Refining Interface Alphabets for Compositional Verification

Mihaela Gheorghiu[1], Dimitra Giannakopoulou[2], and Corina S. Păsăreanu[2]

[1] Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada
mg@cs.toronto.edu
[2] RIACS and QSS, NASA Ames Research Center,
Moffett Field, CA 94035, USA
{dimitra,pcorina}@email.arc.nasa.gov

Abstract. Techniques for learning automata have been adapted to automatically infer assumptions in assume-guarantee compositional verification. Learning, in this context, produces assumptions and modifies them using counterexamples obtained by model checking components separately. In this process, the interface alphabets between components, that constitute the alphabets of the assumption automata, are fixed: they include *all* actions through which the components communicate. This paper introduces *alphabet refinement*, a novel technique that extends the assumption learning process to also infer interface alphabets. The technique starts with only a *subset* of the interface alphabet and adds actions to it as necessary until a given property is shown to hold or to be violated in the system. Actions to be added are discovered by counterexample analysis. We show experimentally that alphabet refinement improves the current learning algorithms and makes compositional verification by learning assumptions more scalable than non-compositional verification.

1 Introduction

Model checking is an effective technique for finding subtle errors in concurrent software. Given a finite model of a system and of a required property, model checking determines automatically whether the property is satisfied by the system. The limitation of this approach, known as the "state-explosion" problem [9], is that it needs to explore all the system states, which may be intractable for realistic systems.

Compositional verification addresses state explosion by 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, one needs to incorporate some knowledge of the contexts in which the components are expected to operate correctly. Assume-guarantee reasoning [18,23] addresses this issue by introducing assumptions that capture the expectations of a component from its environment. Assumptions have traditionally been defined manually, which has limited the practical impact of assume-guarantee reasoning.

Recent work [12,5] has proposed a framework based on learning that *fully automates* assume-guarantee model checking of safety properties. Since then, several similar frameworks have been presented [3,21,25]. To check that a system consisting of

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 292–307, 2007.
© Springer-Verlag Berlin Heidelberg 2007

components M_1 and M_2 satisfies a safety property P, the framework automatically guesses and refines assumptions for one of the components to satisfy P, which it then tries to discharge on the other component. The approach is guaranteed to terminate, 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 [7,15]. Interfaces consist of *all* communication points through which the components may influence each other's behavior. In the learning framework of [12] the alphabet of the assumption automata being built includes *all* the actions in the component interface. However, in a case study presented in [22], we observed that a smaller alphabet was sufficient to prove the 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 [22].

Motivated by the successful use of a smaller assumption alphabet in learning, we investigate here whether we can automate the process of discovering a smaller alphabet that is sufficient for checking the desired properties. Smaller alphabet means smaller interface between components, which may lead to smaller assumptions, and hence to smaller verification problems. We propose a novel technique called *alphabet refinement* that extends the learning framework to start with a small subset of the interface alphabet and to add actions into it as necessary until a required property is shown to hold or to be violated in the system. Actions to be added are discovered by analysis of the counterexamples obtained from model checking the components. We study the properties of alphabet refinement and show experimentally that it leads to time and memory savings as compared to the original learning framework [12] and monolithic model checking. The algorithm has been implemented within the LTSA model checking tool [20].

The algorithm is applicable to and may benefit any of the previous learning-based approaches [3,21,25]; it may also benefit other compositional analysis techniques. Compositional Reachability Analysis (CRA), for example, computes abstractions of component behaviors based on their interfaces. In the context of property checking [7,19], smaller interfaces may result in more compact abstractions, leading to smaller state spaces when components are put together.

The rest of the paper is organized as follows. Sec. 3 presents a motivating example. Sec. 4 summarizes the original learning framework from [12]. Sec. 5 presents the main algorithm for interface alphabet refinement. Sec. 6 discusses properties and Sec. 7 provides an experimental evaluation of the proposed algorithm. Sec. 8 surveys some related work and Sec. 9 concludes the paper. In the next section we review the main ingredients of the LTSA tool and the L* learning algorithm.

2 Background

Labeled Transition Systems (LTSs). LTSA is an explicit-state model checker that analyzes finite-state systems modeled as *labeled transition systems* (LTSs). Let $\mathcal{A}$ be the universal set of observable actions and let τ denote a special action that is unobservable.

