Automated Assume-Guarantee Reasoning by Abstraction Refinement

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

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

Abstract. Current automated approaches for compositional model checking in the assume-guarantee style are based on learning of assumptions as deterministic automata. We propose an alternative approach based on abstraction refinement. Our new method computes the assumptions for the assume-guarantee rules as conservative and not necessarily deterministic abstractions of some of the components, and refines those abstractions using counterexamples obtained from model checking them together with the other components. Our approach also exploits the alphabets of the interfaces between components and performs iterative refinement of those alphabets as well as of the abstractions. We show experimentally that our preliminary implementation of the proposed alternative achieves similar or better performance than a previous learning-based implementation.

1 Introduction

Despite impressive recent progress in the application of model checking to the verification of realistic systems, the essential challenge in model checking remains the well-known state-space explosion problem [8]. Compositional techniques attempt to tame this problem by applying verification to individual components and merging the results without analyzing the whole system. 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 [13,15] 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.

In recent work, automation has been achieved through learning-based techniques [10]. The L* learning algorithm [2] is used to generate the assumptions needed for the assume-guarantee rules. The simplest such rule checks if a system composed of components M_1 and M_2 satisfies a property P by checking that M_1 under assumption A satisfies P (*Premise 1*) and discharging A on the environment M_2 (*Premise 2*). For safety properties, *Premise 2* amounts to checking that A is a conservative abstraction of M_2, *i.e.*, an abstraction that preserves all of M_2's execution paths. This rule is also represented as follows, where the notation is described in more detail in Section 2.

A. Gupta and S. Malik (Eds.): CAV 2008, LNCS 5123, pp. 135–148, 2008.

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

Learning-based assume-guarantee verification is an iterative process, during which L* makes conjectures in the form of automata that represent intermediate assumptions. Each conjectured assumption A is used to check the two premises of Rule 1. The process ends if A passes both premises of the rule, in which case the property holds in the system, or if it uncovers a real violation. Otherwise, a counterexample is returned and L* modifies the conjecture. Similar approaches are proposed in [1,4,17]; the work in [12] uses sampling rather than L* to learn the assumptions in a similar way.

In this paper we propose an alternative approach, AGAR (Assume-Guarantee Abstraction Refinement), that automates assume-guarantee reasoning by iteratively computing assumptions as conservative abstractions of the interface behavior of M_2, *i.e.*, the behavior that concerns the interaction with M_1. In each iteration, the computed assumption A satisfies *Premise 2* of the Rule 1 by construction and it is only checked for *Premise 1*. If the check is successful, we conclude that $M_1 \parallel M_2$ satisfies the property; if the check fails, we get a counterexample trace that we analyze to see if it corresponds to a real error in $M_1 \parallel M_2$ or it is spurious due to the over-approximation in the abstraction. If it is spurious, we used it to refine A and then repeat the entire process. Unlike learning-based assumption generation, AGAR does not constrain assumptions to be *deterministic*. Therefore the assumptions constructed with AGAR can be (potentially) exponentially smaller than those obtained with learning, resulting in smaller verification problems.

To reduce the assumption sizes even further, we also combine the abstraction refinement with an orthogonal technique, *interface alphabet refinement*, which extends AGAR so that it starts the construction of A with a small subset of the interface alphabet and adds actions to the alphabet as necessary until the required property is shown to hold or to be violated in the system. Actions to be added are discovered also by counterexample analysis. We introduced alphabet refinement in [11] for learning-based assume-guarantee reasoning; we adapt it here for AGAR[1]. We have implemented AGAR with alphabet refinement in the explicit state model checker LTSA [14] and performed a series of experiments that demonstrate that it can achieve better performance than L* for Rule 1 above.

Related work. AGAR is a variant of the well-known CEGAR (Counter Example-Guided Abstraction Refinement) [7] with the notable differences that the computed abstractions keep information only about the interface behavior of M_2 that concerns the interaction with M_1 while it abstracts away its internal behavior, and that the counterexamples used for the refinement of M_2's abstractions are obtained in an assume-guarantee style by model checking the other component, M_1.

CEGAR has been used before in compositional reasoning in [5]). In that work, a conservative abstraction of every component is constructed and then all the resulting abstractions are composed and checked. If the check passes, the verification concludes successfully, otherwise the resulting abstract counterexample is analyzed on every

[1] Note that [6] introduced a related alphabet minimization technique for L* as well.

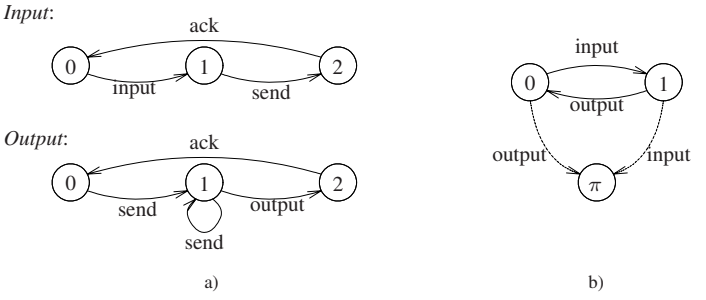

Fig. 1. (a) Example LTSs; (b) *Order* property

abstraction that is refined if needed. The work does not use assume-guarantee reasoning, it does not address the reduction of the interface alphabets and it has not been compared with learning-based techniques.

A comparison of learning and CEGAR-based techniques has been performed in [3] but for a different problem: the "interface synthesis" for a single component whose environment is unknown. In our context, this would mean generating an assumption that passes *Premise 1*, in the absence of a second component against which to check *Premise 2*. The interface being synthesized by the CEGAR-based algorithm in [3] is built as an abstraction of M_1. The work does not apply reduction to interface alphabets, nor does it address the verification of the generated interfaces against other components, *i.e.*, completing the assume-guarantee reasoning.

