riacs Research Institute for Advanced Computer Science

Learning Assumptions for Compositional Verification

Jamieson M. Cobleigh
Dimitra Giannakopoulou
Corina Păsăreanu

RIACS Technical Report 02.09

November 2002

Learning Assumptions for Compositional Verification

Jamieson Cobleigh, University of Massachusets, Amherst
Dimitra Giannakopoulou, RIACS
Corina Păsăreanu, Kestrel Technologies

RIACS Technical Report 02.09

November 2002

Compositional verification is a promising approach to addressing the state explosion problem associated with model checking. One compositional technique advocates proving properties of a system by checking properties of its components in an assume-guarantee style. However, the application of this technique is difficult because it involves non-trivial human input. This paper presents a novel framework for performing assume-guarantee reasoning in an incremental and fully automated fashion. To check a component against a property, our approach generates assumptions that the environment needs to satisfy for the property to hold. These assumptions are then discharged on the rest of the system. Assumptions are computed by a learning algorithm. They are initially approximate, but become gradually more precise by means of counterexamples obtained by model checking the component and its environment, alternately. This iterative process may at any stage conclude that the property is either true or false in the system. We have implemented our approach in the LTSA tool and applied it to the analysis of a NASA system.

This work was supported in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC 2-1006 with the Universities Space Research Association (USRA).

This report is available online at http://www.riacs.edu/trs/

Learning Assumptions for Compositional Verification

Jamieson M. Cobleigh*[1], Dimitra Giannakopoulou[2], and Corina S. Păsăreanu[2]

[1] Department of Computer Science, University of Massachusets
Amherst, MA 01003-9264, USA
jcobleig@cs.umass.edu

[2] NASA Ames Research Center, Moffett Field, CA 94035-1000, USA
{dimitra, pcorina}@email.arc.nasa.gov

Abstract. Compositional verification is a promising approach to addressing the state explosion problem associated with model checking. One compositional technique advocates proving properties of a system by checking properties of its components in an assume-guarantee style. However, the application of this technique is difficult because it involves non-trivial human input. This paper presents a novel framework for performing assume-guarantee reasoning in an incremental and fully automated fashion. To check a component against a property, our approach generates assumptions that the environment needs to satisfy for the property to hold. These assumptions are then discharged on the rest of the system. Assumptions are computed by a learning algorithm. They are initially approximate, but become gradually more precise by means of counterexamples obtained by model checking the component and its environment, alternately. This iterative process may at any stage conclude that the property is either true or false in the system. We have implemented our approach in the LTSA tool and applied it to the analysis of a NASA system.

1 Introduction

Our work is motivated by an ongoing project at NASA Ames Research Center on the application of model checking to the verification of autonomous software. Autonomous software involves complex concurrent behaviors for reacting to external stimuli without human intervention. Extensive verification is a pre-requisite for the deployment of missions that involve autonomy.

Given some formal description of a system and of a required property, model checking automatically determines whether the property is satisfied by the system. If the property is violated, it returns a counterexample, i.e., an execution of the system that exhibits erroneous behavior. The limitation of the approach, referred to as the "state-explosion" problem [8], is that it needs to store the explored system states in memory, which is impossible for most realistic systems.

Compositional verification presents a promising way of addressing state explosion. It advocates a "divide and conquer" approach where properties of the system are decomposed into properties of its components, so that if each component satisfies its respective property, then so does the entire system. Components are therefore model checked separately. It is often the case, however, that components only satisfy properties in specific contexts (also called environments). This has given rise to the assume-guarantee style of reasoning [18, 21].

* This author is grateful for the support received from RIACS to undertake this research while participating in the Summer Student Research Program at the NASA Ames Research Center.

Assume-guarantee reasoning first checks whether a component M guarantees a property P, when it is part of a system that satisfies an assumption A. Intuitively, A characterizes all contexts in which the component is expected to operate correctly. To complete the proof, it must also be shown that the remaining components in the system, i.e., M's environment, satisfy A. This style of reasoning is typically performed in an interactive fashion. Developers first check a component under no assumptions about the environment. If a counterexample is returned that is unrealistic for the system under analysis, they make several attempts at *manually* defining an assumption that is strong enough to eliminate false violations, but that also reflects appropriately the remaining system.

This paper presents a novel framework for performing assume-guarantee reasoning in an *incremental* and *fully automatic* fashion. Our approach iterates a process based on gradually *learning* assumptions. The learning process is based on queries to component M, and on counterexamples obtained by model checking M and its environment, alternately. Each iteration may conclude that the required property is satisfied or violated in the system analyzed. This process is guaranteed to terminate; in fact, it converges to an assumption that is necessary and sufficient for the property to hold in the specific system.

Our approach has been implemented in the Labeled Transition Systems Analyzer (LTSA) [20], and applied to the analysis of the Executive module of an experimental Mars Rover (K9) developed at NASA Ames. We are currently in the process of also implementing it in Java Pathfinder (JPF) [23]. In fact, as our approach relies on standard features of model checkers, it is fairly straightforward to add in any such tool.