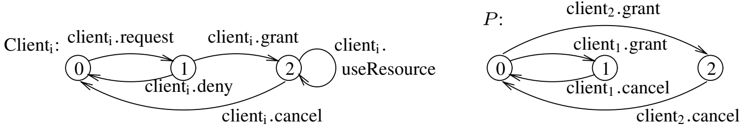

Fig. 1. Example LTS for a client (left) and a mutual exclusion property (right)

An LTS M is a tuple $\langle Q, \alpha M, \delta, q_0 \rangle$, where: Q is a finite non-empty set of states; $\alpha M \subseteq \mathcal{A}$ 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, and q_0 is the initial state. An LTS M is *non-deterministic* if it contains τ-transitions or if $\exists (q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*. We use π to denote a special *error state* that has no outgoing transitions, and Π to denote the LTS $\langle \{\pi\}, \mathcal{A}, \emptyset, \pi \rangle$. 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 $q_0' = \pi$, $M' = \Pi$.

Consider a simple client-server application (from [22]). It consists 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. 1 (left), where $i = 1, 2$. The server can *grant* or *deny* a request, ensuring that the resource is used only by one client at a time (the LTS of the server is shown in [14]).

Parallel Composition. Parallel composition "$\|$" is a commutative and associative operator such that: given LTSs $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$, $M_1 \parallel M_2$ is Π if either one of M_1, M_2 is Π. 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 (the symmetric version also applies):

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

Traces. A *trace* t of an LTS M is a sequence of observable actions starting from the initial state and obeying the transition relation. The set of all traces of M is called the *language* of M, denoted $\mathcal{L}(M)$. Any trace t may also be viewed as an LTS, which we call a *trace LTS*; its language consists of t and its prefixes. We denote by t both a trace and its trace LTS; the meaning should be clear from the context. For $\Sigma \subseteq \mathcal{A}$, we denote by $t{\downarrow}_\Sigma$ the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. Similarly, $M{\downarrow}_\Sigma$ 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 A, A' be the sets of actions occurring in t, t', respectively. By the *symmetric difference* of t and t' we mean the symmetric difference of sets A and A'.

Safety properties. We call a deterministic LTS not containing π a *safety LTS* (any non-deterministic LTS can be made deterministic with the standard algorithm for automata).

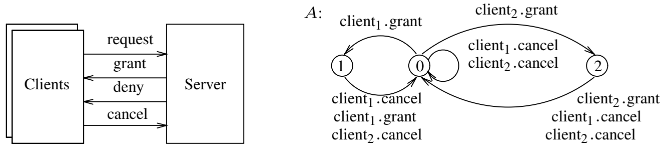

Fig. 2. Client-Server Example: complete interface (left) and derived assumption with alphabet smaller than complete interface alphabet (right).

A safety property P is specified as a *safety LTS* whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP. For example, the mutual exclusion property in Fig. 1 (right) captures the desired behaviour of the client-server application discussed earlier.

An LTS M satisfies P, denoted $M \models P$, iff $\forall \sigma \in M : \sigma{\downarrow_{\alpha P}} \in \mathcal{L}(P)$. For checking a property P, its safety LTS is *completed* by adding error state π and transitions on all the missing outgoing actions from all states into π; the resulting LTS is denoted by P_{err}. LTSA checks $M \models P$ by computing $M \parallel P_{err}$ and checking if π is reachable in the resulting LTS.

Assume-guarantee reasoning. 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. In LTSA, checking $\langle A\rangle M\langle P\rangle$ reduces to checking $A \parallel M \models P$. The simplest assume-guarantee proof rule shows that if $\langle A\rangle M_1\langle P\rangle$ and $\langle \mathit{true}\rangle M_2\langle A\rangle$ hold, then $\langle \mathit{true}\rangle M_1 \parallel M_2\langle P\rangle$ also holds:

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

Coming up with appropriate assumptions used to be a difficult, manual process. Recent work has proposed an off-the-shelf learning algorithm, L*, to derive appropriate assumptions *automatically* [12].

The L* learning algorithm. L* was developed by Angluin [4] and later improved by Rivest and Schapire [24]. L* learns an unknown regular language U over alphabet Σ and produces a deterministic finite state automaton (DFA) that accepts it. L* interacts with a *Minimally Adequate Teacher* that answers two types of questions from L*. The first type is a *membership query* asking whether a string $s \in \Sigma^*$ is in U. For the second type, the learning algorithm generates a *conjecture* A and asks whether $\mathcal{L}(A) = U$. If $\mathcal{L}(A) \neq U$ the Teacher returns a counterexample, which is a string s in the symmetric difference of $\mathcal{L}(A)$ and U. L* is guaranteed to terminate with a minimal automaton A for U. If A has n states, L* makes at most $n-1$ incorrect conjectures. The number of membership queries made by L* is $O(kn^2 + n\log m)$, where k is the size of Σ, n is the number of states in the minimal DFA for U, and m is the length of the longest counterexample returned when a conjecture is made.