2 Preliminaries

Labeled Transition Systems (LTSs). We model components as finite-state *labeled transition systems* (LTSs), as considered by LTSA. Let $\mathbf{U}$ be the universal set of observable actions and let τ denote a special action that is unobservable.

An LTS M is a tuple $\langle Q, \Sigma, \delta, q_0 \rangle$, where: Q is a finite non-empty set of states; $\Sigma \subseteq \mathbf{U}$ is the *alphabet* of M; $\delta \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ is a transition relation, and q_0 is the initial state. We write $(q, a, q') \in \delta$ as $q \xrightarrow{a} q'$. 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*. π denotes an *error state* with no outgoing transitions, and Π denotes the LTS $\langle \{\pi\}, \mathbf{U}, \emptyset, \pi \rangle$. Let $M = \langle Q, \Sigma, \delta, q_0 \rangle$ and $M' = \langle Q', \Sigma', \delta', q'_0 \rangle$; M *transits* into M' with action a, denoted $M \xrightarrow{a} M'$, if $(q_0, a, q'_0) \in \delta$ and either $Q = Q', \Sigma = \Sigma'$, and $\delta = \delta'$ for $q'_0 \neq \pi$, or, in the special case where $q'_0 = \pi$, $M' = \Pi$.

Parallel Composition. Parallel composition "$\|$" is a commutative and associative operator such that: given LTSs $M_1 = \langle Q_1, \Sigma_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q_2, \Sigma_2, \delta^2, q_0^2 \rangle$, $M_1 \parallel M_2$ is Π if either one of M_1, M_2 is Π. Otherwise, $M_1 \parallel M_2$ is an LTS $M = \langle Q, \Sigma, \delta, q_0 \rangle$ where $Q = Q_1 \times Q_2, q_0 = (q_0^1, q_0^2), \Sigma = \Sigma_1 \cup \Sigma_2$, and δ is defined as follows (the symmetric version also applies): $M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2$ if $M_1 \xrightarrow{a} M_1', a \notin \Sigma_2$, and $M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'$ if $M_1 \xrightarrow{a} M_1', M_2 \xrightarrow{a} M_2', a \neq \tau$.

As an example [10], consider a simple communication channel that consists of two components whose LTSs are shown in Fig. 1(a).

Paths and traces. A *path* in an LTSs $M = \langle Q, \Sigma, \delta, q_0 \rangle$ is a sequence p of alternating states and (observable or unobservable actions) of M, $p = q_{i_0}, a_0, q_{i_1}, a_1, \ldots, a_{n-1}, q_{i_n}$ such that for every $k \in \{0, \ldots, n-1\}$ we have $(q_{i_k}, a_k, q_{i_{k+1}}) \in \delta$.

The *trace of path* p, denoted $\sigma(p)$ is the sequence $b_0, b_1, \ldots, b_l$ of actions along p, obtained by removing all τ from $a_0, \ldots, a_{n-1}$. A state q *reaches* a state q' in M with a sequence of actions t, denoted $q \stackrel{t}{\Rightarrow} q'$, if there exists a path p from q to q' in M whose trace is t, *i.e.*, $\sigma(p) = t$. A *trace of* M is the trace of a path in M starting from q_0. The set of all traces of M forms the *language* of M, denoted $\mathcal{L}(M)$. For any trace $t = a_0, a_1, \ldots, a_{n-1}$, a *trace LTS* can be constructed whose only transitions are $q_0 \stackrel{a_0}{\rightarrow} q_1 \stackrel{a_1}{\rightarrow} q_2 \ldots \stackrel{a_{n-1}}{\rightarrow} q_n$. We sometimes abuse the notation and denote by t both a trace and its trace LTS. The meaning should be clear from the context. For $\Sigma' \subseteq \Sigma$, $t{\downarrow_{\Sigma'}}$ is the trace obtained by removing from t all actions $a \notin \Sigma$. Similarly, $M{\downarrow_{\Sigma'}}$ is an LTS over Σ obtained from M by renaming to τ all the action labels not in Σ. Let t_1, t_2 be two traces. Let Σ_1, Σ_2 be the sets of actions occurring in t_1, t_2, respectively. By the *symmetric difference* of t_1 and t_2 we mean the symmetric difference of sets Σ_1 and Σ_2.

Safety properties. A *safety LTS* is a deterministic LTS not containing π. A safety property P is a *safety LTS* whose language $\mathcal{L}(P)$ defines the acceptable behaviors over Σ_P.

An LTS $M = \langle Q, \Sigma, \delta, q_0 \rangle$ satisfies $P = \langle Q_P, \Sigma_P, \delta_P, q_0^P \rangle$, denoted $M \models P$, iff $\forall t \in \mathcal{L}(M) \cdot t{\downarrow_{\Sigma_P}} \in \mathcal{L}(P)$. 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 so that the resulting transition relation is (left-)total (when seen as in $(Q \times (\Sigma \cup \{\tau\})) \times Q$) and deterministic; 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.

For example, the *Order* property in Fig. 1(b) states that inputs and outputs come in matched pairs, with the input always preceding the output. The dashed arrows represent transitions to the error state that were added to obtain $Order_{err}$.

Assume-guarantee triples. An assume-guarantee triple $\langle A \rangle M \langle P \rangle$ is true if whenever component M is part of a system satisfying assumption A, the system must also guarantee property P. In LTSA, this reduces to checking whether $A \parallel M \models P$.

