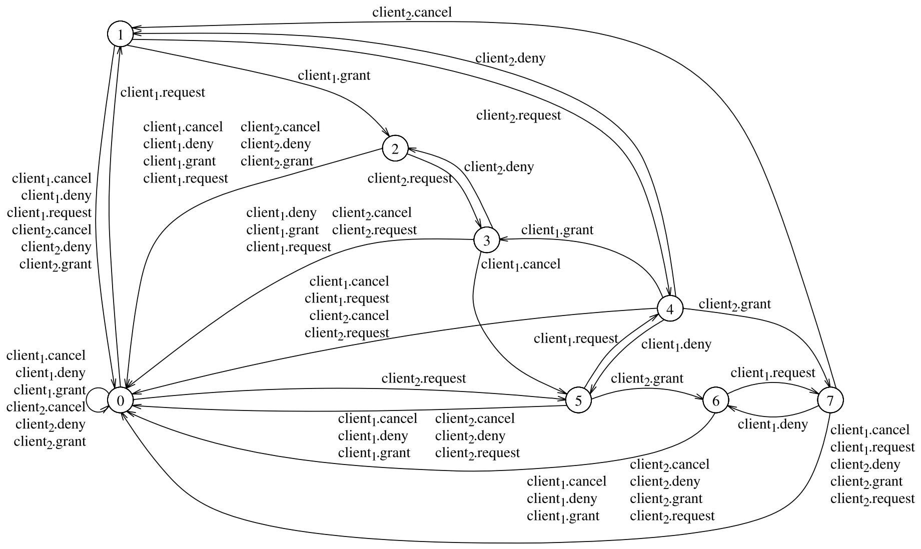


Fig. 15 Assumption learned with an alphabet smaller than the complete interface alphabet

5.2 Learning based assume-guarantee reasoning and small interface alphabets

Before describing the alphabet refinement algorithm, let us first consider the effect of smaller interface alphabets on our learning framework. Let M_1 and M_2 be components, P be a property, Σ_I be the interface alphabet, and Σ be an alphabet such that $\Sigma \subset \Sigma_I$. Suppose that we use the learning framework of Sect. 3 but we now set this smaller Σ to be the alphabet that the framework uses when learning the assumption. From the correctness of the assume-guarantee rule, if the framework reports true, $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$. When it reports false, it is because it finds a trace t in M_2 that falsifies $\langle t \restriction \Sigma \rangle M_1 \langle P \rangle$. This, however, does not necessarily mean that $M_1 \parallel M_2$ violates P. Real violations are discovered by our original framework only when the alphabet is Σ_I, and are traces t' of M_2 that falsify $\langle t' \restriction \Sigma_I \rangle M_1 \langle P \rangle$. In the assume-guarantee triples, $t \restriction \Sigma$ and $t' \restriction \Sigma_I$ are trace LTSs with alphabets Σ and Σ_I, respectively.

Consider again the client-server example. Assume $\Sigma = \{\text{client}_1.\text{cancel}, \text{client}_1.\text{grant}, \text{client}_2.\text{grant}\}$, which is a strict subset of Σ_I. Learning with Σ produces trace: $t = \langle \text{client}_2.\text{request}, \text{client}_2.\text{grant}, \text{client}_2.\text{cancel}, \text{client}_1.\text{request}, \text{client}_1.\text{grant} \rangle$. Projected to Σ, this becomes $t \restriction \Sigma = \langle \text{client}_2.\text{grant}, \text{client}_1.\text{grant} \rangle$. In the context of $t \restriction \Sigma$, M_1 violates the



client2.cancel
client2.deny
client1.grant
client1.request
client2.request
client1.cancel client1.deny client1.grant client1.request
client2.cancel client2.deny client2.grant
client1.cancel client1.deny client1.request client2.cancel client2.deny client2.grant
client2.request
client2.deny
client1.deny client1.grant client1.request
client2.cancel client2.request
client1.grant
client1.cancel
client1.cancel client1.request client2.cancel client2.request
client2.grant
client1.request
client1.deny
client1.cancel client1.deny client1.grant client2.cancel client2.deny client2.grant
client2.request
client2.grant
client1.request
client1.deny
client1.cancel client1.deny client1.grant
client2.cancel client2.deny client2.request
client2.cancel client2.deny client2.grant client2.request
client1.cancel client1.deny client1.grant
client1.cancel client1.request client2.deny client2.grant client2.request

Fig. 16 Assumption obtained with the complete interface alphabet

property since $Client_1 \parallel Client_2 \parallel P_{err}$ contains the following behavior.