3 Assume-Guarantee Reasoning and Small Interface Alphabets

We illustrate the benefits of smaller interface alphabets for assume guarantee reasoning through the client-server example of Sec. 2. To check the property in a compositional way, assume that we break up the system into: M_1 = Client$_1$ $\|$ Client$_2$ and M_2 = Server. The *complete* alphabet of the interface between $M_1 \parallel P$ and M_2 (see Fig. 2 (left)) is: {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 method of [12] yields an assumption with 8 states (see [14]). However, a (much) smaller assumption is sufficient for proving the mutual exclusion property (see Fig. 2 (right)). The assumption alphabet is {client$_1$.cancel, client$_1$.grant, client$_2$.cancel, client$_2$.grant}, which is a strict subset of the complete interface alphabet (and is, in fact, the alphabet of the property). This assumption has just 3 states, and enables more efficient verification than the 8-state assumption obtained with the complete alphabet. In the following sections, we present techniques to infer smaller interface alphabets (and the corresponding assumptions) automatically.

4 Learning for Assume-Guarantee Reasoning

We briefly present here the assume-guarantee framework from [12]. The framework uses L* to infer assumptions for compositional verification. A central notion of the framework is that of the *weakest assumption* [15], defined formally here.

Definition 1 (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, $M_1 \parallel (M_2{\downarrow_\Sigma}) \models P$ iff $M_2 \models A_{w,\Sigma}$*

The notion of a weakest assumption depends on the interface between the component and its environment. Accordingly, projection of M_2 to Σ forces M_2 to communicate with our module only through Σ (second condition above). In [15] we showed that the weakest assumptions exist for components expressed as LTSs and safety properties and provided an algorithm for computing these 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 2 (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 3 (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, to deal with any system-level property, we allow properties in definition 2 to include actions that are not in αM_1 but are in αM_2. These actions need to be in the

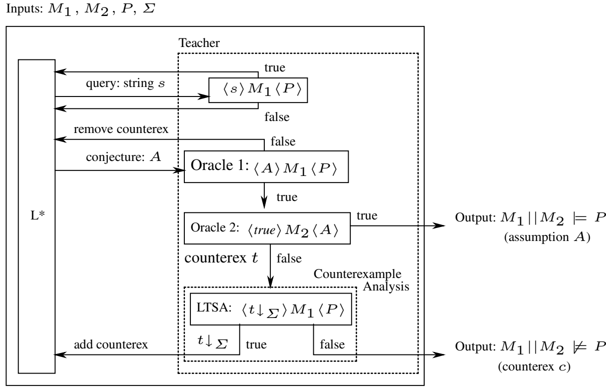

Fig. 3. Learning framework

interface since they are controllable by M_2. Moreover from the above definitions, it follows that $M_1 \parallel M_2 \models P$ iff $M_2 \models A_w$.

Learning framework. The original learning framework from [12] is illustrated in Fig. 3. The framework checks $M_1 \parallel M_2 \models P$ by checking the two premises of the assume-guarantee rule separately, and using the conjectures A from L* as assumptions. The alphabet given to the learner is fixed to $\Sigma = \Sigma_I$. The automaton A output by L* is, in the worst case, the *weakest assumption* A_w.

The Teacher is implemented using model checking. For membership queries on string s, the Teacher uses LTSA to check $\langle s \rangle M_1 \langle P \rangle$. If true, then $s \in \mathcal{L}(A_w)$, so the Teacher returns true. Otherwise, the answer to the query is false. The conjectures returned by L* are intermediate assumptions A. The Teacher implements two *oracles*: *Oracle 1* guides L* towards a conjecture that makes $\langle A \rangle M_1 \langle P \rangle$ true. Once this is accomplished, *Oracle 2* is invoked to discharge A on M_2. If this is true, then the assume guarantee rule guarantees that P holds on $M_1 \parallel M_2$. 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 good enough to prove that the property holds or is violated. If model checking returns a counterexample, further analysis is needed to determine if P is indeed violated in $M_1 \parallel M_2$ or if A is imprecise due to learning, in which case A needs to be modified.

Counterexample analysis. Trace t is the counterexample from Oracle 2 obtained by model checking $\langle true \rangle M_2 \langle A \rangle$. To determine if t is a real counterexample, *i.e.*, if it leads to error in $M_1 \parallel M_2 \parallel P_{err}$, the Teacher analyzes t on $M_1 \parallel P_{err}$. In doing so, the Teacher needs to first project t onto the assumption alphabet Σ, that is the interface of M_2 to $M_1 \parallel P_{err}$. Then the Teacher uses LTSA to check $\langle t\downarrow_\Sigma \rangle M_1 \langle P \rangle$. If the error state is not reached during the model checking, t is not a real counterexample, and $t\downarrow_\Sigma$ is returned to the learner L* to modify its conjecture. If the error state is reached, the model checker returns a counterexample c that witnesses the violation of P on M_1 in the context of $t\downarrow_\Sigma$. With the assumption alphabet $\Sigma = \Sigma_I$, c is guaranteed to be a real

error trace on $M_1 \parallel M_2 \parallel P_{err}$ [12]. However, as we shall see in the next section, if $\Sigma \subset \Sigma_I$, c is not necessarily a real counterexample and further analysis is needed.