Learning assumptions with L*. Previous work [10] uses the L* algorithm [2] to iteratively learn the assumption A for Rule 1, as a *deterministic* finite state automaton. L* needs to interact with a *teacher* that answers *queries* and validates *conjectures*. For membership queries on string s, the teacher uses LTSA to check $\langle s \rangle\ M_1\ \langle P \rangle$; if true, then $s \in \mathcal{L}(A)$ and the Teacher returns "true". Otherwise, the answer to the query is "false". The conjectures returned by L* are intermediate assumptions; the teacher implements two *oracles* to validate these conjectures: *Oracle 1* guides L* towards a conjecture that makes $\langle A \rangle\ M_1\ \langle P \rangle$ true and then *Oracle 2* is invoked to discharge A on M_2. If this is also true, then the assume guarantee rule ensures that P holds on $M_1 \parallel M_2$; the teacher returns "true" and the computed assumption A. If model checking returns "false", the returned counterexample is analyzed 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 is modified

and the process repeats. 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 A's alphabet and m is the length of the longest counterexample returned when a conjecture is made.

Interface alphabet. When reasoning in an assume-guarantee style, there is a natural notion of the complete *interface* between M_1 and M_2, when property P is checked. Let $M_1 = \langle Q_1, \Sigma_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q_2, \Sigma_2, \delta^2, q_0^2 \rangle$ be LTSs modeling two components and let $P = \langle Q_P, \Sigma_P, \delta^P, q_0^P \rangle$ be a safety property. The interface alphabet Σ_I is defined as $\Sigma_I = (\Sigma_1 \cup \Sigma_P) \cap \Sigma_2$.

3 Motivating Example

We motivate our approach using the input-output example from Section 2. We show that even on this simple example AGAR leads to smaller assumptions in fewer iterations than the learning approach, and therefore it potentially leads faster to smaller verification problems.

Let $M_1 =$ *Input*, $M_2 =$ *Output*, and $P =$ *Order*. As mentioned, we aim to automatically compute an assumption according to Rule 1. Instead of "guessing" an assumption and then checking both premises of the rule, as in the learning approaches, we *build* an abstraction that satisfies *Premise 2* by construction. Therefore, all that needs to be checked is *Premise 1*.

The initial abstraction A of *Output* is illustrated in Figure 2(a). Its alphabet consists of the interface between *Input* and the *Order* property on one side, and *Output* on the other, *i.e.*, the alphabet of A is $\Sigma_I = \{(\Sigma_{Input} \cup \Sigma_{Order}) \cap \Sigma_{Output}$. The LTS A is constructed simply by mapping all concrete states in *Output* to the same abstract state 0 which has a self-loop on every action in Σ_I and no other transitions. By construction, A is an overapproximation of M_2, *i.e.*, $\mathcal{L}(M_2 \downarrow_{\Sigma_I}) \subseteq \mathcal{L}(A)$, and therefore *Premise 2* $\langle true \rangle\ M_2\ \langle A \rangle$ holds. Checking *Premise 1* of the assume-guarantee rule using A as the assumption fails, with abstract counterexample: $0,\ output,\ 0$. We simulate this counterexample on M_2 and find that it is spurious (*i.e.*, it does not correspond to a trace in M_2), therefore A needs to be refined so that the refined abstraction no longer contains this trace. We split abstract state 0 into two new abstract states: abstract state 0, representing concrete states 0 and 2 that do not have an outgoing *output* action, and abstract state 1, representing concrete state 1 that has an outgoing *output* action, and adjust the transitions accordingly. The refined abstraction A', shown in Figure 2(a), is checked again for *Premise 1* and this time it passes, therefore AGAR terminates and reports that the property holds.

The sequence of assumptions learned with L* is shown in Figure 2(b). The assumption computed by AGAR thus has two states fewer than that obtained from learning and is computed in two fewer iterations.

4 Assume-Guarantee Abstraction Refinement (AGAR)

The abstraction refinement presented here is an adaptation of the CEGAR framework of [7], with the following notable differences: 1) abstraction refinement is performed

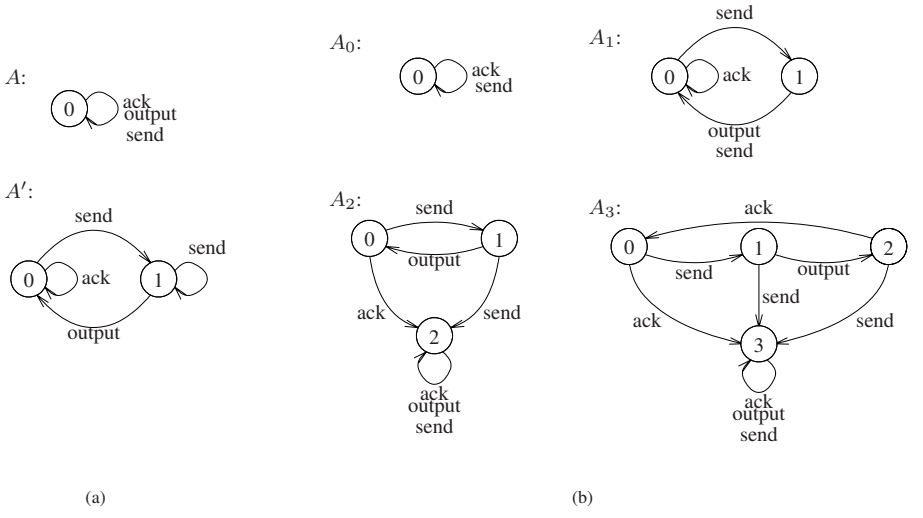

Fig. 2. Assumptions computed (a) with our algorithm and (b) with L*

in the context of LTSs; abstract transitions for LTSs are computed using *closure* with respect to actions that are not in their interface alphabet, and 2) counterexample analysis is performed in an assume-guarantee style: a counterexample obtained from model checking one component is used to refine abstractions of a different component.