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

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

5.3 Algorithm for alphabet refinement

Alphabet refinement extends the learning framework to deal with alphabets that are smaller than Σ_I while avoiding spurious counterexamples. The steps of the algorithm are as follows (see Fig. 17):

(1) **Initialize** Σ such that $\Sigma \subseteq \Sigma_I$.
(2) Use the classic learning framework for Σ. If the framework returns true, then report true and STOP. If the framework returns false with counterexamples c and t, go to the next step.
(3) Perform **extended counterexample analysis** with c and t. If c is a real counterexample, then report false and STOP. If c is spurious, then **refine** Σ, which consists of adding actions to Σ from Σ_I. Go to step 2.

When spurious counterexamples are detected, the Refiner extends the alphabet with actions from the alphabet of the weakest assumption and the learning of assumptions is restarted. In the worst case, Σ_I is reached and, as proven in our previous work, learning then only

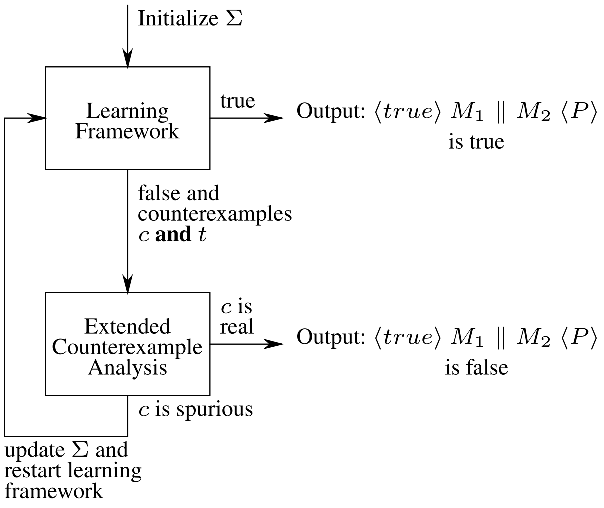


Fig. 17 Learning with alphabet refinement
Initialize Σ
Learning Framework
true
Output: $\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$ is true
false and counterexamples c **and** t
Extended Counterexample Analysis
c is real
Output: $\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$ is false
c is spurious
update Σ and restart learning framework

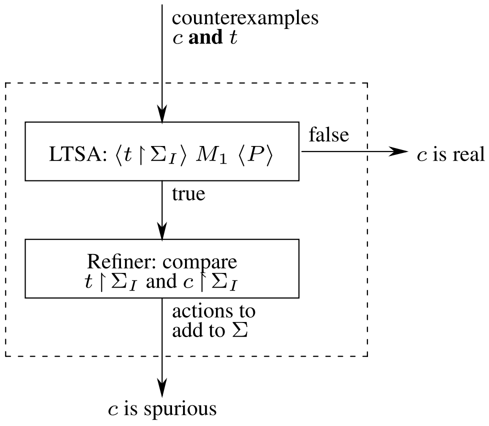


Fig. 18 Extended counterexample analysis
counterexamples c **and** t
LTSA: $\langle t \upharpoonright \Sigma_I \rangle\ M_1\ \langle P \rangle$
false
c is real
true
Refiner: compare $t \upharpoonright \Sigma_I$ and $c \upharpoonright \Sigma_I$
actions to add to Σ
c is spurious

reports real counterexamples. The highlighted steps in the above high-level algorithm are further specified next.

