Assume-Guarantee Verification for Interface Automata

Michael Emmi[1,⋆], Dimitra Giannakopoulou[2], and Corina S. Păsăreanu[2]

[1] University of California, Los Angeles
mje@cs.ucla.edu
[2] NASA Ames Research Center
{Dimitra.Giannakopoulou,Corina.S.Pasareanu}@email.arc.nasa.gov

Abstract. Interface automata provide a formalism capturing the high level interactions between software components. Checking compatibility, and other safety properties, in an automata-based system suffers from the scalability issues inherent in exhaustive techniques such as model checking. This work develops a theoretical framework and automated algorithms for modular verification of interface automata. We propose sound and complete assume-guarantee rules for interface automata, and learning-based algorithms to automate assumption generation. Our algorithms have been implemented in a practical model-checking tool and have been applied to a realistic NASA case study.

1 Introduction

Modern software systems are comprised of numerous components, and are made larger through the use of software frameworks. Formal analysis of such systems naturally suffers from scalability issues. Modular analysis techniques address this with a "divide and conquer" approach: properties of a system are decomposed into properties of its constituents, each verified separately. Assume-guarantee reasoning [14, 21] is a modular formal analysis technique that uses assumptions when checking components in isolation. A simple assume-guarantee rule infers that a system composed of components M_1 and M_2 satisfies safety property P by checking that M_1 under assumption A satisfies P (*Premise 1*) and discharging A on the environment M_2 (*Premise 2*). Finding suitable assumptions can be non-trivial, and has traditionally been performed manually.

Previous work [8] has proposed a framework using learning techniques to automate assumption generation for the aforementioned rule; that work addresses safety property checking for Labeled Transition Systems (LTSs). LTSs interact through synchronization of shared actions, and have been used extensively in the analysis of high-level abstract systems, for example at the architecture level [19]. However, as LTSs do not distinguish between input/output actions that a component can receive/emit, respectively, they are often inadequate for more detailed analyses, testing, or test-case generation [23].

⋆ This work was completed during an Mission Critical Technologies Inc. internship at NASA Ames.

J. Cuellar and T. Maibaum (Eds.): FM 2008, LNCS 5014, pp. 116–131, 2008.

I/O automata [17] and interface automata [9] are formalisms that differentiate between the input and output actions of a component. The main distinguishing factor between the two formalisms is that I/O automata are required to be *input-enabled*, meaning they must be receptive at every state to each possible input action. For interface automata, some inputs may be illegal in particular states, i.e., the component is not prepared to service these inputs. Two components are said to be *compatible* if in their composition each component is prepared to receive any request that the other may issue.

Compatibility is an important property for the analysis of component-based systems [16]. In the case study we discuss in Section 7, compatibility checking uncovered subtle errors that were undetectable when components were modeled as LTSs. Consequently, parts of the model were unexplored, even though the system is deadlock-free. (Note: these particular errors are undetectable if components are assumed input-enabled, as I/O automata are.)

Checking compatibility and traditional safety properties of interface automata suffers from the inherent scalability issues appearing in model checking techniques. In this work we develop a theoretical framework with automated algorithms for modular verification of systems modeled as interface automata. Specifically, we provide the first sound and complete assume-guarantee rules for checking properties of interface automata. This includes rules for compatibility checking, traditional safety properties, and alternating refinement (the notion of refinement associated with interface automata [9]).

We define a construction that reduces compatibility and alternating refinement to checking error state reachability, by adding to each component transitions to error states. We provide algorithms that automate the application of the assume-guarantee rules by computing assumptions (we provide both a direct and a learning-based construction of the assumptions). Although we reduce compatibility checking to error detection, we cannot simply use the rules and frameworks from our previous work [8]; that work assumed error states to be introduced only by checking the property in *Premise 1* and discharging the assumption in *Premise 2*; in this work error states are also introduced by our reduction. We describe this further in Sections 5 and 6.

For a system where component compatibility has been established we also define an optimized assumption construction when checking traditional safety properties. Our new algorithms have been implemented in the LTSA model checking tool [18] and have been applied to a NASA case study.