In this section, we start by describing, independently of the assume-guarantee rule, abstraction refinement as applied to LTSs. We then describe how we use this abstraction refinement in an iterative algorithm (AGAR) that computes assumptions for Rule 1. Later on, we combine AGAR with an orthogonal algorithm that performs iterative refinement of the interface alphabet between the analyzed components.

4.1 Abstraction Refinement for LTSs

Abstraction. Let $C = \langle Q_C, \Sigma_C, \delta^C, q_0^C \rangle$ be an LTS that we refer to as *concrete*. Let alphabet Σ_A be such that $\Sigma_A \subseteq \Sigma_C$. An *abstraction* A of C is an LTS $\langle Q_A, \Sigma_A, \delta^A, q_0^A \rangle$ such that there exists a surjection $\alpha : Q_C \rightarrow Q_A$, called the *abstraction* function, that maps each *concrete state* $q^C \in Q_C$ to an *abstract state* $q^A \in Q_A$; q_0^A must be such that $\alpha(q_0^C) = q_0^A$. The *concretization* function $\gamma : Q_A \rightarrow 2^{Q_C}$ is defined for any $q^A \in Q_A$ as $\gamma(q^A) = \{q^C \in Q_C \mid \alpha(q^C) = q^A\}$. Note that γ induces a partition on Q_C, namely $\{\gamma(q^A) \mid q^A \in Q_A\}$.

To define the abstract transition relation δ^A, we first introduce the notion of reachability with respect to a subset alphabet. For $q^C \in C, a \in \Sigma_C$, we define the set $\mathit{Reachable}_C(q^C, a, \Sigma_A)$ of concrete states q_i^C reachable from q^C on action a, under the transitive closure of δ^C over actions in $(\Sigma_C \setminus \Sigma_A) \cup \{\tau\}$:

$$\mathit{Reachable}_C(q^C, a, \Sigma_A) = \{q_i^C \in C | \exists t, t' \in ((\Sigma_C \setminus \Sigma_A) \cup \{\tau\})^* \cdot q^C \stackrel{t}{\Rightarrow} q_i^C \text{ or } q^C \stackrel{t,b,t'}{\Rightarrow} q_i^C\}.$$

Algorithm 1. CEGAR for LTSs with respect to subset alphabets

Inputs: Concrete LTS C, its abstraction A, and an abstract counterexample $p = q_0^A, a_1, q_1^A, a_2, \ldots, a_n, q_n^A$ in A.

Outputs: a concrete counterexample t, if p is not spurious, or a refined abstraction A' without path p, if p is spurious.

1: $i \leftarrow 0$
2: $S_0 \leftarrow \{q_0^C\}$
3: **while** $S_i \neq \emptyset \wedge i \leq n-1$ **do**
4: $\quad i \leftarrow i+1$
5: $\quad S_i \leftarrow \gamma(q_i^A) \cap \mathit{Reachable}_C(S_{i-1}, a_i, \Sigma_A)$
6: **end while**
7: **if** $S_i = \emptyset$ **then**
8: $\quad$ split q_{i-1}^A into two new abstract states x_{i-1}^A, z_{i-1}^A s.t. $\gamma(x_{i-1}^A) = \gamma(q_{i-1}^A) \cap \{q^C \mid \mathit{Reachable}_C(q^C, a_i, \Sigma_A) \cap q_i^A \neq \emptyset\}$, $\gamma(z_{i-1}^A) = \gamma(q_{i-1}^A) \setminus \gamma(x_{i-1}^A)$
9: $\quad$ build new abstraction A' with $Q_{A'} = Q_A \setminus \{q_{i-1}^A\} \cup \{x_{i-1}^A, z_{i-1}^A\}$
10: $\quad$ change only incoming and outgoing transitions for q_{i-1}^A in A to/from $\{x_{i-1}^A, z_{i-1}^A\}$ in refined abstraction A', according to Definition 2
11: $\quad$ **return** A'
12: **else**
13: $\quad$ **return** concrete trace $t \leftarrow \sigma(p)$
14: **end if**

We define the abstraction to be *existential*, but using $\mathit{Reachable}_C$ instead of the usual transition relation of C [7]: $\exists(q_i^A, a, q_j^A) \in \delta^A$ iff

$$\exists q_i^C, q_j^C \in C \cdot \alpha(q_i^C) = q_i^A, \alpha(q_j^C) = q_j^A, \text{ and } q_j^C \in \mathit{Reachable}_C(q_i^C, a, \Sigma_A) \quad (2)$$

From the above definition and that of weak simulation [16], it follows that the abstraction defines a weak simulation relation between $C{\downarrow_{\Sigma_A}}$ and A. It is known that weak simulation implies trace inclusion [16]. We therefore have the following:

Proposition 1. *Given concrete LTS C and and its abstraction A defined as above, $\mathcal{L}(C{\downarrow_{\Sigma_A}}) \subseteq \mathcal{L}(A)$, and consequently $\langle true \rangle\ C\ \langle A \rangle$ hold.*

The CEGAR algorithm for LTSs is defined by Algorithm 1. It takes as inputs a concrete system C, an abstraction A (as defined above), and an abstract counterexample path p (in A). The algorithm analyzes the counterexample (lines 1–6) to see if it is real, in which case it is returned (line 13) or spurious, in which case it is used to refine the abstraction (lines 7–11). The refined abstraction A' is such that it no longer contains p. We discuss Algorithm 1 in more detail below.