5 Learning with Alphabet Refinement

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 $\Sigma \subset \Sigma_I$. Assume that we use the learning framework of the previous section, but we now set this smaller Σ to be the alphabet of the assumption that the framework learns. From the correctness of the assume-guarantee rule, if the framework reports true, $M_1 \parallel M_2 \models P$. When it reports false, it is because it finds a trace t in M_2 that falsifies $\langle t\downarrow_\Sigma \rangle M_1 \langle P \rangle$. This, however, does not necessarily mean that $M_1 \parallel M_2 \not\models P$. Real violations are discovered by our original framework only when the alphabet is Σ_I, and are traces t' of M_2 that falsify $\langle t'\downarrow_{\Sigma_I} \rangle M_1 \langle P \rangle$[1].

Consider again the client-server example. Assume Σ = {client$_1$.cancel, client$_1$.grant, client$_2$.grant}, which is smaller than Σ_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}. Learning with Σ produces trace: $t = \langle$client$_2$.request, client$_2$.grant, client$_2$.cancel, client$_1$.request, client$_1$.grant$\rangle$. Projected to Σ, this becomes $t\downarrow_\Sigma = \langle$client$_2$.grant, client$_1$.grant$\rangle$. In the context of $t\downarrow_\Sigma$, M_1 = Clients violates the property since Client$_1$ $\parallel$ Client$_2$ $\parallel$ P_{err} contains the following behavior (see Fig. 2):

$$(0,0,0) \xrightarrow{\text{client}_1\text{.request}} (1,0,0) \xrightarrow{\text{client}_2\text{.request}} (1,1,0) \xrightarrow{\text{client}_2\text{.grant}} (1,2,2) \xrightarrow{\text{client}_1\text{.grant}} (2,2,\mathit{error}).$$

Learning therefore reports *false*. This behavior is not feasible, however, in the context of $t\downarrow_{\Sigma_I} = \langle$client$_2$.request, client$_2$.grant, client$_2$.cancel, client$_1$.request, client$_1$.grant$\rangle$. This trace requires a client$_2$.cancel to occur before the client$_1$.grant. 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.

5.1 Algorithm

We propose a technique called *alphabet refinement*, which extends the learning framework from [12] to deal with smaller alphabets than Σ_I while avoiding spurious counterexamples. The steps of the algorithm are as follows (see Fig. 4 (a)):

1. **Initialize** Σ to a set S such that $S \subseteq \Sigma_I$.
2. Use the classic learning framework for Σ. If the framework returns *true*, then report *true* and go to step 4 (END). If the framework returns false with counterexamples c and t, go to the next step.
3. Perform **extended counterexample analysis** for c. If c is a real counterexample, then report *false* and go to step 4 (END). If c is spurious, then **refine** Σ, which consists of adding to Σ actions from Σ_I. Go to step 2.
4. END of algorithm.

[1] In the assume guarantee triples: $t\downarrow_\Sigma$, $t'\downarrow_{\Sigma_I}$ are trace LTSs with alphabets Σ, Σ_I respectively.

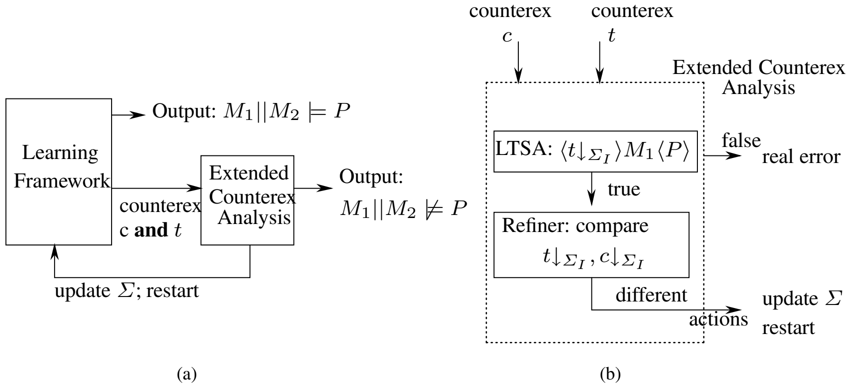

Fig. 4. Learning with alphabet refinement (a) and additional counterexample analysis (b)

When spurious counterexamples are detected, the refiner extends the alphabet with actions in the alphabet of the weakest assumption and the learning of assumptions is restarted. In the worst case, Σ_I is reached, and as proved in our previous work, learning then only reports real counterexamples. In the above high-level algorithm, the highlighted steps are further specified in the following.