2 Background

Labeled Transition Systems. A *labeled transition system* (LTS) A is a tuple $\langle Q, a_0, \alpha A, \delta \rangle$, where Q is a finite set of states, $a_0 \in Q$ is an initial state, αA is a set of observable actions called the *alphabet* of A, and $\delta \subseteq Q \times \alpha A \times Q$ is a transition relation. For readability, we write $a \xrightarrow{\alpha} a'$ when $\langle a, \alpha, a' \rangle \in \delta$, in which case we say that α is *enabled* at a, and that a' is a *destination* of α from a. A state a' is *reachable* from a if there exists $n \in \mathbb{N}$ and sequences $\langle a_i \rangle_{0 \leq i \leq n}$ and

$\langle \alpha_i \rangle_{0 \le i < n}$ with $a = a_0$ and $a' = a_n$ such that $a_i \xrightarrow{\alpha_i} a_{i+1}$ for $0 \le i < n$. The LTS A is *deterministic* if δ is a function (i.e., for all $a \in Q$ and $\alpha \in \alpha A$, $a \xrightarrow{\alpha} a'$ for at most one state $a' \in Q$), and is otherwise *non-deterministic*.

We use π to denote a special *error state* without enabled transitions. The *error completion* of A is defined to be the LTS $A^\pi = \langle Q \cup \{\pi\}, a_0, \alpha A, \delta' \rangle$ where δ' agrees with δ, and adds transitions $a \xrightarrow{\alpha} \pi$ for all states a where α is not enabled. We say A is *safe* when π is not reachable from the initial state.

Parallel Composition. The parallel composition operator $\|$ is (up to isomorphism) a commutative and associative operator on LTSs. Given LTSs $A = \langle Q_A, a_0, \alpha A, \delta_A \rangle$ and $B = \langle Q_B, b_0, \alpha B, \delta_B \rangle$, the composition $A \| B$ is an LTS with states $Q_A \times Q_B$,[1] initial state $\langle a_0, b_0 \rangle$, alphabet $\alpha A \cup \alpha B$, and a transition relation defined by the rules (including the symmetric versions):

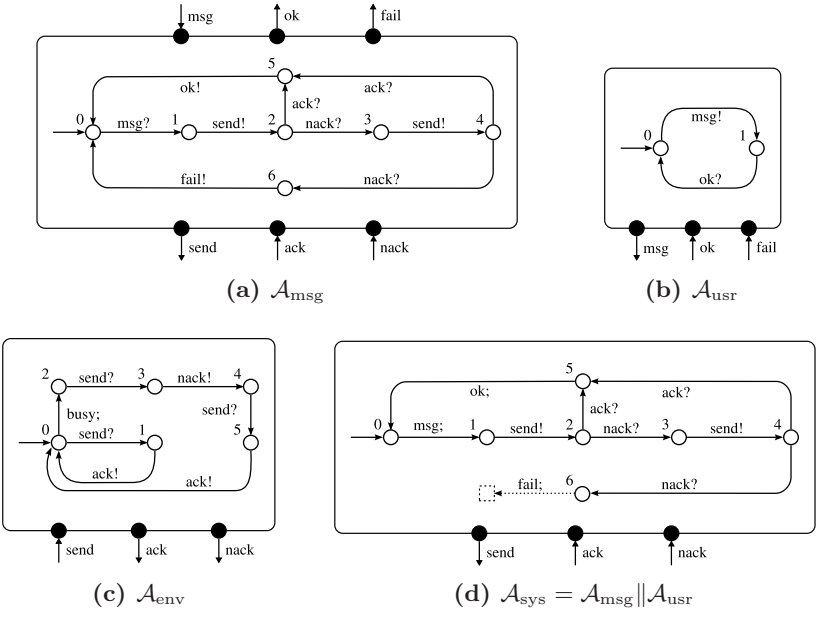
$$\frac{a \xrightarrow{\alpha} a' \quad a' \neq \pi \quad \alpha \notin \alpha B}{\langle a, b \rangle \xrightarrow{\alpha} \langle a', b \rangle}, \quad \frac{a \xrightarrow{\alpha} a' \quad b \xrightarrow{\alpha} b' \quad a', b' \neq \pi}{\langle a, b \rangle \xrightarrow{\alpha} \langle a', b' \rangle}, \text{ and } \frac{a \xrightarrow{\alpha} \pi}{\langle a, b \rangle \xrightarrow{\alpha} \pi}.$$