Analysis of abstract counterexamples. Suppose we have obtained an *abstract counterexample* in the form of a path $p = q_0^A, a_1, q_1^A, a_2, \ldots, a_n, q_n^A$ in the abstraction A of C. We want to determine if it corresponds to a concrete path in C. For this we need to "play" (i.e. symbolically simulate) p in C from the initial state q_0^C. We do so considering that $\Sigma_A \subseteq \Sigma_C$ and thus we use $\mathit{Reachable}_C$ again.

We first extend $\mathit{Reachable}_C$ to sets: for $S \subseteq Q_C$, $\mathit{Reachable}_C(S, a, \Sigma_A) = \{q_j^C \in C \mid \exists q_i^C \in S. q_j^C \in \mathit{Reachable}(q_i^C, a, \Sigma_A)\}$. We play the abstract counterexample p

following [7]. We start at step 0 with the set $S_0 = \{q_0^C\}$ of concrete states, and the first transition $q_0^A \xrightarrow{a_1} q_1^A$ from p. Note that $S_0 = \{q_0^C\} \cap \gamma(q_0^A)$. At each step $i \in \{1, \ldots, n\}$, we compute the set $S_i = \gamma(q_i^A) \cap \mathit{Reachable}_C(S_{i-1}, a_i, \Sigma_A)$. If, for some $i \leq n$, S_i is empty, the abstract counterexample is spurious and we need to refine the abstraction to eliminate it. Otherwise, the counterexample corresponds to a concrete path.

Abstraction refinement. The abstraction refinement is performed in lines 8–10 of Algorithm 1: p is spurious because abstract state q_{i-1}^A does not distinguish between two disjoint, non-empty sets of concrete states [7]: (i) those that reach, with action a_i, states in the concretization of q_i^A (these are the states defined as $\gamma(x_{i-1}^A)$ in line 8) and (ii) those reached so far from q_0^C with the prefix $a_1, a_2, \ldots, a_{i-1}$, *i.e.*, the states in S_{i-1}.

To eliminate the spurious abstract path, we need to refine A by splitting its state q_{i-1}^A into (at least) two new abstract states that separate the (concrete) states of types (i) and (ii) (line 9). We split q_{i-1}^A into x_{i-1}^A where $\gamma(x_{i-1}^A)$ contains the set of states in (i) and z_{i-1}^A where $\gamma(z_{i-1}^A)$ contains the set of states in (ii) and any remaining states in $\gamma(q_{i-1}^A)$. Note that this results in a finer partition of the concrete states. After the splitting, we update the abstract transitions in line 10. The refined abstraction A' has the same transitions as A except for those incoming or outgoing for the split state q_{i-1}^A: they are readjusted to point to or from the states x_{i-1}^A, z_{i-1}^A according to condition 2. We therefore can conclude that:

Lemma 1. *If a counterexample p input to Algorithm 1 is spurious, the returned abstraction A' results in a strictly finer partition than A and does not contain p.*

4.2 The AGAR Algorithm

The pseudocode that combines Algorithm 1 with Rule 1 is given in Algorithm 2. Recall that Σ_I denotes the alphabet $(\Sigma_{M_1} \cup \Sigma_P) \cap \Sigma_{M_2}$ of the interface between M_1 and M_2, with respect to P. The algorithm checks that $M_1 \parallel M_2$ satisfies P using Rule 1. It builds abstractions A of M_2 in an iterative fashion (while loop at lines 2–15); these abstractions are used to check *Premise 1* of the assume guarantee rule using model checking (lines 3–5). If the check is successful, then, according to the rule (and since A satisfies *Premise 2* by construction), P indeed holds in $M_1 \parallel M_2$ and the algorithm returns "true". Otherwise, a counterexample p is obtained from model checking *Premise 1* (line 7) and Algorithm 1 is invoked to check if p corresponds to a real path in M_2 (in which case it means p is a real error in $M_1 \parallel M_2$ and this is reported to the user in line 11). If p is spurious, Algorithm 1 returns a refined abstraction A' for which we repeat the whole process starting from checking *Premise 1*.

Obtaining an abstract counterexample. As mentioned, we use counterexamples from failed checks of Premise 1 (that involves checking component M_1) to refine abstractions of M_2. Obtaining an abstract counterexample involves several steps (lines 7–9). First, a counterexample from line 4 is a path $o = q_0, b_1, q_1, b_2, \ldots, b_l, q_l$ in $A \parallel M_1 \parallel P_{err}$. Thus, for every $i \in \{0, l\}$, q_i is a triple of states (q_i^A, q_i^1, p_i) from $A \times M_1 \times P_{err}$. We first project every triple on A to obtain the sequence $o' = q_0^A, b_1, q_1^A, b_2, q_2^A, \ldots, b_l q_l^A$; o' is not yet a path in A as it may contain actions from M_1 and P_{err} that are not observable to A; those actions have to be between the same consecutive abstract states in the

Algorithm 2. AGAR: assume-guarantee verification by abstraction-refinement

Inputs: Component LTSs M_1, M_2, safety property LTS P, and alphabet $\Sigma_A = \Sigma_I$.
Outputs: **true** if $M_1 \parallel M_2$ satisfies P, **false** with a counterexample, otherwise.
Uses: Algorithm 1

1: Compute initial abstraction A of M_2, with a single state q_0^A having self-loops on all actions in Σ_A
2: **while true do**
3: Check *Premise 1*: $\langle A \rangle\ M_1\ \langle P \rangle$
4: **if** successful **then**
5: **return true**
6: **else**
7: Get counterexample $o = q_0, b_1, q_1, b_2, \ldots, b_l, q_l$ from line 3, where each $q_i = (q_i^A, q_i^1, p_i)$
8: Project o on A to get $o' = q_0^A, b_1, q_1^A, b_2, q_2^A, \ldots b_l, q_l^A$
9: Project o' on Σ_A to get abstract counterexample $p = q_0^A, a_1, q_1^A, \ldots, a_n, q_n^A$ in A.
10: **end if**
11: Call Algorithm 1 with inputs: M_2, A, p
12: **if** Algorithm 1 returned real counterexample t **then**
13: **return false** with counterexample t
14: **else**
15: $A = A'$
16: **end if**
17: **end while**

sequence, since they do not change the state of A; we eliminate from o' those actions and the duplicate abstract states that they connect, and finally obtain p that we pass to Algorithm 1.