The remainder of the paper is organized as follows. We first provide some background in Section 2, followed by a high level description of the framework that we propose in Section 3. The algorithms that implement this framework are presented in Section 4. We discuss the applicability of our approach in practice and extensions that we are considering in Section 5. Section 6 describes our experience with applying our approach to the Executive of the K9 Rover. Finally, Section 7 presents related work, and Section 8 concludes the paper.

2 Background

The presentation of our approach is based on techniques for modeling and checking concurrent programs implemented in the LTSA tool [20]. The LTSA supports Compositional Reachability Analysis (CRA) of a software system based on its architecture, which, in general, has a hierarchical structure. CRA incrementally computes and abstracts the behavior of composite components based on the behavior of their immediate children in the hierarchy [13]. The flexibility that the LTSA provides in selecting any component in the hierarchy for analysis or abstraction makes it ideal for experimenting with our approach.

Labeled Transition Systems. The LTSA uses Labeled Transition Systems to model the behavior of communicating components in a concurrent system. Let *Act* be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment. We use π to denote a special *error state*, which models the fact that a safety violation has occurred in the

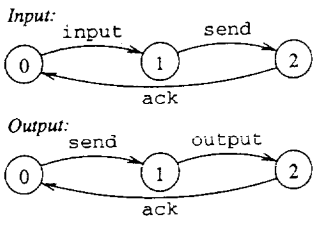

Fig. 1. Example LTSs

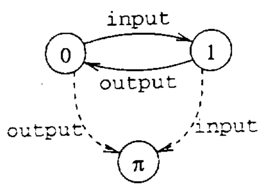

Fig. 2. *Order* Property

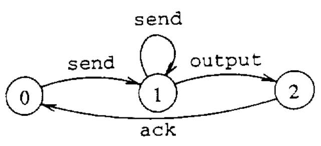

Fig. 3. LTS for *Output'*

associated system. We require that the error state has no outgoing transitions. Formally, a *Labeled Transition System* (LTS) M is a four tuple $\langle Q, \alpha M, \delta, q_0 \rangle$ where:

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

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

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

Traces. A *trace* σ of an LTS M is a sequence of observable actions that M can perform starting at its initial state. For example, $\langle$`input`$\rangle$ and $\langle$`input, send`$\rangle$ are both traces of the *Input* LTS in Fig. 1. For $\Sigma \subseteq \mathcal{A}ct$, we use $\sigma \upharpoonright \Sigma$ to denote the trace obtained by removing from σ all occurrences of actions $a \notin \Sigma$. The set of all traces of M is called the *language* of M, denoted $\mathcal{L}(M)$.

Parallel Composition. We provide transitional semantics for parallel composition in a typical process algebra style, although our aim here is not to define an algebra. 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 $\alpha M = \alpha M'$ and $\delta = \delta'$ for $q'_0 \neq \pi$, or, in the special case where $q'_0 = \pi$, $M' = \Pi$.

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

Formally, let $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$ be two LTSs. If $M_1 = \Pi$ or $M_2 = \Pi$, then $M_1 \parallel M_2 = \Pi$. Otherwise, $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows (note that the symmetric rules are implied by the fact that the operator is commutative):

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

Properties. We call a deterministic LTS that contains no π states a *safety LTS*. A *safety property* is specified as a safety LTS P, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP. An LTS M satisfies P, denoted as $M \models P$, if and only if $\forall \sigma \in \mathcal{L}(M) : (\sigma \upharpoonright \alpha P) \in \mathcal{L}(P)$.

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

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

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

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

To detect violations of property P by component M, the parallel composition $M \parallel P_{err}$ is computed. It has been proved that M violates P if and only if the π state is reachable in $M \parallel P_{err}$ [5]. For the example system, state π is not reachable in $Input \parallel Output \parallel Order_{err}$, so $Input \parallel Output \models Order$.

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 [21]. The formula is true if whenever M is part of a system satisfying A, then the system must also guarantee P.

The LTSA is particularly flexible in performing assume-guarantee reasoning. Both assumptions and properties are defined as safety LTSs[3]. In fact, a safety LTS A can be used as an assumption *or* as a property. To be used as an assumption for module M, A itself is composed with M, thus playing the role of an abstraction of M's environment. To be used as a property to be checked on M, A is turned into A_{err} and then composed with M.

To check an assume-guarantee formula $\langle A \rangle\ M\ \langle P \rangle$, where both A and P are safety LTSs, the LTSA computes $A \parallel M \parallel P_{err}$ and checks if state π is reachable in the composition. If it is, then $\langle A \rangle\ M\ \langle P \rangle$ is violated by component M, otherwise it is satisfied.

[3] Any LTS without π states can be transformed into a safety LTS by determinization.