Traces. A *trace* t of *length* n on an LTS A is a finite sequence $\langle \alpha_i \rangle_{1 \le i \le n}$ of enabled actions on A starting from the initial state: formally, there exist states $a_1, \ldots, a_n \in Q$ such that $a_{i-1} \xrightarrow{\alpha_i} a_i$ for $1 \le i \le n$. The set of traces of A is called the *language* of A, denoted $\mathcal{L}(A)$. A trace t may also be viewed as an LTS, called a *trace LTS*, whose language consists of all prefixes of t (including t itself). As the meaning will be clear from the context, we will use t to denote both a trace and a trace LTS. We write $t{\downarrow_\Sigma}$ for the trace obtained from t by removing every occurrence of an action outside of Σ.

Safety properties. We call a deterministic LTS without the state π a *safety LTS* (any non-deterministic LTS can be made deterministic with the standard algorithm for automata). A safety property P is specified as a *safety LTS* whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP. An LTS M satisfies P, written $M \models P$, if and only if for every trace t of M, $t{\downarrow_{\alpha P}}$ is a trace of P. Note that $M \models P$ can be checked algorithmically by searching for a trace to π in M composed with the error completion of P.

The L* learning algorithm. Our algorithms for automating assume-guarantee reasoning use the L* algorithm for learning appropriate assumptions. L* was developed by Angluin [3] and later improved by Rivest and Schapire [22]. To synthesize an automaton for a regular language U over alphabet Σ, the algorithm interacts with a "black box", called a *teacher*, who answers questions about U. The teacher answers *membership queries* (given a string s, is $s \in U$?), and *refinement queries* (given an automaton A over Σ, does $\mathcal{L}(A) = U$?). We henceforth refer to the latter query as a *conjecture*, and the former simply as a *query*. If the conjectured automaton A's language is not U, the teacher is obligated to produce a *counterexample trace* in the symmetric difference of $\mathcal{L}(A)$ and U. This algorithm is guaranteed to terminate with a minimal automaton for U which has at most $n + 1$ states, where n is the number of incorrect conjectures.

[1] Each state $\langle a, \pi \rangle$ or $\langle \pi, b \rangle$ in the composition is identified with π.



(a) $\mathcal{A}_{\text{msg}}$ **(b)** $\mathcal{A}_{\text{usr}}$

(c) $\mathcal{A}_{\text{env}}$ **(d)** $\mathcal{A}_{\text{sys}} = \mathcal{A}_{\text{msg}} \| \mathcal{A}_{\text{usr}}$

Fig. 1. A messaging system comprised of three components (a, b, and c). The dotted transition to the error state π in (d) originates from the *error completed* automaton $\mathcal{A}^{?}_{\text{usr}}$ (explained in Sec.3.3), and survives in the composition $\mathcal{A}^{?}_{\text{msg}} \| \mathcal{A}^{?}_{\text{usr}}$. We omit other transitions due to error completion for readability.

3 Interface Automata (IA)

Definition 1. *An* interface automaton A *is a tuple* $\langle Q_A, I_A, \alpha A^{\text{I}}, \alpha A^{\text{O}}, \alpha A^{\text{H}}, \delta_A \rangle$ *where* Q_A *is a set of states,* $I_A \in Q_A$ *is the initial state,* αA^{I}*,* αA^{O}*, and* αA^{H} *are respectively disjoint* input*,* output*, and* internal alphabets *(we define* $\alpha A = \alpha A^{\text{I}} \cup \alpha A^{\text{O}} \cup \alpha A^{\text{H}}$*), and* $\delta_A \subseteq Q_A \times \alpha A \times Q_A$ *is a transition relation.*

Our running example of composable interface automata is borrowed from [9].

Example 1. The automaton $\mathcal{A}_{\text{msg}}$ (Fig. 1a) transmits messages over a lossy communication channel. The input actions `msg`, `ack`, and `nack` (resp., `send`, `ok`, and `fail`) are depicted by incoming (resp., outgoing) arrows to the enclosing box, and question (resp., exclamation) marks on edge labels. Internal actions (see Fig. 1c) do not appear on the interface boundaries, and suffixed by semicolons on edge labels.