Theorem 1. *Our algorithm (AGAR) computes a sequence of increasingly refined abstractions of M_2 until both premises of Rule 1 are satisfied, and we conclude that the property is satisfied by $M_1 \parallel M_2$, or a real counterexample is found that shows the violation of the property on $M_1 \parallel M_2$.*

Proof. Correctness The algorithm terminates when *Premise 1* is satisfied by the current abstraction or when a real counterexample is returned by Algorithm 1. In the former case, since the abstraction satisfies *Premise 2* by construction (Proposition 1), Rule 1 ensures that $M_1 \parallel M_2$ indeed satisfies P, so AGAR correctly returns answer "true". In the latter case, the counterexample returned by Algorithm 1 is a common trace of M_1 and of M_2 that leads to error in P_{err}. This shows that property P is violated on $M_1 \parallel M_2$ and in this case again AGAR correctly returns answer "false".

Termination. AGAR continues to refine the abstraction until a real counterexample is reported or the property holds. Refining the abstraction always results in a finer partition of its states (Lemma 1), and is thus guaranteed to terminate since in the worst case it converges to M_2 which is finite-state. □

If M_2 has n states, AGAR makes at most n refinement iterations, and in each iteration, counterexample analysis performs at most m closure operations, each of cost $O(n^3)$,

Algorithm 3. AGAR with alphabet refinement

Inputs: Component LTSs M_1, M_2, safety property LTS P, and alphabet $\Sigma_A \subseteq \Sigma_I$.
Outputs: **true** if $M_1 \parallel M_2$ satisfies P, **false** with a counterexample, otherwise.
Uses: Algorithm 2

```
1: while true do
2:   Call Algorithm 2 with M1, M2, P, ΣA.
3:   if Algorithm 2 returned true then
4:     return true
5:   else
6:     Obtain counterexample t = a1, ..., an from Algorithm 2 and trace s = σ(o') from
       line 8 of Algorithm 2.
7:     Check if error reachable in s^err↓ΣI || M2 where s^err↓ΣI is the trace-LTS ending with an
       extra transition into error state π
8:     if error reached then
9:       return false with counterexample s↓ΣI
10:    else
11:      Compare t to s↓ΣI to find difference action set Σ
12:      ΣA ← ΣA ∪ Σ
13:    end if
14:  end if
15: end while
```

where m is the length of the longest counterexample analyzed. This bound is not very tight as the closure steps are done on-the-fly to seldom exhibit worst-case behavior, and actually involve only parts of M_2's transition relation as needed.

4.3 AGAR with Interface Alphabet Refinement

In [11] we introduced an *alphabet refinement* technique to reduce the alphabet of the assumptions learned with L*. This technique improved significantly the performance of compositional verification. We show here how alphabet refinement can be similarly introduced in AGAR. Instead of the full interface alphabet Σ_I, we start AGAR from a small subset $\Sigma_A \subseteq \Sigma_I$. A good strategy is to start from those actions in Σ_I that appear in the property to be verified, since the verification should depend on them. We then run Algorithm 2 with this small Σ_A. Alphabet refinement introduces an extra layer of approximation, due to the smaller alphabet being used.

The pseudocode is in Algorithm 3. This algorithm adds an outer loop to AGAR (lines 1–15). At each iteration, it invokes AGAR (line 2) for the current alphabet Σ_A. If AGAR returns "true", it means that alphabet Σ_A is enough for proving the property (and "true" is returned to the user). Otherwise, the returned counterexample needs to be further analyzed (lines 5–13) to see if it corresponds to a real error (which is returned to the user in line 9) or it is spurious due to the approximation introduced by the smaller interface alphabet, in which case it is used to refine this alphabet (lines 11–12).

Additional counterexample analysis. As explained in [11], when $\Sigma_A \subset \Sigma_I$, the counterexamples obtained by applying Rule 1 may be spurious, in which case Σ_A needs to be extended. Intuitively, a counterexample is real if it is still a counterexample when

Table 1. Comparison of AGAR and learning for 2 components, with and without alphabet refinement