Alphabet initialization. The correctness of our algorithm is insensitive to the initial alphabet. We implement two options: 1) we set the initial alphabet to the empty set to allow the algorithm to only take into account actions that it discovers, and 2) we set the initial alphabet to those actions in the alphabet of the property that are also in Σ_I,*i.e.*, $\alpha P \cap \Sigma_I$ (in the experiments from Sec. 7 we used the second option). The intuition for the latter option 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. 4(a). The steps of this analysis are shown in Fig. 4(b). The extension takes as inputs both the counterexample t returned by Oracle 2, and the counterexample c that is returned by the original counterexample analysis. We modified the "classic" learning framework (Fig. 3) to return both c **and** t to be used in alphabet refinement (as explained below). As discussed, c is obtained because $\langle t{\downarrow}_{\Sigma}\rangle M_1\langle P\rangle$ 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 $\langle t{\downarrow}_{\Sigma_I}\rangle M_1\langle P\rangle$. 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 enrich the current alphabet Σ so that these counterexamples are eventually eliminated.

A counterexample c is spurious if in the context of $t{\downarrow_{\Sigma_I}}$ it would not be obtained. Our refinement heuristics are therefore based on comparing c and $t{\downarrow_{\Sigma_I}}$ 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 $t{\downarrow_{\Sigma_I}}$ and $c{\downarrow_{\Sigma_I}}$; a potential problem is that it may add too many actions too soon, but if it happens to add useful actions, it may terminate after fewer iterations;

Forward: scans the traces in parallel from beginning to end looking for the first index i where they disagree; if such an i is found, both actions $t{\downarrow_{\Sigma_I}}(i), c{\downarrow_{\Sigma_I}}(i)$ are added to the alphabet;

Backward: same as Forward but scans from the end of the traces to the beginning.

5.2 Extension to n Modules

So far, we have discussed our algorithm for two components. We have extended alphabet refinement to n modules $M_1, M_2, \ldots M_n$, for any $n \geq 2$. Previous work has extended learning (without refinement) to n components [12,22]. To check if system $M_1 \parallel M_2 \parallel \ldots \parallel M_n$ satisfies P, we decompose it into: M_1 and $M_2' = M_2 \parallel ... \parallel M_n$ and the learning algorithm (without refinement) is invoked recursively for checking the second premise of the assume-guarantee rule.

Learning with alphabet refinement uses recursion in a similar way. At each recursive invocation for M_j, we solve the following problem: find assumption A_j and alphabet Σ_{A_j} such that the rule premises hold, *i.e.*

Oracle 1: $M_j \parallel A_j \models A_{j-1}$ and

Oracle 2: $M_{j+1} \parallel M_{j+2} \parallel ... \parallel M_n \models A_j$.

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 $\Sigma_I^j = (\alpha M_j \cup \alpha A_{j-1}) \cap (\alpha M_{j+1} \cup \alpha M_{j+2} \cup \ldots \cup \alpha M_n)$. If Oracle 2 returns a counterexample, then the counterexample analysis and alphabet refinement proceed exactly as in the 2 component case. At a new recursive call for M_j with a new A_{j-1}, the alphabet of the weakest assumption is recomputed.

6 Properties of Learning with Refinement

In this section, we discuss properties of the proposed algorithm. We present here the main results (proofs are given in [14]) We first re-state the correctness and termination of learning *without* refinement proven in [12].

Theorem 1 (Termination and correctness of learning without refinement). *Given components M_1 and M_2, and property P, the learning framework in [12] terminates and it returns true if $M_1 || M_2 \models P$ and false otherwise.*

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

Theorem 2 (Termination and correctness of learning with alphabet refinement – 2 components). *Given components M_1 and M_2, and property P, the algorithm* with alphabet refinement *terminates and returns true if $M_1||M_2 \models P$ and false otherwise.*

Theorem 3 (Termination and correctness of learning with alphabet refinement – n components). *Given components M_1, M_2, ... M_n and property P, the recursive algorithm* with alphabet refinement *terminates and returns true if $M_1||M_2||...||M_n \models P$ and false otherwise.*

Correctness for two (and n) components 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 and of n. Moreover, progress implies that the refinement algorithm for two components has at most $|\Sigma_I|$ iterations.

We also note a property of weakest assumptions, namely that by adding actions to an alphabet Σ, the corresponding weakest assumption becomes *weaker*, *i.e.*, it contains more behaviors.