The semantics of interface automata are defined here by reduction to labeled transition systems (LTSs). In particular, given an interface automaton $\langle Q_A, I_A, \alpha A^{\text{I}}, \alpha A^{\text{O}}, \alpha A^{\text{H}}, \delta_A \rangle$, $\text{lts}(A)$ is the LTS $\langle Q_A, I_A, \alpha A, \delta_A \rangle$. We lift the semantics from LTSs by writing $a \xrightarrow{\alpha} a'$ when $\text{lts}(A)$ has such a move, we say A is *(non-) deterministic* when $\text{lts}(A)$ is, and an action α is *enabled* in a state a when it is in $\text{lts}(A)$. The *traces* and the *language* of A are defined similarly.

For the remainder of this section we fix A and B to be interface automata.

Definition 2. *A and B are* composable *when their* signatures *do not conflict, i.e.* $\alpha A^{\mathrm{I}} \cap \alpha B^{\mathrm{I}} = \alpha A^{\mathrm{O}} \cap \alpha B^{\mathrm{O}} = \alpha A^{\mathrm{H}} \cap \alpha B = \alpha A \cap \alpha B^{\mathrm{H}} = \emptyset$.

Example 2. The "user" component of Figure 1b expects that the message-sending component of Figure 1a will never encounter `fail`ure. This implicit assumption is a key feature of IAs; its expressed here by the lack of a `fail?`-labeled edge from state 1 of $\mathcal{A}_{\mathrm{usr}}$.

The communication channel "environment" of Figure 1c either transmits a message on the first attempt, or delays on the first attempt, and transmits on the second. The internal action `busy` is not observed by other components. Both $\mathcal{A}_{\mathrm{usr}}$ and $\mathcal{A}_{\mathrm{env}}$ are composable with the $\mathcal{A}_{\mathrm{msg}}$, albeit on disjoint interfaces.

Note that composable IAs need not have any common actions, but each common action must be an input of one and an output of the other. We identify the set of common actions with $\alpha\mathsf{Shared}(A, B)$.

Definition 3. *When A and B are composable, the* composition *of A and B, written* $A \| B$*, is the interface automaton* $C = \langle Q_C, I_C, \alpha C^{\mathrm{I}}, \alpha C^{\mathrm{O}}, \alpha C^{\mathrm{H}}, \delta_C \rangle$*, where* $\langle Q_C, I_C, \alpha C, \delta_C \rangle = \mathrm{lts}(A) \| \mathrm{lts}(B)$*, and the alphabet is partitioned as*

- $\alpha C^{\mathrm{I}} = \alpha A^{\mathrm{I}} \cup \alpha B^{\mathrm{I}} \setminus \alpha\mathsf{Shared}(A, B)$,
- $\alpha C^{\mathrm{O}} = \alpha A^{\mathrm{O}} \cup \alpha B^{\mathrm{O}} \setminus \alpha\mathsf{Shared}(A, B)$, *and*
- $\alpha C^{\mathrm{H}} = \alpha A^{\mathrm{H}} \cup \alpha B^{\mathrm{H}} \cup \alpha\mathsf{Shared}(A, B)$.

The composition of an IA A and an LTS T is an IA extending A by substituting $\mathrm{lts}(A) \| T$ for $\mathrm{lts}(A)$.

Example 3. Since the constituents of $\mathcal{A}_{\mathrm{sys}}$ (Fig. 1d) synchronize on `fail`, and $\mathcal{A}_{\mathrm{usr}}$ `fail`ure never occurs, there are no transitions from state 6 in the composition. The signature of $\mathcal{A}_{\mathrm{sys}}$ does not mention common actions of $\mathcal{A}_{\mathrm{msg}}$ and $\mathcal{A}_{\mathrm{usr}}$ which have been internalized.

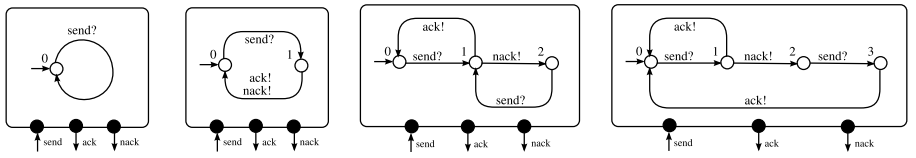
3.1 Compatibility

Although the semantics of a single IA is the same as its underlying LTS, the distinction between input and output actions results in a more stringent behavioral specification between components that cannot be checked for LTSs.