Case	k	No alpha. ref.						With alpha. ref.						Sizes	
		AGAR			Learning			AGAR			Learning				
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|M_1 \parallel P_{err}\|$	$\|M_2\|$
Gas Station	3	**16**	**4.11**	**3.33**	177	42.83	–	**5**	**2.99**	**2.09**	8	3.28	3.40	1960	643
	4	**19**	**37.43**	**23.12**	195	100.17	–	**5**	**22.79**	**12.80**	8	25.21	19.46	16464	1623
	5	**22**	**359.53**	**278.63**	45	206.61	–	**5**	216.07	**83.34**	8	**207.29**	188.98	134456	3447
Chiron, Property 2	2	10	**1.30**	**0.92**	9	**1.30**	1.69	10	1.30	**1.56**	**8**	**1.22**	5.17	237	102
	3	36	**2.59**	**5.94**	**21**	5.59	7.08	36	**2.44**	**10.23**	**20**	6.00	30.75	449	1122
	4	160	**8.71**	152.34	**39**	27.1	**32.05**	160	**8.22**	252.06	**38**	41.50	**180.82**	804	5559
	5	4	55.14	–	**111**	**569.23**	**676.02**	3	58.71	–	110	–	386.6	2030	129228
Chiron, Property 3	2	**4**	**1.07**	**0.50**	9	1.14	1.57	4	1.23	**0.62**	**3**	**1.06**	0.91	258	102
	3	**8**	**1.84**	**1.60**	25 n jmj	4.45	7.72	8	**2.00**	3.65	**3**	2.28	**1.12**	482	1122
	4	**16**	**4.01**	**18.75**	45	25.49	36.33	16	**5.08**	107.50	**3**	**7.30**	1.95	846	5559
	5	4	52.53	–	**122**	**134.21**	**271.30**	1	81.89	–	**3**	**163.45**	**19.43**	2084	129228
MER	2	**34**	**1.42**	11.38	40	6.75	**9.89**	**5**	**1.42**	5.02	6	1.89	**1.28**	143	1270
	3	**67**	**8.10**	**247.73**	335	133.34	–	9	11.09	180.13	**8**	**8.78**	**12.56**	6683	7138
	4	58	341.49	–	38	377.21	–	9	532.49	–	**10**	**489.51**	**1220.62**	307623	22886
Rover Exec.	2	**10**	**4.07**	**1.80**	11	2.70	2.35	**3**	2.62	**2.07**	4	2.46	3.30	544	41

considered with Σ_I. For counterexample analysis, we modify Algorithm 2 to also output the trace $s = \sigma(o')$ of actions along the intermediate path o' obtained at its line 8. Since p is a path obtained from o' by eliminating transitions labeled with actions from $\Sigma_I \setminus \Sigma_A$ (See Section 4.2) and $t = \sigma(p)$, it follows that s is an "extension" of t to Σ_I.

We check whether $s{\downarrow_{\Sigma_I}}$ is a trace of M_2 by making it into a trace LTS ending with the error state π, and whose alphabet is Σ_I (line 7). Since M_2 does not contain π, the only way to reach error is if $s{\downarrow_{\Sigma_I}}$ is a trace of M_2; if we reach error, the counterexample t is real. If $s{\downarrow_{\Sigma_I}}$ is not a trace of M_2, since t is, we need to refine the current alphabet Σ_A. At this point we have two traces, $s{\downarrow_{\Sigma_I}}$ and t that agree with respect to Σ_A and only differ on the actions from $\Sigma_I \setminus \Sigma_A$; since one trace is in M_2 and the other is not, we are guaranteed to find in their symmetric difference at least an action that we can add to Σ_A to eliminate the spurious counterexample t. We include the new action(s) and then repeat AGAR with the new alphabet. Termination follows from the fact that the interface alphabet is finite.

5 Evaluation

We implemented AGAR with alphabet refinement for Rule 1 in the LTSA tool. We compared AGAR with learning based assume guarantee reasoning, using a similar experimental setup as in [11]. The case studies are: *Gas Station* (with $3 \ldots 5$ customers), *Chiron* – a model of a GUI (with $2 \ldots 5$ event handlers), and two NASA models: *MER* resource arbiter (with $2 \ldots 4$ threads competing for a common resource) and *Rover*, with an executive and an event monitoring component. We first used the same two-way decompositions of these models as described in [11]. For *Gas Station* and *Chiron*, these decompositions were demonstrated to be the best for the performance of learning (without alphabet refinement) among all possible two-way decompositions [9].

Table 2. Comparison of AGAR and learning for balanced decompositions

Case	k	No alpha. ref.						With alpha. ref.						Sizes	
		AGAR			Learning			AGAR			Learning				
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|M_1 \parallel P\|$	$\|M_2\|$
Gas Station	3	**10**	**3.35**	**3.36**	294	367.13	–	**5**	**2.16**	**3.06**	59	11.14	81.19	1692	1942
	4	269	174.03	–	433	188.94	–	10	15.57	191.96	**5**	**9.25**	**4.73**	4608	6324
	5	**7**	**47.91**	**184.64**	113	82.59	–	2	47.48	–	**15**	**52.41**	**71.29**	31411	32768
Chiron, Property 2	2	**41**	**2.45**	**5.46**	140	118.59	395.56	**9**	**1.91**	**3.89**	17	2.73	13.09	906	924
	3	**261**	**81.24**	**710.1**	391	134.57	–	**79**	**39.94**	**663.53**	217	36.12	–	6104	6026
	4	**54**	**7.11**	**37.91**	354	383.93	–	**45**	**9.55**	**121.66**	586	213.78	–	1308	1513
	5	402	73.74	–	112	90.22	–	**33**	**19.66**	**157.35**	46	30.05	686.37	11157	11748
Chiron, Property 3	2	**2**	**0.98**	**0.37**	40	5.21	8.30	**2**	**1.02**	**0.49**	3	1.04	0.91	168	176
	3	**88**	**15.45**	**102.93**	184	284.83	–	46	41.40	115.77	**3**	**5.97**	**2.26**	4240	4186
	4	**2**	**5.60**	**2.65**	408	222.54	–	**2**	**6.14**	11.90	20	9.33	**7.44**	4156	4142
	5	**79**	**44.16**	405.03	179	104.25	–	42	42.04	430.47	**3**	**21.94**	**7.00**	16431	16840
MER	4	9	27.62	–	311	104.72	–	2	27.60	–	**10**	**65.42**	**35.78**	10045	66230

All 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 Core 4 and Sun's Java SDK version 1.5. We report the maximum assumption size (*i.e.*, number of states) reached ("$|A|$"), the memory consumed ("Mem.") in MB, the time ("Time") in seconds, and the numbers of states on each side of the two-way decomposition: "$|M_1 \parallel P_{err}|$" and "$|M_2|$". A "–" indicates that the limit of 1G of memory or 30 minutes has been exceeded. For those cases, the other quantities are shown as they were when the limit was reached. We also highlight in bold font the best results.