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 Σ_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. The steps of this analysis are outlined in Fig. 18. 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 (Fig. 4) to return both c **and** t to be used in alphabet refinement (as explained below). As discussed, c is obtained because $\langle t \upharpoonright \Sigma \rangle M_1 \langle P \rangle$

does not hold. The next step is to check whether in fact t uncovers a real violation in the system. As illustrated by the client-server example, the results of checking $M_1 \parallel P_{err}$ in the context of t projected to different alphabets may be different. The correct (non-spurious) results are obtained by projecting t on the alphabet Σ_I of the weakest assumption. Counterexample analysis therefore calls LTSA to check $\langle t\lceil\Sigma_I\rangle M_1\langle P\rangle$. If LTSA finds an error, the resulting counterexample c is real. If error is not reached, then the counterexample is spurious and the alphabet Σ needs to be refined. Refinement proceeds as described next.

Alphabet refinement When spurious counterexamples are detected, we need to augment the current alphabet Σ so that these counterexamples are eventually eliminated. A counterexample c is spurious if in the context of $t\lceil\Sigma_I$ it would not be obtained. Our refinement heuristics are therefore based on comparing c and $t\lceil\Sigma_I$ to discover actions in Σ_I to be added to the learning alphabet (for this reason c is also projected on Σ_I in the refinement process). We have currently implemented the following heuristics:

AllDiff: adds all the actions in the symmetric difference of $t\lceil\Sigma_I$ and $c\lceil\Sigma_I$. A potential problem of this heuristic is that it may add too many actions too soon. If it happens to add useful actions, however, it may terminate after a small number of iterations.
Forward: scans the traces $t\lceil\Sigma_I$ and $c\lceil\Sigma_I$ in parallel from beginning to end looking for the first index i where they disagree; if such an i is found, both actions $t\lceil\Sigma_I(i), c\lceil\Sigma_I(i)$ are added to the alphabet. By adding fewer actions during each iteration, the algorithm may end up with a smaller alphabet. But, it may take more iterations before it does not produce a spurious result.
Backward: is similar to Forward, but scans from the end of the traces to the beginning.

5.3.1 Correctness and termination

For correctness and termination of learning with alphabet refinement, we first show progress of refinement, meaning that at each refinement stage, new actions are discovered to be added to Σ.

Proposition 10 (Progress of alphabet refinement) *Let $\Sigma_I = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$ be the alphabet of the weakest assumption and let $\Sigma \subset \Sigma_I$ be that of the assumption at the current alphabet refinement stage. Let t be a trace of $M_2 \parallel A_{err}$ such that $t\lceil\Sigma$ leads to error on $M_1 \parallel P_{err}$ by an error trace c, but $t\lceil\Sigma_I$ does not lead to error on $M_1 \parallel P_{err}$. Then $t\lceil\Sigma_I \neq c\lceil\Sigma_I$ and there exists an action in their symmetric difference that is not in Σ.*

Proof We prove by contradiction that $t\lceil\Sigma_I \neq c\lceil\Sigma_I$. Suppose $t\lceil\Sigma_I = c\lceil\Sigma_I$. We know that c is an error trace on $M_1 \parallel P_{err}$. Since actions of c that are not in Σ_I are internal to $M_1 \parallel P_{err}$, then $c\lceil\Sigma_I$ also leads to error on $M_1 \parallel P_{err}$. But then $t\lceil\Sigma_I$ leads to error on $M_1 \parallel P_{err}$, which is a contradiction.

We now show that there exists an action in the symmetric difference between $t\lceil\Sigma_I$ and $c\lceil\Sigma_I$ that is not in Σ (this action will be added to Σ by alphabet refinement). Trace $t\lceil\Sigma_I$ is $t\lceil\Sigma$, with some interleaved actions from $\Sigma_I \setminus \Sigma$. Similarly, $c\lceil\Sigma_I$ is $t\lceil\Sigma$ with some interleaved actions from $\Sigma_I \setminus \Sigma$, since c is obtained by composing the trace LTS $t\lceil\Sigma$ with $M_1 \parallel P_{err}$. Thus $t\lceil\Sigma = c\lceil\Sigma$. We again proceed by contradiction. If all the actions in the symmetric difference between $t\lceil\Sigma_I$ and $c\lceil\Sigma_I$ were in Σ, we would have $t\lceil\Sigma_I = t\lceil\Sigma = c\lceil\Sigma = c\lceil\Sigma_I$, which contradicts $t\lceil\Sigma_I \neq c\lceil\Sigma_I$. □

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

Theorem 11 *Given components M_1 and M_2, and property P, L* with alphabet refinement terminates and returns true if $M_1 \parallel M_2$ satisfies P and false otherwise.*

Proof Correctness: When the teacher returns true, then correctness is guaranteed by the assume-guarantee compositional rule. If the teacher returns false, the extended counterexample analysis reports an error for a trace t of M_2, such that $t\lceil\Sigma_I$ in the context of M_1 violates the property (the same test is used in the algorithm from [13]) hence $M_1 \parallel M_2$ violates the property.