Proposition 2. *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: $\mathcal{L}(A_{w,\Sigma}) \subseteq \mathcal{L}(A_{w,\Sigma'}) \subseteq \mathcal{L}(A_{w,\Sigma_I})$.*

With alphabet refinement, our framework adds actions to the alphabet, which translates into adding more behaviors to the weakest assumption that L* tries to prove. This means that at each refinement stage i, when the learner 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 learner might stop earlier, *i.e.*, before computing the corresponding weakest assumption. The above property allows re-use of learning results across refinement stages (see Sec. 9).

7 Experiments

We implemented learning with alphabet refinement in LTSA and we evaluated it on checking safety properties for the concurrent systems described below. The goal of the evaluation is to assess the effect of alphabet refinement on learning, and to compare compositional with non-compositional verification.

Models and properties. We used the following case studies. *Gas Station* [11] describes a self-serve gas station consisting of k customers, two pumps, and an operator. For

Table 1. Comparison of learning for 2-way decompositions with and without alphabet refinement

Case	k	No refinement			Refinement + bwd			Refinement + fwd			Refinement + allDiff		
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time
Gas Station	3	177	4.34	–	8	3.29	2.70	37	6.47	36.52	18	4.58	7.76
	4	195	100.21	–	8	24.06	19.58	37	46.95	256.82	18	36.06	52.72
	5	53	263.38	–	8	248.17	183.70	20	414.19	–	18	360.04	530.71
Chiron, Property 1	2	9	1.30	1.23	8	1.22	3.53	8	1.22	1.86	8	1.22	1.90
	3	21	5.70	5.71	20	6.10	23.82	20	6.06	7.40	20	6.06	7.77
	4	39	27.10	28.00	38	44.20	154.00	38	44.20	33.13	38	44.20	35.32
	5	111	569.24	607.72	110	–	300	110	–	300	110	–	300
Chiron, Property 2	2	9	116	110	3	1.05	0.73	3	1.05	0.73	3	1.05	0.74
	3	25	4.45	6.39	3	2.20	0.93	3	2.20	0.92	3	2.20	0.92
	4	45	25.49	32.18	3	8.13	1.69	3	8.13	1.67	3	8.13	1.67
	5	122	131.49	246.84	3	163.85	18.08	3	163.85	18.05	3	163.85	17.99
MER	2	40	6.57	7.84	6	1.78	1.01	6	1.78	1.02	6	1.78	1.01
	3	377	158.97	–	8	10.56	11.86	8	10.56	11.86	8	10.56	11.85
	4	38	391.24	–	10	514.41	1193.53	10	514.41	1225.95	10	514.41	1226.80
Rover Exec.	2	11	2.65	1.82	4	2.37	2.53	11	2.67	4.17	11	2.54	2.88

$k = 3, 4, 5$, we checked that the operator correctly gives change to a customer for the pump that he/she used. *Chiron* [11] 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* [22] 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* [12] 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.

Note that the Gas Station and Chiron were analyzed before, in [11], using learning based assume guarantee reasoning (with 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 approach, we selected the latter, more challenging properties for our study here.

Experimental set-up and results. We performed two sets of experiments. First, we compared learning *with* different alphabet refinement heuristics to learning *without* alphabet refinement for *2-way* decompositions. Second, we compared the recursive implementation of the refinement algorithm with monolithic (non-compositional) verification, for increasing number of components. All the experiments were performed on a Dell PC with a 2.8 GHz Intel Pentium 4 CPU and a 1.0 GB RAM, running Linux Fedora

Table 2. Comparison of recursive learning with and without alphabet refinement and monolithic verification

Case	k	No refinement			Refinement + bwd			Monolithic	
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	Mem.	Time
Gas Station	3	299	238.27	–	25	2.42	14.65	1.42	0.034
	4	289	298.22	–	25	3.43	23.60	2.11	0.126
	5	313	321.72	–	25	5.29	49.72	6.47	0.791
Chiron, Property 1	2	344	118.80	–	4	0.96	2.51	0.88	0.030
	3	182	114.57	–	4	1.12	2.97	1.53	0.067
	4	182	117.93	–	4	2.21	4.59	2.42	0.157
	5	182	115.10	–	4	7.77	6.97	13.39	1.22
Chiron, Property 2	2	229	134.85	–	11	1.68	40.75	1.21	0.035
	3	344	99.12	–	114	28.94	2250.23	1.63	0.068
	4	295	86.03	–	114	35.65	–	2.93	0.174
	5	295	90.57	–	114	40.49	–	15.73	1.53
MER	2	40	8.66	24.95	6	1.85	1.94	1.04	0.024
	3	440	200.55	–	8	3.12	3.58	4.22	0.107
	4	273	107.73	–	10	9.61	9.62	14.28	1.46
	5	200	83.07	–	12	18.95	23.55	143.11	27.84
	6	162	84.96	–	14	47.60	93.77	–	900

Core 4 and using Sun's Java SDK version 1.5. For the first set of experiments, for Gas Station and Chiron we used the best 2-way decompositions described in [11]. 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 [12]. For the second set of experiments, we used an additional heuristic to compute the *ordering* of the modules in the sequence $M_1, \ldots, M_n$ for the recursive learning with refinement so as 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 $\sum_{j=1..n} |\Sigma_I^j|$.