The results for the first set of experiments are shown in Table 1. Overall, AGAR shows similar or better results than learning in more than half of the cases. From the results, we noticed that the relative sizes of $M_1 \parallel P_{err}$ and M_2 seem to influence the performance of the two algorithms; *e.g.*, for *Gas Station*, where M_2 is consistently smaller, AGAR is consistently better, while for *Chiron*, as the size of M_2 becomes much larger, the performance of AGAR seems to degrade. Furthermore, we observed that the learning runs exercise more the first component, whereas AGAR exercises both. We therefore considered a second set of experiments were we tried to compare the relative performance of the two approaches for two-way system decompositions that are more balanced in terms of number of states.

We generated off-line all the possible two-way decompositions and chose those minimizing the difference in number of states between $M_1 \parallel P_{err}$ and M_2. The rest of the setup remained the same. The results for these new decompositions are in Table 2 (for MER, in only one case we found a more balanced partition than previously). These results show that with these new decompositions AGAR is consistently better in terms of time (14/21 cases), memory (16/21 cases) and assumption size (16/21 cases)[2]. The results also indicate that the benefits of alphabet refinement are more pronounced for learning. The results are somewhat non-uniform as k increases because for each larger value of k we re-computed balanced decompositions independently of those for smaller values. This is why we even found smaller components for larger parameter, as for Chiron, Property 2, $k = 3$ vs. $k = 4$.

[2] We did not count the cases when both algorithms ran out of limits.

6 Conclusions and Future Work

We have introduced an assume-guarantee abstraction-refinement technique (AGAR) as an alternative to learning-based approaches. Our preliminary results clearly indicate that the alternative is feasible. We are currently extending AGAR with the following rule (for reasoning about n components).

$$\frac{\begin{array}{l}\textit{(Premise 1)}\ \langle A_1\rangle\ M_1\ \langle P\rangle \\ \textit{(Premise 2)}\ \langle A_2\rangle\ M_2\ \langle A_1\rangle \\ \quad\ldots \\ \textit{(Premise n)}\ \langle \textit{true}\rangle\ M_n\ \langle A_{n-1}\rangle\end{array}}{\langle \textit{true}\rangle\ M_1 \parallel M_2 \parallel \ldots \parallel M_n\ \langle P\rangle} \tag{3}$$

In previous work [11], learning with this rule overcame the intermediate state explosion related to two-way decompositions (*i.e.*, when components are larger than the entire system). That helped us demonstrate better scalability of compositional vs. non-compositional verification which we believe to be the ultimate test of any compositional technique. We expect to similarly achieve better scalability for AGAR.

The implementation of AGAR for Rule 3 involves the creation of $n-1$ instances AR_i of our abstraction-refinement code for computing each A_i as an abstraction of $M_{i+1} \parallel A_{i+1}$, except for A_{n-1} which abstracts M_n . Counterexamples obtained from *(Premise 1)* are used to refine the intermediate abstractions $A_1, \ldots, A_{n-1}$. When A_i is refined, all the abstractions $A_1, \ldots, A_{i-1}$ are refined as well to eliminate the spurious trace. In the future, we also plan to explore extensions of AGAR to liveness properties.

Acknowledgements. We thank Moshe Vardi and Orna Grumberg for helpful suggestions and the CAV reviewers for their comments.

References

1. Alur, R., Madhusudan, P., Nam, W.: Symbolic Compositional Verification by Learning Assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. and Comp. 75(2), 87–106 (1987)
3. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for Interface Synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 4–19. Springer, Heidelberg (2007)
4. Chaki, S., Clarke, E.M., Sinha, N., Thati, P.: Automated Assume-Guarantee Reasoning for Simulation Conformance. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 534–547. Springer, Heidelberg (2005)
5. Chaki, S., Ouaknine, J., Yorav, K., Clarke, E.: Automated Compositional Abstraction Refinement for Concurrent C Programs: A Two-Level Approach. ENTCS 89(3) (2003)
6. Chaki, S., Strichman, O.: Optimized L*-Based Assume-Guarantee Reasoning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 276–291. Springer, Heidelberg (2007)
7. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)

8. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT, Cambridge (2000)
9. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking Up is Hard to Do: An Investigation of Decomposition for Assume-Guarantee Reasoning. In: Proc. of ISSTA 2006, pp. 97–108. ACM, New York (2006)
10. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning Assumptions for Compositional Verification. In: ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
11. Gheorghiu, M., Giannakopoulou, D., Pasareanu, C.S.: Refining Interface Alphabets for Compositional Verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007)
12. Gupta, A., McMillan, K.L., Fu, Z.: Automated Assumption Generation for Compositional Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 420–432. Springer, Heidelberg (2007)
13. Jones, C.B.: Specification and Design of (Parallel) Programs. In: Inf. Proc. 1983: Proc. of IFIP 9th World Congress, pp. 321–332. North Holland, Amsterdam (1983)
14. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs. John Wiley & Sons, Chichester (1999)
15. Pnueli, A.: In Transition from Global to Modular Temporal Reasoning about Programs. Logic and Models of Conc. Sys. 13, 123–144 (1984)
16. Milner, R.: Communication and Concurrency. Prentice-Hall, New York (1989)
17. Sinha, N., Clarke, E.M.: SAT-Based Compositional Verification Using Lazy Learning. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 39–54. Springer, Heidelberg (2007)