Deterministic Automata and Safety LTSs. One of the components of our framework is a learning algorithm that produces Deterministic Finite-State Automata, which our framework then uses as safety LTSs. A Deterministic Finite-State Automaton (DFA) M is a five tuple $\langle Q, \alpha M, \delta, q_0, F\rangle$ where:

- Q is a finite set of states
- $\alpha M \subseteq Act$ is a set of observable actions that make up the alphabet of M
- $\delta : Q \times \alpha M \rightarrow Q$ is a transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states

For a DFA M and a string σ, we use $\delta(q, \sigma)$ to denote the state that M will be in after reading σ starting at state q. A string σ is said to be *accepted* by a DFA $M = \langle Q, \alpha M, \delta, q_0, F\rangle$ if $\delta(q_0, \sigma) \in F$. The *language accepted by* M, designated $\mathcal{L}(M)$ is the set $\{\sigma \mid \delta(q_0, \sigma) \in F\}$.

The DFAs returned by the learning algorithm in our context are *complete, minimal,* and *prefix-closed* (an automaton M is prefix-closed if $\mathcal{L}(M)$ is prefix-closed, i.e., for every $\sigma \in \mathcal{L}(M)$, every prefix of σ is also in $\mathcal{L}(M)$). These DFAs therefore contain a single non-accepting state. They can easily be transformed into safety LTSs by removing the non-accepting state, which corresponds to state π of an error LTS, and all transitions that lead into it.

3 Framework for Incremental Compositional Verification

For simplicity, let us consider the case where a system is made up of two components, M_1 and M_2. As mentioned in the previous section, a formula $\langle A\rangle\ M\ \langle P\rangle$ is true if, whenever M is part of a system satisfying A, then the system must also guarantee property P. The simplest compositional proof rule shows that if $\langle A\rangle\ M_1\ \langle P\rangle$ and $\langle true\rangle\ M_2\ \langle A\rangle$ hold, then $\langle true\rangle\ M_1 \parallel M_2\ \langle P\rangle$ is true. This proof strategy can also be expressed as an inference rule as follows:

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

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

For the use of the compositional rule in proving $\langle true\rangle\ M_1 \parallel M_2\ \langle P\rangle$ to be justified, the assumption must be more abstract than M_2. An appropriate assumption for the rule needs to be strong enough for M_1 to satisfy P in Step 1. Moreover, the restrictions it places on M_2 should reflect M_2's behavior. Coming up directly with an appropriate assumption for the application of the compositional rule is a non-trivial process. So in practice, the rule is typically applied in an iterative fashion as illustrated in Fig. 4. At each iteration i, an assumption A_i is provided based on some knowledge about the system and on the results of the previous iteration. A model checker can then be used to automatically apply the two steps of the compositional rule.

Step 1 is applied first, to check whether M_1 guarantees P in environments that satisfy A_i. If the result is false, it means that this assumption is too *weak*, i.e., A_i does not restrict the environment

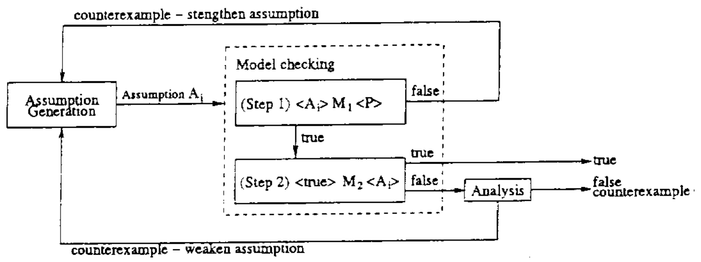

Fig. 4. Incremental compositional verification during iteration i

enough for P to be satisfied. The assumption therefore needs to be *strengthened* (which corresponds to removing behaviors from it) with the help of the counterexample produced by Step 1. In the context of the next assumption A_{i+1}, component M_1 should not exhibit the violating behavior reflected by this counterexample, at least.

If Step 1 returns true, it means that A_i is strong enough for the property to be satisfied. To complete the proof, Step 2 must be applied to discharge A_i on M_2. If Step 2 returns true, then the compositional rule guarantees that P holds in $M_1 \parallel M_2$. If it returns false, further analysis is required to identify whether P is indeed violated in $M_1 \parallel M_2$ or whether A_i was stronger than necessary. Such analysis is usually based on the counterexample returned by Step 2.

If assumption A_i is too strong it must be *weakened* (i.e., behaviors must be added) in iteration $i + 1$. The result of such weakening will be that at least the behavior that the counterexample represents will be allowed by assumption A_{i+1}. The new assumption may of course be too weak, and therefore the entire process must be repeated.

The bottleneck in the application of the above process lies in the fact that coming up with and refining assumptions manually tends to be a mental challenge. To address this issue, we have developed a framework that implements this iterative, incremental process in a fully automated way. A learning algorithm, described in detail in the following section, is used for assumption generation.