The experimental results shown in Tables 1 and 2 are for running the learning framework with 'No refinement', and for refinement with backward ('+bwd'), forward ('+ fwd') and '+allDiff' heuristics. For each run, we report $|A|$ (the *maximum* assumption size reached during learning), 'Mem.' (the *maximum* memory used by LTSA to check assume-guarantee triples, measured in MB) and 'Time' (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 exception is Chiron, Property 2, in our second study (Table 2) where the limit was 60 minutes (this was a challenging property and we increased the time limit in order to collect final results for our approach). The sign '–' indicates that the limit of 1GB of memory or the time limit has been exceeded. For these cases, the data is reported as it was when the limit was reached.

Discussion. The results overall show that alphabet refinement improves upon learning. Table 1 shows that alphabet refinement improved the assumption size in all cases, and in a few, up to two orders of magnitude (see Gas Station with $k = 2, 3$, Chiron, Property 3, 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 2: ‘No refinement’ exceeded the time limit in all but one case, whereas refinement completed in 14 of 16 cases, producing smaller assumptions and using less memory in all the cases, and up to two orders of magnitude in a few. Table 1 also indicates that the performance of the ‘bwd’ strategy is (slightly) better than the other refinement strategies. Therefore we used this strategy for the experiments reported in Table 2.

Table 2 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.

Chiron (Property 2) was particularly challenging for learning with (or without) alphabet refinement. At a closer inspection of the models, we noticed that several modules do not influence Property 2. However, these modules do communicate with the rest of the system through actions that appear in the counterexamples reported in our framework. As a result, alphabet refinement introduces ‘un-necessary’ actions. If we eliminate these modules, the property still holds in the remaining system, and the performance of learning with refinement is greatly improved, *e.g.*, for $k = 3$, the size of the largest assumption is 13 and is better than monolithic. In the future, we plan to investigate slicing techniques to eliminate modules that do not affect a given property.

8 Related Work

Several frameworks have been proposed to support assume guarantee reasoning [18,23,10,16]. For example, the Calvin tool [13] uses assume-guarantee reasoning for the analysis of Java programs, while Mocha [2] supports modular verification of components with requirements specified based in the Alternating-time Temporal logic. The practical impact of these previous approaches has been limited because they require non-trivial human input in defining appropriate assumptions.

Previous work [15,12] 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. This symbolic version was extended in [21] with algorithms that decompose models using hypergraph partitioning, to optimize the performance of learning on resulting decompositions. Different decompositions are also studied in [11] where the best two-way decompositions are computed for model-checking with the LTSA and FLAVERS tools. We follow a direction orthogonal to the latter two approaches and try to improve learning not by automating and optimizing decompositions, but rather by discovering small interface alphabets. Our approach can be combined with the decomposition approaches, by applying interface alphabet refinement in the context of the discovered decompositions.

L* has also been used in [1] to synthesize interfaces for Java classes, and in [25] to check component compatibility after component updates.

Our approach is similar in spirit to counterexample-guided abstraction refinement (CEGAR) [8]. 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 [17] 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 [6]. In that work, L* is started with an empty alphabet, and, similarly to ours, 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 by [6] focuses on assume-guarantee problems involving two components and it is not clear how it extends to reasoning about n components. The experiments in [6] report on the speed-up obtained with alphabet refinement. In all the reported cases, the alphabet needed for verification is very small. It is not clear if the same speed-up would be obtained for more challenging problems with bigger alphabets that would require many stages of refinement. In our experience, the memory savings obtained by smaller assumption sizes is the most significant gain. More experimentation is needed to fully assess the benefits of alphabet refinement and the relative strengths and weaknesses of the two approaches.

9 Conclusions and Future Work

We have introduced a novel technique for automatic and incremental refinement of interface alphabets in compositional model checking. Our approach extends an existing framework for learning assumption automata in assume-guarantee reasoning. The extension consists of using interface alphabets smaller than the ones previously used in learning, and using counterexamples obtained from model checking the components to add actions to these alphabets as needed. We have studied the properties of the new learning algorithm and have experimented with various refinement heuristics. Our experiments show improvement with respect to previous learning approaches in terms of the sizes of resulting assumptions and memory and time consumption, and with respect to non-compositional model checking, as the sizes of the checked models increase.