Definition 4. *Given two composable automata A and B, a state* $\langle a, b \rangle$ *of* $A \| B$ *is* illegal *if some action* $\alpha \in \alpha\mathsf{Shared}(A, B)$ *is an enabled output action in a (resp., b), but a disabled input action in b (resp., a).*

Example 4. State 6 of $\mathcal{A}_{\mathrm{sys}}$ (Fig. 1d) is illegal, since `fail` is an enabled output in state 6 of $\mathcal{A}_{\mathrm{msg}}$ and a disabled input in state 1 of $\mathcal{A}_{\mathrm{usr}}$.

Definition 5. *The automaton A is* closed *when* $\alpha A^{\mathrm{I}} = \alpha A^{\mathrm{O}} = \emptyset$*, and is otherwise* open.



(a) $\mathcal{C}_1$ **(b)** $\mathcal{E}_{\text{bad}}$ **(c)** $\mathcal{C}_3$ **(d)** $\mathcal{E}_{\text{good}}$

Fig. 2. Four environments for $\mathcal{A}_{\text{sys}}$ (Fig. 1d)

The optimistic notion of compatibility between IAs [9] is associated with the existence of illegal states in their composition. An open automaton with illegal states is not necessarily incompatible with its environments, since illegal states may not be reachable in the composition.

Definition 6. *When $A\|B$ is closed, A and B are said to be* compatible*, written $A \sim B$, if $A\|B$ does not have reachable illegal states.*

3.2 Refinement

For convenience we will write $\alpha \in \text{I-Enabled}_A(a)$ (resp., $\alpha \in \text{O-Enabled}_A(a)$) when α is an enabled input (resp., output) action in a. The *internal-closure* of a, written $\text{H-Closure}_A(a)$, is the set of states reachable from a via internal actions. An *externally observable move*, denoted $a \overset{\alpha}{\rightsquigarrow} a'$, exists when $a_1 \overset{\alpha}{\rightarrow} a_2$ for $a_1 \in \text{H-Closure}_A(a)$ and $a' \in \text{H-Closure}_A(a_2)$, in which case we say a' is an *external destination* from a by α. An action α is an *externally enabled* input (resp., output) in a, written $\alpha \in \text{I-ExtEn}_A(a)$ (resp., $\alpha \in \text{O-ExtEn}_A(a)$), if α is enabled in all (resp., some) states of $\text{H-Closure}_A(a)$.

Definition 7. *A binary relation $\preceq \subseteq (Q_A \times Q_B)$ is an* alternating simulation *from A to B if for all $a \in Q_A$ and $b \in Q_B$ such that $a \preceq b$:*

(1) $\text{I-ExtEn}_A(a) \supseteq \text{I-ExtEn}_B(b)$.
(2) $\text{O-ExtEn}_A(a) \subseteq \text{O-ExtEn}_B(b)$.
(3) For all actions $\alpha \in \text{O-ExtEn}_A(a) \cup \text{I-ExtEn}_B(b)$ and states a' such that $a \overset{\alpha}{\rightsquigarrow} a'$, there exists a state b' such that $b \overset{\alpha}{\rightsquigarrow} b'$ and $a' \preceq b'$.

We say A refines *B, written $A \preceq B$, if $\alpha A^{\text{I}} \supseteq \alpha B^{\text{I}}$, $\alpha A^{\text{O}} \subseteq \alpha B^{\text{O}}$ and there exists an alternating simulation $\preceq'$ from A to B such that $\langle I_A, I_B \rangle \in \preceq'$.*

Example 5. Figure 2 gives four automata with signatures matching $\mathcal{A}_{\text{env}}$'s (Fig. 1c). One can easily check that $\mathcal{A}_{\text{env}} \preceq \mathcal{E}_{\text{bad}}, \mathcal{E}_{\text{good}}$, but $\mathcal{A}_{\text{env}} \not\preceq \mathcal{C}_1, \mathcal{C}_3$.

The following refinement properties are known [9].