4 Algorithms

4.1 The L* Algorithm

The learning algorithm used by our approach was developed by Angluin [3] and was later improved by Rivest and Schapire [22]. In this paper, we will refer to the *improved* version by the name of the original algorithm, L*. L* learns an unknown regular language and produces a DFA that accepts it. Let U be an unknown regular language over some alphabet Σ. In order to learn U, L* needs to interact with a *Minimally Adequate Teacher*, from now on called *Teacher*. A Teacher needs to be able to answer correctly two types of questions from the algorithm. The first type is a *membership query*, consisting of a string $\sigma \in \Sigma^*$; the answer is *true* if $\sigma \in U$, and *false* otherwise. The second type of question is a *conjecture*, that is, a candidate DFA C whose language $\mathcal{L}(C)$ the algorithm believes to be identical to U. The answer is *true* if $\mathcal{L}(C) = U$. Otherwise the Teacher returns a counterexample, which is a string σ in the symmetric difference of $\mathcal{L}(C)$ and U.

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

Fig. 5. The L* Algorithm

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

In the following more detailed presentation of the algorithm, line numbers refer to L*'s illustration in Fig. 5. L* builds an observation table (S, E, T) where S and E are a set of prefixes and suffixes, respectively, both over Σ^*. In addition, T is a function mapping $(S \cup S \cdot \Sigma) \cdot E$ to $\{\text{true}, \text{false}\}$, where the operator "$\cdot$" is defined as follows. Given two sets of event sequences P and Q, $P \cdot Q = \{pq \mid p \in P \text{ and } q \in Q\}$, where pq represents the concatenation of the events sequences p and q. Initially, L* sets S and E to be $\{\lambda\}$ (line 1), where λ represents the empty string. Subsequently, it updates the function T by making membership queries so that it has a mapping for every string in $(S \cup S \cdot \Sigma) \cdot E$ (line 2). It then checks whether the observation table is *closed*, i.e., whether

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

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

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

The counterexample c is analyzed by L* to find a suffix e of c that witnesses a difference between $\mathcal{L}(C)$ and U; e must be such that adding it to E will cause the candidate to reflect this difference[4] (line 8). Once e has been added to E, the L* Algorithm iterates the entire process by looping around to line 2.

Characteristics of L*. Let M be the minimal automaton such that $\mathcal{L}(M) = U$. The L* algorithm is guaranteed to terminate with M as its last conjecture. The characteristic of L* that makes it particularly attractive for our work is that at any stage, the automata that it generates are *minimal*. In other words, if L* makes a conjecture C based on observation table (S, E, T), then any other DFA C' that is consistent with (S, E, T) but not equivalent to C contains *more* states than C. In addition, the conjectures made by L* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than M. If M has n states, L* makes at most $n-1$ incorrect conjectures. The number of membership queries made by L* is $\mathcal{O}\left(kn^2 + n \log m\right)$, where k is the size of the alphabet of the language U, 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.

4.2 Learning for Assume-Guarantee Reasoning

Assume a system $M_1 \parallel M_2$, and a property P that needs to be satisfied in the system. In the context of the compositional rule presented in Section 3, the learning algorithm is called to guess an assumption that can be used in the rule to prove or disprove P. An assumption with which the rule is guaranteed to return conclusive results is the *weakest assumption* A_w, which restricts the environment of M_1 no more and no less than is necessary for P to be satisfied. Assumption A_w describes exactly those traces over $\Sigma = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$ which, when simulated on $M_1 \parallel P_{err}$ cannot lead to state π. The language $\mathcal{L}(A_w)$ of the assumption contains at least *all* traces of M_2 abstracted to Σ that prevent M_1 from violating P.

Formally, A_w is such that, for any environment component M_E, $\langle true \rangle\ M_1 \parallel M_E\ \langle P \rangle$ if and only if $\langle true \rangle\ M_E\ \langle A_w \rangle$ [14]. In our framework, L* learns the traces of A_w through the iterative process described in Section 3. The process terminates as soon as compositional verification returns conclusive results, which is often before the weakest assumption A_w is computed by L*.

For L* to learn A_w, we need to provide a Teacher that is able to answer the two different kinds of questions that L* asks. Our approach uses model checking to implement such a Teacher.

Membership Queries. To answer a membership query for $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ in Σ^* the Teacher simulates the query on $M_1 \parallel P$. For clarity of presentation we will reduce such simulations to model checking, although we have implemented them more efficiently, directly as simulations. So for string σ, the Teacher first builds $A_\sigma = \langle Q, \alpha A_\sigma, \delta, q_0 \rangle$ where $Q = \{q^0, q^1, \ldots, q^n\}$, $\alpha A_\sigma = \Sigma$, $\delta = \{(q^i, a^{i+1}, q^{i+1}) \mid 0 \leq i < n\}$, and $q_0 = q^0$. The Teacher then model checks $\langle A_\sigma \rangle\ M_1\ \langle P \rangle$. If true is returned, it means that $\sigma \in \mathcal{L}(A_w)$, because M_1 does not violate P in the context of σ, so the Teacher returns true. Otherwise, the answer to the membership query is false.