Termination: From the correctness of L*, we know that at each refinement stage (with alphabet Σ), if L* keeps receiving counterexamples, it is guaranteed to generate $A_{w,\Sigma}$. At that point, Oracle 1 will return true (from Lemma 5). Therefore, Oracle 2 will be applied, which will return either true, and terminate, or a counterexample t. This counterexample is a trace that is not in $\mathcal{L}(A_{w,\Sigma})$. It is either a real counterexample (in which case the algorithm terminates) or it is a trace t such that $t\lceil\Sigma$ leads to error on $M_1 \parallel P_{err}$ by an error trace c, but $t\lceil\Sigma_I$ does not lead to error on $M_1 \parallel P_{err}$. Then from Proposition 10, we know that $t\lceil\Sigma_I \neq c\lceil\Sigma_I$ and there exists an action in their symmetric difference that is not in Σ. The Refiner will add this action (and possibly more actions, depending on the refinement strategy) to Σ and the learning algorithm is repeated for this new alphabet. Since Σ_I is finite, in the worst case, Σ grows into Σ_I, for which termination and correctness follow from Theorem 7. $\square$

We also note a property of weakest assumptions, which states that by adding actions to an alphabet Σ, the corresponding weakest assumption becomes *weaker* (*i.e.*, contains more behaviors) than the previous one.

Proposition 12 *Assume components M_1 and M_2, property P and the corresponding interface alphabet Σ_I. Let Σ, Σ' be sets of actions such that: $\Sigma \subset \Sigma' \subset \Sigma_I$. Then: $\mathcal{L}(A_{w,\Sigma}) \subseteq \mathcal{L}(A_{w,\Sigma'}) \subseteq \mathcal{L}(A_{w,\Sigma_I})$.*

Proof Since $\Sigma \subseteq \Sigma'$, we know that $A_{w,\Sigma}\lceil\Sigma' = A_{w,\Sigma}$. By substituting, in Definition 2, $A_{w,\Sigma}$ for M_2, we obtain that: $\langle A_{w,\Sigma}\rangle M_1 \langle P\rangle$ if and only if $\langle true\rangle A_{w,\Sigma} \langle A_{w,\Sigma'}\rangle$. From Lemma 5 we know that $\langle A_{w,\Sigma}\rangle M_1 \langle P\rangle$. Therefore, $\langle true\rangle A_{w,\Sigma}\langle A_{w,\Sigma'}\rangle$ holds, which implies that $\mathcal{L}(A_{w,\Sigma}) \subseteq \mathcal{L}(A_{w,\Sigma'})$. Similarly, $\mathcal{L}(A_{w,\Sigma'}) \subseteq \mathcal{L}(A_{w,\Sigma_I})$. $\square$

With alphabet refinement, our framework adds actions to the alphabet, which translates into adding more behaviors to the weakest assumption that L* tries to learn. This means that at each refinement stage i, when the learning framework is started with a new alphabet Σ_i such that $\Sigma_{i-1} \subset \Sigma_i$, it will try to learn a weaker assumption A_{w,Σ_i} than $A_{w,\Sigma_{i-1}}$, which was its goal in the previous stage. Moreover, all these assumptions are *under-approximations* of the weakest assumption A_{w,Σ_I} that is necessary and sufficient to prove the desired property. Note that at each refinement stage the learning framework might stop before computing the corresponding weakest assumption. The above property allows reuse of learning results across refinement stages (see Sect. 8).