Theorem 1. *The alternating simulation relation over interface automata is reflexive and transitive.*

Theorem 2. *Let A, B, and C be interface automata such that B and C are composable, and $\alpha B^{\mathrm{I}} \cap \alpha C^{\mathrm{O}} \subseteq \alpha A^{\mathrm{I}} \cap \alpha C^{\mathrm{O}}$. If $A \sim C$ and $B \preceq A$, then $B \sim C$ and $B \| C \preceq A \| C$.*

3.3 Checking Compatibility and Refinement

We reduce compatibility and refinement checking to model checking on automata completed with error states.

Definition 8. *The* input (resp., output) error completion of A, *denoted $A^?$ (resp., $A^!$), extends A with the state π and the transition $a \xrightarrow{\alpha} \pi$ whenever α is a disabled input (resp., output) action at a.*

Supposing that A and B are *composable* but incompatible, there must exist a transition to π in $A^? \| B^?$, since either A or B performs an action which the other does not anticipate, causing either $B^?$ or $A^?$, respectively, to move to π. Likewise, if A and B are of the same signature but the behaviors of A are not contained within those of B, then either A performs some output action which B cannot, in which case $B^!$ moves to π, or B performs some input action which A cannot, in which case $A^?$ moves to π.

Theorem 3 (Checking Compatibility). *Let $A \parallel B$ be a closed automaton. Then $A \sim B$ if and only if π is not reachable in $A^? \| B^?$.*

Theorem 4 (Checking Refinement). *Let A and B be safe interface automata with matching signatures such that B is deterministic. $A \preceq B$ if and only if π is not reachable in $A^? \| B^!$.*

We omit the proofs in the interest of space.

Example 6. $\mathcal{A}_{\mathrm{msg}}$, $\mathcal{A}_{\mathrm{usr}}$, and $\mathcal{E}_{\mathrm{bad}}$ (Figs. 2b, 1a, and 1b) are incompatible since π is reachable in the composition of $\mathcal{A}^?_{\mathrm{msg}}$, $\mathcal{A}^?_{\mathrm{usr}}$, and $\mathcal{E}^?_{\mathrm{bad}}$ by the sequence msg; send; nack; send; nack; fail; (see Fig. 1d). On the other hand, $\mathcal{A}_{\mathrm{env}}$ (Fig. 1c) does not refine $\mathcal{C}_1$ (Fig. 2a) since π is reachable in the composition of $\mathcal{A}^?_{\mathrm{env}}$ and $\mathcal{C}^!_1$ by the sequence send; ack;.

4 Assume-Guarantee Reasoning for Interface Automata

Although in the simple setting of Example 5 the system and environment are relatively small, their composition could, in general, be very complex. We then seek to find a smaller environment model which is descriptive enough to prove the absence of error states in the composition, in the case that there are none.

Figure 3 introduces assume-guarantee rules for reasoning with interface automata. Since composition and compatibility are only defined for composable interface automata, we'll henceforth assume that the automata said to take part in these relations are composable. For the remainder of this section, the symbols M_1, M_2, S, and A range over interface automata, and P denotes a property specified by a safety LTS. Completeness, in the present setting, means that an assumption for use in the premises of rule exists whenever the conclusion holds.

	IA-Compat	IA-Prop	IA-AltRef
Premise 1	$M_1 \sim A$	$M_1 \| A \models P$	$M_1 \sim A \wedge M_1 \| A \preceq S$
Premise 2	$M_2 \preceq A$	$M_2 \models \text{lts}(A)$	$M_2 \preceq A$
Conclusion	$M_1 \sim M_2$	$M_1 \| M_2 \models P$	$M_1 \sim M_2 \wedge M_1 \| M_2 \preceq S$

Fig. 3. Assume-guarantee rules for interface automata. Rule IA-Compat gives a modular characterization of compatibility; IA-Prop is the IA instantiation of the classical safety rule [8]; IA-AltRef modularly establishes alternating refinement with respect to a high-level specification S.

Theorem 5. IA-Compat, IA-Prop, IA-AltRef *are sound and complete.*

Soundness of Rule IA-Compat is an immediate consequence of Theorem 2. Soundness of Rule IA-Prop is guaranteed by the soundness of the original rule for transition systems [11, 8]. Soundness of Rule IA-AltRef is guaranteed by Theorems 1 and 2. Completeness, for any of the rules, follows directly by replacing A with M_2.