Conjectures. Due to the fact that in our case the language $\mathcal{L}(A_w)$ that is being learned is prefix-closed, all conjectures returned by L* are also prefix-closed. Our framework transforms these conjectures into safety LTSs (see Section 2), which constitute the intermediate assumptions A_i.

[4] The procedure for finding e is beyond the scope of this paper, but is described in [22].

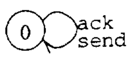

Fig. 6. A_1

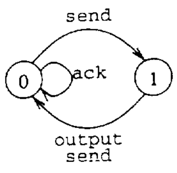

Fig. 7. A_2

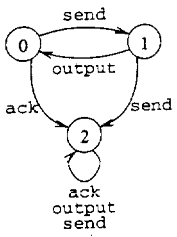

Fig. 8. A_3

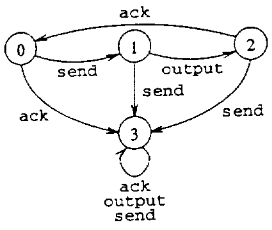

Fig. 9. A_4

In our framework, the first priority is to guide L* towards a conjecture that is strong enough to make Step 1 of the compositional rule return true. Once this is accomplished, the resulting conjecture may be too strong, in which case our framework guides L* towards a conjecture that is weak enough to make Step 2 return conclusive results about whether the system satisfies P. The way the Teacher that we have implemented reflects this approach is by using two Oracles and Counterexample Analysis to answer conjectures as follows:

- **Oracle 1** performs Step 1 in Fig. 4, i.e., it checks $\langle A_i \rangle\ M_1\ \langle P \rangle$. If this does not hold, the model checker returns a counterexample c. The Teacher informs L* that its conjecture A_i is not correct and provides $c \upharpoonright \Sigma$ to witness this fact. If, instead, $\langle A_i \rangle\ M_1\ \langle P \rangle$ holds, the Teacher forwards A_i to Oracle 2.
- **Oracle 2** performs Step 2 in Fig. 4 by checking $\langle true \rangle\ M_2\ \langle A_i \rangle$. If the result of model checking is true, the teacher returns true. Our framework then terminates the compositional verification because P has been proved on $M_1 \parallel M_2$ (according to the compositional rule). If model checking returns a counterexample, the Teacher performs some analysis to distinguish the underlying reason (see Section 3 and Fig. 4).
- **Counterexample Analysis** is performed by the Teacher in a way similar to that used for answering membership queries. Let c be the counterexample returned by Oracle 2. The Teacher computes $A_{c\upharpoonright\Sigma}$ and checks $\langle A_{c\upharpoonright\Sigma} \rangle\ M_1\ \langle P \rangle$. If true, it means that M_1 does not violate P in the context of c, so $c \upharpoonright \Sigma$ is returned by the Teacher as a counterexample for conjecture A_i. If the model checker returns false with some counterexample c', it means that P is violated in $M_1 \parallel M_2$, so there is no need for a more precise assumption to be generated. Our framework then appropriately combines c with c' in order to generate a counterexample for $\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$.

4.3 Example

Given components *Input* and *Output* as shown in Fig. 1 and the property *Order* shown in Fig. 2, we will check $\langle true \rangle\ Input \parallel Output\ \langle Order \rangle$ by using our approach. The alphabet of the assumptions that will be used in the compositional rule is $\Sigma = ((\alpha Input \cup \alpha Order) \cap \alpha Output) = \{\mathtt{send}, \mathtt{output}, \mathtt{ack}\}$.

As described, at each iteration L* updates its observation table and produces a candidate assumption whenever the table becomes closed. The first closed table obtained is shown in Table 1

Table 1. Mapping T_1

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

Table 2. Mapping T_2

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

and its associated assumption, A_1, is shown in Fig. 6. The Teacher answers conjecture A_1 by first invoking Oracle 1, which checks $\langle A_1 \rangle$ *Input* $\langle P \rangle$. Oracle 1 returns false, with counterexample $\sigma = \langle$`input, send, ack, input`$\rangle$, which describes a trace in $A_1 \parallel$ *Input* $\parallel P_{err}$ that leads to state π.

The Teacher therefore returns counterexample $\sigma \upharpoonright \Sigma = \langle$`send, ack`$\rangle$ to L*, which uses queries to update its observation table until it is closed. From this table, shown in Table 2, the assumption A_2, shown in Fig. 7, is constructed and then conjectured to the Teacher. This time, Oracle 1 reports that $\langle A_2 \rangle$ *Input* $\langle P \rangle$ is true, meaning the assumption is not too weak. The Teacher calls Oracle 2 to determine if $\langle true \rangle$ *Output* $\langle A_2 \rangle$. This is also true, so our algorithm reports that $\langle true \rangle$ *Input* $\parallel$ *Output* $\langle P \rangle$ holds.