5.4 Generalization to n components

Alphabet refinement can also be used when reasoning about more than two components using rule ASYM. Recall from Sect. 3 that to check if system $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ satisfies P we decompose it into: M_1 and $M_2' = M_2 \parallel M_3 \parallel \cdots \parallel M_n$ and the learning algorithm (without refinement) is invoked recursively for checking the second premise of the assume-guarantee rule.

Learning with alphabet refinement follows this recursion. At each recursive invocation for M_j and $M_j' = M_{j+1} \parallel M_{j+2} \parallel \cdots \parallel M_n$, we solve the following problem: find assumption A_j and alphabet Σ_{A_j} such that the rule premises hold, *i.e.*

Oracle 1: $\langle A_j \rangle M_j \langle A_{j-1} \rangle$ and
Oracle 2: $\langle \mathit{true} \rangle M_{j+1} \parallel M_{j+2} \parallel \cdots \parallel M_n \langle A_j \rangle$.

Here A_{j-1} is the assumption for M_{j-1} and plays the role of the property for the current recursive call. Thus, the alphabet of the weakest assumption for this recursive invocation is $\Sigma_I^j = (\alpha M_j \cup \alpha A_{j-1}) \cap (\alpha M_{j+1} \cup \alpha M_{j+2} \cup \cdots \cup \alpha M_n)$. If Oracle 2 returns a counterexample, then the counterexample analysis and alphabet refinement proceed exactly as in the two-component case. Note that at a new recursive call for M_j with a new A_{j-1}, the alphabet of the weakest assumption is recomputed.

Correctness and termination of this extension follow from Theorem 11 (and from finiteness of n). The proof proceeds by induction on n.

5.5 Extension to circular and symmetric rules

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

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

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

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

6 Experiments

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

Models and properties We used the following LTSA models. *Gas Station* [20] models a self-serve gas station consisting of k customers, two pumps, and an operator. For $k = 3, 4, 5$, we checked that the operator correctly gives change to a customer for the pump that he/she used. *Chiron* [5, 23] models a graphical user interface consisting of k artists, a wrapper, a manager, a client initialization module, a dispatcher, and two event dispatchers. For $k = 2 \dots 5$, we checked two properties: the dispatcher notifies artists of an event before receiving a next event, and the dispatcher only notifies artists of an event after it receives that event. *MER* [27] models the flight software component for JPL's Mars Exploration Rovers. It contains k users competing for resources managed by an arbiter. For $k = 2 \dots 6$, we checked that communication and driving cannot happen at the same time as they share common resources. *Rover Executive* [13] models a subsystem of the Ames K9 Rover. The models consists of a main 'Executive' and an 'ExecCondChecker' component responsible for monitoring state conditions. We checked that for a specific shared variable, if the Executive reads its value, then the ExecCondChecker should not read it before the Executive clears it.

Gas Station and Chiron were analyzed before, in [14], using learning-based assume-guarantee reasoning (with ASym and no alphabet refinement). Four properties of Gas Station and nine properties of Chiron were checked to study how various 2-way model decompositions (*i.e.*, grouping the modules of each analyzed system into two "super-components") affect the performance of learning. For most of these properties, learning performs better than non-compositional verification and produces small (one-state) assumptions. For some other properties, learning does not perform that well, and produces much larger assumptions. To stress-test our implementation, we selected some of the latter, more challenging properties for our studies here.

Results We performed several sets of experiments. All experiments were performed on a Dell PC with a 2.8 GHz Intel Pentium 4 CPU and 1.0 GB RAM, running Linux Fedora Core 4 and using Sun's Java SDK version 1.5. The results are shown in Tables 3, 4, 5, and 6. In the tables, $|A|$ is the *maximum* assumption size reached during learning, 'Mem.' is the *maximum* memory used by LTSA to check assume-guarantee triples, measured in MB, and 'Time' is the total CPU running time, measured in seconds. Column 'Monolithic' reports the memory and run-time of non-compositional model checking. We set a limit of 30 minutes for each run. The sign '–' indicates that the limit of 1 GB of memory or the time limit has been exceeded. For these cases, the data is reported as it was when the limit was reached.