Since Rule IA-AltRef is only meaningful for open systems, and the current work deals with closed systems, its study is deferred for future work.

5 Weakest Assumptions

A central notion in the work on automatic assumption generation for assume-guarantee rules is the construction of a "weakest assumption". For a rule and a given alphabet (the communication alphabet between M_1 and M_2) the weakest assumption A_W is such that, for any assumption A that makes the premises of a rule hold, A necessarily refines A_W, i.e., A_W is as abstract as possible.

Lemma 1. *Given a rule and its associated weakest assumption A_W, the premises of the rule hold for A_W if and only if the conclusion of the rule holds.*

Therefore, to automate assume-guarantee reasoning based on a rule, it is sufficient to build the corresponding A_W and to use it when checking the premises of the rule. We begin with a description of a *direct* construction of the weakest assumptions for various rules. Since this construction involves expensive determinization, we also define in the next section algorithms that learn the traces of A_W *as needed*, using L*. Let us first introduce the following definition.

Definition 9. *The* mirror *of A, written* $\mathsf{Mirror}(A)$*, is an automaton identical to A, except for a symmetric alphabet partitioning:* $\alpha\mathsf{Mirror}(A)^{\mathrm{I}} = \alpha A^{\mathrm{O}}$*,* $\alpha\mathsf{Mirror}(A)^{\mathrm{O}} = \alpha A^{\mathrm{I}}$*, and* $\alpha\mathsf{Mirror}(A)^{\mathrm{H}} = \alpha A^{\mathrm{H}}$*.*

Rule IA-Compat**.** The weakest assumption A_W of an interface automaton M_1 is an interface automaton with: $\alpha A_W^{\mathrm{I}} = \alpha M_1^{\mathrm{O}}$, $\alpha A_W^{\mathrm{O}} = \alpha M_1^{\mathrm{I}}$, and $\alpha A_W^{\mathrm{H}} = \emptyset$. It is constructed from $M_1^?$ as follows: 1) determinize[2]; 2) remove all transitions to

[2] The determinization of our interface automata identifies sets containing π, with π.

π (intuitively, all the inputs that M_1 does not accept lead to error in $M_1^?$, so its environment should not provide these), and 3) mirror the resulting automaton.

The above construction is similar to the property extraction step in the generation of assumptions when checking safety properties of LTSs [11]. However in [11], a completion step adds transitions to an (accepting) sink state. For compatibility, such completion of $M_1^?$ would add outputs ($M_1^?$ is input complete), and would force the environment to accept more inputs than necessary, i.e., the obtained assumption would not be the *weakest*. Also, the extra mirroring step here is needed to obtain a correct representation of the environment.

Rule IA-Prop**.** As mentioned, Rule IA-Propis the same as the original rule for LTSs [11,8], so the same notion of a weakest assumption applies. However, in the context of interface automata, the knowledge that $M_1 \sim M_2$ holds may be used for defining a weaker assumption with fewer states than the one in previous work [8]. Let A_W be the weakest assumption for M_2 with respect to M_1 and P, as previously defined [11]. A_W is an interface automaton with: $\alpha A_W^{\mathrm{I}} = \alpha M_1^{\mathrm{O}} \cap \alpha M_2^{\mathrm{I}}$, $\alpha A_W^{\mathrm{O}} = \alpha M_1^{\mathrm{I}} \cap \alpha M_2^{\mathrm{O}}$, and $\alpha A_W^{\mathrm{H}} = \alpha P \cap \alpha M_2^{\mathrm{H}}$. Assume that we already checked $M_1 \sim M_2$ (using e.g. Rule IA-Compat) and it holds. We build assumption A_C such that

$$\mathcal{L}(A_C) = \mathcal{L}(A_W) \setminus \{t \mid \exists u \in (\alpha M_1^{\mathrm{O}})^+ \text{ s.t. } tu \notin \mathcal{L}(A_W)\}.$$

In other words, the assumption does not restrict the behaviors of M_1 by non-acceptance of M_1's outputs (i.e. the assumption is compatible with M_1). A_C is constructed from $M_1 \| P^\pi$ using the same steps of previous work [11], with the following differences. Since $M_1 \sim M_2$ holds, backwards error propagation is performed along *output* transitions *in addition to* internal transitions. Therefore A_C has potentially fewer states than A_W. Moreover, the resulting automaton needs to be mirrored since we are dealing with interface automata.