In future work we will address further algorithmic optimizations. Currently, after one refinement stage we restart the learning process from scratch. The property formulated in Proposition 2 in Sec. 6 facilitates reuse of query answers obtained during learning.

A query asks whether a trace projected on the current assumption alphabet leads to error on $M_1 \parallel P_{err}$. If the answer is 'no', by Proposition 2 the same trace will not lead to error when the alphabet is refined. Thus, we could cache these query answers. Another feasible direction is to reuse the learning table as described in [25]. We also plan to use multiple counterexamples for refinement. This may enable faster discovery of relevant interface actions and smaller alphabets. Finally we plan to perform more experiments to fully evaluate our technique.

Acknowledgements. We thank Jamie Cobleigh for providing the models Gas station and Chiron and their decompositions. Mihaela Gheorghiu acknowledges the financial support from MCT/Nasa Ames for a Summer Research Internship, and a Graduate Award from the University of Toronto.

References

1. R. Alur, P. Cerny, P. Madhusudan, and W. Nam. "Synthesis of interface specifications for Java classes". In *Proceedings of POPL'05*, pages 98–109, 2005.
2. R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. "MOCHA: Modularity in Model Checking". In *Proceedings of CAV'98*, volume 1427 of *LNCS*, pages 521–525, 1998.
3. R. Alur, P. Madhusudan, and Wonhong Nam. "Symbolic Compositional Verification by Learning Assumptions". In *Proceedings of CAV05*, pages 548–562, 2005.
4. D. Angluin. "Learning regular sets from queries and counterexamples". *Information and Computation*, 75(2):87–106, November 1987.
5. H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. "Proof Rules for Automated Compositional Verification through Learning". In *Proceedings of SAVCBS'03*, pages 14–21, 2003.
6. S. Chaki and O. Strichman. "Optimized L*-based Assume-guarantee Reasoning". In *Proceedings of TACAS'07 (to appear)*, 2007.
7. S.C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, 1999.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement". In *Proceedings of CAV'00*, volume 1855 of *LNCS*, pages 154–169, 2000.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
10. E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking". In *Proceedings of LICS'89*, pages 353–362, 1989.
11. J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. "Breaking Up is Hard to Do: An Investigation of Decomposition for Assume-Guarantee Reasoning". In *Proceedings of ISSTA'06*, pages 97–108. ACM Press, 2006.
12. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. "Learning Assumptions for Compositional Verification". In *Proceedings of TACAS'03*, volume 2619 of *LNCS*, pages 331–346, 2003.
13. C. Flanagan, S. N. Freund, and S. Qadeer. "Thread-Modular Verification for Shared-Memory Programs". In *Proceedings of ESOP'02*, pages 262–277, 2002.
14. M. Gheorghiu, D. Giannakopoulou, and C. S. Păsăreanu. "Refining Interface Alphabets for Compositional Verification". RIACS Technical Report, 2006.
15. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. "Assumption Generation for Software Component Verification". In *Proceedings of ASE'02*, pages 3–12. IEEE Computer Society, 2002.

16. O. Grumberg and D. E. Long. "Model Checking and Modular Verification". In *Proceedings of CONCUR'91*, pages 250–265, 1991.
17. T. A. Henzinger, R. Jhala, and R. Majumdar. "Permissive Interfaces". In *Proceedings of ESEC/SIGSOFT FSE'05*, pages 31–40, 2005.
18. C. B. Jones. "Specification and Design of (Parallel) Programs". In *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.
19. J.-P. Krimm and L. Mounier. "Compositional State Space Generation from Lotos Programs". In *Proceedings of TACAS'97*, pages 239–258, 1997.
20. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
21. W. Nam and R. Alur. "Learning-Based Symbolic Assume-Guarantee Reasoning with Automatic Decomposition". In *Proceedings of ATVA'06*, volume 4218 of *LNCS*, 2006.
22. C. S. Păsăreanu and D. Giannakopoulou. "Towards a Compositional SPIN". In *Proceedings of SPIN'06*, volume 3925 of *LNCS*, pages 234–251, 2006.
23. A. Pnueli. "In Transition from Global to Modular Temporal Reasoning about Programs". In *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, 1984.
24. R. L. Rivest and R. E. Shapire. "Inference of finite automata using homing sequences". *Information and Computation*, 103(2):299–347, April 1993.
25. N. Sharygina, S. Chaki, E. Clarke, and N. Sinha. "Dynamic Component Substitutability Analysis". In *Proceedings of FM'05*, volume 3582 of *LNCS*, pages 512–528, 2005.