This example did not involve weakening of the assumptions produced by L*, since the assumption A_2 was sufficient for the compositional proof. This will not always be the case. For example, let us substitute *Output* by *Output'* illustrated in Fig. 3, which allows multiple `send` actions to occur before producing `output`. The process will be identical to the previous case, until Oracle 2 is invoked by the Teacher for conjecture A_2. Oracle 2 returns that $\langle true \rangle$ *Output'* $\langle A_2 \rangle$ is false, with counterexample $\langle$`send, send, output`$\rangle$. The Teacher analyzes this counterexample and determines that in the context of this trace, *Input* does not violate P. The trace is returned to the L* Algorithm, which will weaken the conjectured assumption. The process involves two more iterations, during which assumptions A_3 (Fig. 8) and A_4 (Fig. 9), are produced. Using assumption A_4, which is the weakest assumption A_w, both Oracles report true, so our assume-guarantee framework reports that $\langle true \rangle$ *Input* $\parallel$ *Output'* $\langle P \rangle$ holds.

5 Discussion

5.1 Correctness

Theorem 1. *Given components M_1 and M_2, and property P, the algorithm implemented by our framework terminates and it returns true if P holds on $M_1 \parallel M_2$ and false otherwise.*

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

- **Correctness**. The Teacher in our framework uses the two steps of the compositional rule to answer conjectures. It only returns true when both steps return true, and therefore correctness is guaranteed by the compositional rule. Our framework reports an error when it detects a trace σ of M_2 which, when simulated on M_1, violates the property, which implies that $M_1 \parallel M_2$ violates P.
- **Termination.** At any iteration, our algorithm returns true or false and terminates, or continues by providing a counterexample to L*. By correctness of the L* Algorithm we are guaranteed that if L* keeps receiving counterexamples, it will eventually, at some iteration i, produce A_w. During this iteration, Step 1 will return true by definition of A_w. The Teacher will therefore apply Step 2, which will return either true and terminate, or a counterexample. This counterexample represents a trace of M_2 that is not contained in $L(A_w)$. Since, as discussed in Section 4, A_w is both necessary and sufficient, analysis of the counterexample will return false, and the algorithm will terminate. □

5.2 Practical Considerations

In our context, the languages queried by the L* Algorithm are prefix-closed. This is because our technique applies to purely safety properties; any finite prefix of a trace that satisfies such a property must also satisfy the property. Therefore, when for some string σ a membership query $\langle A_\sigma \rangle\ M_1\ \langle P \rangle$ returns false, we know that for any extension of σ the answer will also be false. We can thus improve the efficiency of the algorithm by reducing the cost of some of the membership queries that are answered by the Teacher. For example, in the observation table shown in Table1, the entry for $\langle$`output`$\rangle$ is false. The Teacher can return false for the queries $\langle$`output,ack`$\rangle$, $\langle$`output,send`$\rangle$, and $\langle$`output,output`$\rangle$ without invoking the model checker.

In the framework presented, membership queries, conjectures and counterexample analysis all involve model checking, which is performed on-the-fly. The assumptions that are used in these steps are increasing in size, and grow no larger than the size of A_w. In our experience, for well-designed systems, the interfaces between components are small. Therefore, assumptions are expected to be significantly smaller than the environment that they represent in the compositional rules. Although the L* algorithm needs to maintain an observation table, this table does not need to be kept in memory while the model checking is performed.

Note that our framework provides an "any time" [11] approach to compositional verification. If memory is not sufficient to reach termination, intermediate assumptions are generated, which may be useful in approximating the requirements that a component places on its environment in order to satisfy certain properties.

5.3 Extensions

Generalization. Our approach has been presented in the context of two components. Assume now that a system consists of n components $M_1 \parallel \cdots \parallel M_n$. The simplest way to generalize our approach is to group these components into two higher level components, and apply the compositional rules as already discussed. Another possibility is to handle the general case without computing the composition of any components directly. Our algorithm provides a way of checking $\langle true \rangle$ $M_1 \parallel M_2$ $\langle P \rangle$ in a compositional way. If M_2 consists of more than one component, our algorithm could be applied recursively for Step 2. This is an interesting future direction, in particular since the membership queries concentrate on a single component at a time. However, we need to further investigate how meaningful such an approach would be in practice.

Computing the Weakest Assumption. The L* Algorithm can also be used to learn the weakest possible assumption A_w that will prevent a component M_1 from violating a property P. This assumption will be generated without knowing M_2, the component M_1 interacts with. The only place in our assume-guarantee framework where M_2 is used is in Oracle 2, when the Teacher tries to determine if the Assumption generated is too strong. Oracle 2 can be replaced by a conformance checker, for example the W-Method [6], which is designed to expose a difference between a specification and an implementation. This will produce a set of sequences that are guaranteed to expose an error in the conjectured assumption if one exists. This approach to learning the weakest assumption is an *anytime* algorithm [11]. The sequence of intermediate assumptions conjectured by the teacher are approximate and become more refined the longer the L* Algorithm runs. As discussed previously, an approximate assumption can still be useful.