In Table 3, we show the performance of learning with the ASym rule, without alphabet refinement, and with different alphabet refinement heuristics, for two-way decompositions of the systems we studied. For Gas Station and Chiron we used decompositions generalized from the best two-way decompositions at size 2, as described in [14]. For Gas Station, the operator and the first pump are one component, and the rest of the modules are the other. For Chiron, the event dispatchers are one component, and the rest of the modules are the other. For MER, half of the users are in one component, and the other half with the arbiter in the other. For the Rover we used the two components described in [13]. As these results indicate that 'bwd' heuristic is slightly better than the others, we used this heuristic for alphabet refinement in the rest of the experiments.

Table 4 shows the performance of the recursive implementation of learning with rule ASym, with and without alphabet refinement, as well as that of monolithic (non-compositional) verification, for increasing number of components. For these experiments

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,	2	9	1.30	1.23	8	1.22	3.53	8	1.22	1.86	8	1.22	1.90
Property 1	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,	2	9	1.14	1.57	3	1.05	0.73	3	1.05	0.73	3	1.05	0.74
Property 2	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

Table 4 Comparison of recursive learning for ASYM rule with and without alphabet refinement, and monolithic verification

Case	k	ASYM			ASYM + ref			Monolithic	
		$\lvert A\rvert$	Mem.	Time	$\lvert A\rvert$	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,	2	352	343.62	–	4	0.93	2.38	0.88	0.041
Property 1	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,	2	190	107.45	–	11	1.68	40.11	1.21	0.035
Property 2	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

Table 5 Comparison of learning for SYM-N rule with and without alphabet refinement

Case	k	SYM-N			SYM-N + ref		
		$\lvert A\rvert$	Mem.	Time	$\lvert A\rvert$	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,	2	19	2.21	–	21	4.56	52.14
Property 1	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,	2	7	1.16	–	8	0.93	6.35
Property 2	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

Table 6 Comparison of learning for CIRC-N rule with and without alphabet refinement

Case	k	CIRC-N			CIRC-N + 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,	2	259	78.03	–	4	0.96	2.71
Property 1	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,	2	67	100.91	–	327	44.17	–
Property 2	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

we used an additional heuristic to compute the *ordering* of the modules in the sequence $M_1, \ldots M_n$ for the recursive learning, to minimize the sizes of the interface alphabets $\Sigma_I^1, \ldots \Sigma_I^n$. We generated offline all possible orders with their associated interface alphabets and then chose the order that minimizes the sum $\sum_{j=1}^{n} |\Sigma_I^j|$. Automatic generation of orderings was not always possible because of the combinatorial explosion. In some cases with large parameter n, we lifted the results obtained for small values of the parameter on the same model to the model with the larger parameter.

We also compared learning with and without alphabet refinement for rules SYM-N and CIRC-N under the same conditions as in the previous experiments. The results are in Tables 5 and 6.

Discussion The results overall show that rule ASYM is more effective than the other rules and that alphabet refinement improves learning significantly.

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

Table 3 shows that alphabet refinement improved the assumption size in all cases, and in a few, up to almost two orders of magnitude (see Gas Station with $k = 3, 4$, Chiron, Property 2, with $k = 5$, MER with $k = 3$). It improved memory consumption in 10 out of 15 cases, and also improved running time, as for Gas Station and for MER with $k = 3, 4$ learning without refinement did not finish within the time limit, whereas with refinement it did. The benefit of alphabet refinement is even more obvious in Table 4 where 'No refinement' exceeded the time limit in all but one case, whereas refinement completed in almost all cases, producing smaller assumptions, and using less memory in all the cases, up to two orders of magnitude less in a few.

Table 4 indicates that learning with refinement scales better than without refinement for increasing number of components. As k increases, the memory and time consumption for 'Refinement' grows slower than that of 'Monolithic'. For Gas Station, Chiron (Property 1), and MER, for small values of k, 'Refinement' consumes more memory than 'Monolithic', but as k increases the gap is narrowing, and for the largest k 'Refinement' becomes better than 'Monolithic'. This leads to cases such as MER with $k = 6$ where, for a large enough parameter value, 'Monolithic' runs out of memory, whereas 'Refinement' succeeds.