6 Experience

We implemented the assume-guarantee framework described above in the LTSA tool, and experimented with our approach in the analysis of the executive subsystem for the K9 Mars Rover, developed at NASA Ames. The executive receives flexible plans from a Planner, which it executes according to the plan language semantics. A plan is a hierarchical structure of actions that the Rover must perform. For increased autonomy, each action is associated with a number of state or temporal pre-, maintenance, and post-conditions, which must hold before, during, and on completion of the action execution, respectively.

The executive has been implemented as a multi-threaded system, where synchronization between threads is performed through mutexes and condition variables. The developer provided design documents for several versions of the system. These documents described the synchronization between components in an ad-hoc flowchart-style language. They looked very much like LTSs, which allowed us to translate them in a straightforward and systematic way into the input language of the LTSA.

One of the properties described by the developer refers to a subsystem of the executive consisting of two components: the main coordinating component named "Executive", and a component responsible for monitoring state conditions named "ExecCondChecker". The property places the following requirement on this subsystem, irrespective of the behavior of the subsystem's environment: for a specific variable of the ExecCondChecker shared with the Executive, if the Executive reads the value of the variable, then the ExecCondChecker should not read this value before the Executive clears it first.

We used our compositional verification framework to check this property on one of the newer versions of the model. We set M_1 = ExecCondChecker and M_2 = Executive. The experiment was conducted on a Pentium III 500 MHz with 1 Gb of memory running RedHat Linux 7.2 using Sun's Java SDK version 1.4.0_01. To check the property directly by composing the two modules with the property required searching 3,630 states and 34,653 transitions and took 0.535 seconds.

Table 3. Results of the Rover Example

Iteration	$\|A_i\|$	States	Transitions	Result
1 - Oracle 1	1	5	24	Too weak
2 - Oracle 1	2	268	1,408	Too weak
3 - Oracle 1	3	235	1,209	Too weak
4 - Oracle 1	5	464	2,500	Not too weak
4 - Oracle 2	5	32	197	Incompatible

Table 3 shows the results of using our assume-guarantee framework on this example, which took 8.639 seconds. The $|A_i|$ column gives the number of states of the assumptions generated. The States and Transitions columns give the number of states and transitions explored during the analysis of the assumption and the Result column gives the result of the analysis. When using Oracle 1 to determine if the Assumption was too weak, iterations 1-3 all determined that the assumption was too weak. In iteration 4, Oracle 1 determined that the learned assumption was not too weak. The assumption was then given to Oracle 2 that returned a counterexample which, when simulated on the ExecCondChecker, led to an error state. Thus, the assume-guarantee analysis concluded that the property does not hold. The largest analysis occurred when using Oracle 1 during iteration 4, and this required exploring fewer states than checking the property directly.

We also used the L* Algorithm to generate the weakest assumption that the ExecCondChecker makes on the Executive for the property described to be satisfied in this subsystem. The method described in [14] needed 24.623 seconds to generate the 6 state weakest assumption. The L* Algorithm, using the W-Method for Oracle 2, took 9.530 seconds to generate the 6 state assumption, although conformance testing kept running after that since it could not yet conclude if this was the weakest possible assumption.

Assumptions for Java Programs. We are currently working on an implementation of our approach in Java PathFinder (JPF) [23], a model checker for Java programs developed at NASA Ames. We have already developed a prototype with the assumption generation algorithm based on the W-Method, and have experimented with it on several Java programs. To generate assumptions for Java programs, no changes needed to be made to JPF, but we needed to add a method call at program points where actions of interest occurred. The Teacher calls JPF to answer both queries and conjectures.

7 Related Work

One way of addressing both the design and verification of large systems is to use their natural decomposition into components. Formal techniques for support of component-based design are gaining prominence, see for example [9, 10]. In order to reason formally about components in isolation, some form of assumption (either implicit or explicit) about the interaction with, or interference from, the environment has to be made. Even though we have sound and complete reasoning systems for assume-guarantee reasoning, see for example [7, 16, 18, 21] and more recently [24], it is always a mental challenge to obtain the most appropriate assumption [17].

It is even more of a challenge to find automated techniques to support this style of reasoning. The thread modular reasoning underlying the Calvin tool [12] is one start in this direction. In the framework of temporal logic, the work on Alternating-time Temporal Logic ATL (and transition systems) [1] was proposed for the specification and verification of open systems together with automated support via symbolic model checking procedures. The Mocha toolkit [2] provides support for modular verification of components with requirement specifications based on the ATL.

In previous work [14], we presented an algorithm for automatically generating the weakest possible assumption for a component to specify a required property. Although the motivation of that work is different, the ability to generate the weakest assumption can also be used to automate assume-guarantee reasoning. A disadvantage of that algorithm is that it does not compute partial results, meaning no assumption is obtained if the computation runs out of memory. This may happen if the state-space of the component is too large. Our approach generates assumptions incrementally and may terminate before A_w is computed. Moreover, even if it runs out of memory before reaching conclusive results, intermediate assumptions may be used to give some indication to the developer of the requirements that the component analyzed places on its environment.