7 Related work

Several frameworks have been proposed to support assume-guarantee reasoning [10, 19, 22, 28]. For example, the Calvin tool [16] uses assume-guarantee reasoning for the analysis of Java programs, while Mocha [1] supports modular verification of components with requirements specified based in the Alternating-time Temporal Logic. The practical impact of these approaches has been limited because they require non-trivial human input in defining appropriate assumptions.

Our previous work [13, 18] proposed to use L* to automate assume-guarantee reasoning. Since then, several other frameworks that use L* for learning assumptions have been developed; [3] presents a symbolic BDD implementation using NuSMV [9]. This symbolic version was extended in [26] with algorithms that decompose models using hypergraph partitioning, to optimize the performance of learning on resulting decompositions. Different decompositions are also studied in [14] where the best two-way decompositions are computed for model-checking with the FLAVERS [15] and LTSA tools. L* has also been used in [2] to synthesize interfaces for Java classes, and in [30] to check component compatibility after component updates.

Our approach for alphabet refinement is similar in spirit to counterexample-guided abstraction refinement (CEGAR) [11]. CEGAR computes and analyzes abstractions of programs (usually using a set of abstraction predicates) and refines them based on spurious counter-examples. However, there are some important differences between CEGAR and our algorithm. Alphabet refinement works on actions rather than predicates, it is applied compositionally in an assume-guarantee style and it computes under-approximations (of assumptions) rather than behavioral over-approximations (as it happens in CEGAR). In the future, we plan to investigate more the relationship between CEGAR and our algorithm. The work of [21] proposes a CEGAR approach to interface synthesis for C libraries. This work does not use learning, nor does it address the use of the resulting interfaces in assume-guarantee verification.

A similar idea to our alphabet refinement for L* in the context of assume-guarantee verification has been developed independently in [7]. In that work, L* is started with an empty alphabet, and, similar to our approach, the assumption alphabet is refined when a spurious counterexample is obtained. At each refinement stage, a new minimal alphabet is computed that eliminates all spurious counterexamples seen so far. The computation of such a minimal alphabet is shown to be NP-hard. In contrast, we use much cheaper heuristics, but do not guarantee that the computed alphabet is minimal. The approach presented in [31] improves upon assume-guarantee learning for systems that communicate based on shared memory, by using SAT based model checking and alphabet clustering.

The theoretical results in [25] show that circular assume-guarantee rules can not be both sound and complete. These results do not apply to rules such as ours that involve additional assumptions which appear only in the premises and not in the conclusions of the rules. Note that completeness is not required by our framework (however incompleteness may lead to inconclusive results).

8 Conclusions and future work

We have introduced a framework that uses a learning algorithm to synthesize assumptions that automate assume-guarantee reasoning for finite-state machines and safety properties. The framework incorporates symmetric, asymmetric and circular assume-guarantee rules and uses alphabet refinement to compute small assumption alphabets that are sufficient for verification. The framework has been applied to a variety of systems where it showed its effectiveness.

In future work we plan to look beyond checking safety properties and to address further algorithmic optimizations, *e.g.*, reuse of query results and learning tables across alphabet refinement stages. Moreover, we plan to explore techniques alternative to learning for computing assumptions, *e.g.*, we are investigating abstraction refinement techniques for computing assumptions incrementally as abstractions of environments. Finally we plan to perform more experiments to further evaluate our framework.

References

1. Alur R, Henzinger T, Mang F, Qadeer S, Rajamani S, Tasiran S (1998) MOCHA: modularity in model checking. In: Proceedings of CAV'98. LNCS, vol 1427. Springer, New York, pp 521–525
2. Alur R, Cerny P, Madhusudan P, Nam W (2005) Synthesis of interface specifications for Java classes. In: Proceedings of POPL'05, pp 98–109
3. Alur R, Madhusudan P, Nam W (2005) Symbolic compositional verification by learning assumptions. In: Proceedings of CAV'05. LNCS, vol 3576. Springer, New York, pp 548–562
4. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
5. Avrunin GS, Corbett JC, Dwyer MB, Păsăreanu CS, Siegel SF (1999) Comparing finite-state verification techniques for concurrent software. TR 99-69, University of Massachusetts, Department of Computer Science, November
6. Barringer H, Giannakopoulou D, Păsăreanu CS (2003) Proof rules for automated compositional verification through learning. In: Proceedings of SAVCBS'03, pp 14–21
7. Chaki S, Strichman O (2007) Optimized L*-based assume-guarantee reasoning. In: Proceedings of TACAS'07. LNCS, vol 4424. Springer, New York, pp 276–291
8. Cheung SC, Kramer J (1999) Checking safety properties using compositional reachability analysis. ACM Trans Softw Eng Methodol 8(1):49–78
9. Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV 2: an opensource tool for symbolic model checking. In: Proceedings of CAV'02. LNCS, vol 2404. Springer, New York, pp 359–364
10. Clarke EM, Long DE, McMillan KL (1989) Compositional model checking. In: Proceedings of LICS'89, pp 353–362
11. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Proceedings of CAV'00. LNCS, vol 1855. Springer, New York, pp 154–169
12. Clarke EM, Grumberg O, Peled D (2000) Model checking. MIT Press, Cambridge
13. Cobleigh JM, Giannakopoulou D, Păsăreanu CS (2003) Learning assumptions for compositional verification. In: Proceedings of TACAS'03. LNCS, vol 2619. Springer, New York, pp 331–346
14. Cobleigh JM, Avrunin GS, Clarke LA (2006) Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In: Proceedings of ISSTA'06. ACM, New York, pp 97–108
15. Dwyer MB, Clarke LA, Cobleigh JM, Naumovich G (2004) Flow analysis for verifying properties of concurrent software systems. ACM Trans Softw Eng Methodol 13(4):359–430
16. Flanagan C, Freund SN, Qadeer S (2002) Thread-modular verification for shared-memory programs. In: Proceedings of ESOP'02, pp 262–277

17. Gheorghiu M, Giannakopoulou D, Păsăreanu CS (2007) Refining interface alphabets for compositional verification. In: Proceedings of TACAS'07. LNCS, vol 4424. Springer, New York, pp 292–307
18. Giannakopoulou D, Păsăreanu CS, Barringer H (2005) Component verification with automatically generated assumptions. Autom Softw Eng 12(3):297–320
19. Grumberg O, Long DE (1991) Model checking and modular verification. In: Proceedings of CONCUR'91, pp 250–265
20. Helmbold D, Luckham D (1985) Debugging Ada tasking programs. IEEE Softw 2(2):47–57
21. Henzinger TA, Jhala R, Majumdar R (2005) Permissive interfaces. In: Proceedings of ESEC/SIGSOFT FSE'05, pp 31–40
22. Jones CB (1983) Specification and design of (parallel) programs. In: Information processing 83: proceedings of the IFIP 9th world congress. IFIP: North-Holland, Amsterdam, pp 321–332
23. Keller RK, Cameron M, Taylor RN, Troup DB (May 1991) User interface development and software environments: the Chiron-1 system. In: Proceedings of ICSE'91, pp 208–218
24. Magee J, Kramer J (1999) Concurrency: state models & Java programs. Wiley, New York
25. Maier P (2003) Compositional circular assume-guarantee rules cannot be sound and complete. In: Proceedings of FOSSACS
26. Nam W, Alur R (2006) Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In: Proceedings of ATVA'06. LNCS, vol 4218. Springer, New York
27. Păsăreanu CS, Giannakopoulou D (2006) Towards a compositional SPIN. In: Proceedings of SPIN'06. LNCS, vol 3925. Springer, New York, pp 234–251
28. Pnueli A (1984) In transition from global to modular temporal reasoning about programs. Log Models Concurr Syst 13:123–144
29. Rivest RL, Shapire RE (1993) Inference of finite automata using homing sequences. Inf Comput 103(2):299–347
30. Sharygina N, Chaki S, Clarke EM, Sinha N (2005) Dynamic component substitutability analysis. In: Proceedings of FM'05. LNCS, vol 3582. Springer, New York, pp 512–528
31. Sinha N, Clarke EM (2007) SAT-based compositional verification using lazy learning. In: Proceedings of CAV'07. LNCS, vol 4590. Springer, New York, pp 39–54