The problem of generating an assumption for a component is similar to the problem of generating component interfaces to deal with intermediate state explosion in CRA. Several approaches have been defined for automatically abstracting a component's environment to obtain interfaces [4, 19]. These approaches do not address the issue of incrementally refining interfaces, as needed for carrying out an assume-guarantee proof.

Groce et al have used the L* Algorithm for Adaptive Model Checking [15]. In this work, the goal is to learn a model of a software system which can then be given to a model checker. This approach is similar to our approach for learning assumptions using a conformance checker and the L* Algorithm. To determine if the model accurately describes the system, a conformance checker is used, which is expensive in terms of time. Until an accurate model is obtained, this process cannot be used to show that a property is satisfied and can only be used to help the analyst discover a counterexample. In our approach, an inaccurate assumption can still be used to complete an assume-guarantee proof.

8 Conclusions

Although theoretical frameworks for sound and complete assume-guarantee reasoning have existed for decades, their practical impact has been limited because they involve non-trivial human interaction. In this paper, we presented a novel approach to automating such reasoning in an incremental fashion. Our approach uses a learning algorithm to generate and refine assumptions based on queries and counterexamples, in an iterative process. The process is guaranteed to terminate, and return true if a property holds in a system, and a counterexample otherwise. If memory is not sufficient to reach termination, intermediate assumptions are generated, which may be useful in approximating the requirements that a component places on its environment in order to satisfy certain properties.

One advantage of our approach is its generality. It relies on standard features of model checkers, and could therefore easily be introduced in any such tool. Moreover, the architecture of our framework is modular, so its components can easily be substituted by more efficient ones. It remains to be shown, of course, how useful our approach will prove in practice. However, our early experiments with real case studies provide strong evidence in favor of this line of research.

In the future we plan to investigate a number of topics including whether the learning algorithm can be made more efficient in our context; whether different algorithms would be more appropriate for generating the assumptions; whether we could benefit by querying a component and its environment at the same time, or by implementing more powerful compositional rules. An interesting challenge will also be to extend the types of properties that our framework can handle to include liveness, fairness, and timed properties.

Acknowledgements

The authors would like to thank Alex Groce for his help with the L* Algorithm, Willem Visser and Flavio Lerda for their help in understanding the inner workings of JPF, and Zhendong Su for useful discussions on this work.

References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Compositionality: The Significant Difference - An International Symposium*, pages 23–60, Sept. 1997.
2. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. of the Tenth Int. Conf. on Comp.-Aided Verification (CAV)*, pages 521–525, June 28–July 2, 1998.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.
4. S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. on Soft. Eng. and Methodology*, 5(4):334–377, Oct. 1996.
5. S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. on Soft. Eng. and Methodology*, 8(1):49–78, Jan. 1999.
6. T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. on Soft. Eng.*, SE-4(3):178–187, May 1978.
7. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. of the Fourth Symp. on Logic in Comp. Sci.*, pages 353–362, June 1989.
8. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
9. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. of the Eighth European Soft. Eng. Conf. held jointly with the Ninth ACM SIGSOFT Symp. on the Found. of Soft. Eng.*, pages 109–120, Sept. 2001.
10. L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *Proc. of the First Int. Workshop on Embedded Soft.*, pages 148–165, Oct. 2001.
11. T. Dean and M. S. Boddy. An analysis of time-dependent planning. In *Proc. of the Seventh National Conf. on Artificial Intelligence*, pages 49–54, Aug. 1988.
12. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. of the Eleventh European Symp. on Prog.*, pages 262–277, Apr. 2002.
13. D. Giannakopoulou, J. Kramer, and S. C. Cheung. Behaviour analysis of distributed systems using the Tracta approach. *Auto. Soft. Eng.*, 6(1):7–35, July 1999.
14. D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the Seventeenth IEEE Int. Conf. on Auto. Soft. Eng.*, Sept. 2002.
15. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. of the Eighth Int. Conf. on Tools and Alg. for the Construction and Analysis of Sys.*, pages 357–370, Apr. 2002.
16. O. Grumberg and D. E. Long. Model checking and modular verification. In *Proc. of the Second Int. Conf. on Concurrency Theory*, pages 250–265, Aug. 1991.
17. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. of the Tenth Int. Conf. on Comp.-Aided Verification (CAV)*, pages 440–451, June 28–July 2, 1998.

18. C. B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *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 *Proc. of the Third Int. Workshop on Tools and Alg. for the Construction and Analysis of Sys.*, pages 239–258, Apr. 1997.
20. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
21. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, New York, 1984. Springer-Verlag.
22. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, Apr. 1993.
23. W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the Fifteenth IEEE Int. Conf. on Auto. Soft. Eng.*, pages 3–12, Sept. 2000.
24. Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.