Lemma 2. *Let $M_1 \sim M_2$, and let A_W and A_C be the assumptions defined above. Then $M_1 \| A_C \models P \wedge M_2 \models \mathrm{lts}(A_C) \iff M_1 \| M_2 \models P$.*

The above Lemma establishes that A_C, which has at most as many states as A_W, is the weakest assumption. For Rule IA-Prop, we will henceforth use the term weakest assumption (A_W) to refer to A_C.

6 Learning-Based Assume-Guarantee Reasoning

We develop iterative techniques based on L* [3, 22] to check $M_1 \parallel M_2$ compositionally, by automatically computing the weakest assumptions for Rules IA-Compat and IA-Prop. We provide L* with *teachers* using error state reachability analysis to answer membership queries and conjectures. We use L* conjectures as an assumption to check the premises of the rules (using an *oracle* for each premise). When both oracles return `OK` then the premise is satisfied, and the analysis terminates. Failure of Premise 1 gives L* a counterexample to refine its conjecture, while failure of Premise 2 either corresponds to a real

system violation (and the analysis terminates) or gives L* a counterexample for refinement.

The techniques presented here are similar in spirit to existing techniques [8], but must be significantly more complex to address the non-trivial notions of compatibility and alternating refinement (the techniques learn the traces of the new weakest assumptions that we defined in the previous section). Indeed existing algorithms [8] check a strictly weaker property—a consequence of not distinguishing input from output actions.

We make use of the following auxiliary procedures in our L* teachers.

`simulate(M,t)` returns a set of `M`-states to which `t` is a trace, or π with the shortest prefix of `t` tracing to π, or $\emptyset$ with the shortest prefix of `t` which is not a trace of `M`.

`analyze(M‖N)` returns `ERROR(M)` (resp., `ERROR(N)`) when π is reachable in an `M`-component (resp., `N`-component) of the composition, and otherwise `OK`.

Algorithm for compatibility. In our algorithm for obtaining the weakest assumption for Rule IA-Compat, we use the procedures in Fig. 4 for answering queries and conjectures. Oracle 1 uses Theorem 3 to check $M_1 \sim A$, while Oracle 2 uses Theorem 4 to check $M_2 \preceq A$. If either case fails, the L* teacher emits a counterexample trace witnessing such failure. For the case of Oracle 2, further analysis determines whether the trace represents an actual incompatibility between M_1 and M_2, or the assumption needs further refinement. If the trace turns out to be an error in M_1, or an error in M_2 which does not block M_1, $M_1 \not\sim M_2$; otherwise the trace is not a feasible incompatibility of the system, so the assumption needs refinement.

procedure Query-IA-Compat(t)
1: (states,_) := simulate($M_1^?$, t)
2: **if** $\pi \in$ states **then**
3: **return** NO
4: **else if** states = $\emptyset$ **then**
5: **return** NO
6: **end if**
7: **return** YES

procedure Oracle1-IA-Compat(A)
1: (result,t) := analyze($M_1^? \| \mathtt{A}^?$)
2: t := t $\downarrow_{\alpha \mathtt{A}}$
3: **if** result = ERROR **then**
4: **return** REFINE(t)
5: **else**
6: **return** OK
7: **end if**

procedure Oracle2-IA-Compat(A)
1: (result,t) := analyze($M_2^? \| \mathtt{A}^!$)
2: **if** result = OK **then**
3: **return** OK
4: **end if**
5: t := t $\downarrow_{\alpha \mathtt{A}}$
6: (states,t') := simulate($M_1^?$, t)
7: **if** $\pi \in$ states **then**
8: **return** INCOMPATIBLE
9: **else if** states = $\emptyset$
or result = ERROR(A) **then**
10: **return** REFINE(t')
11: **else if** result = ERROR($M_2^?$) **then**
12: **return** INCOMPATIBLE
13: **end if**

Fig. 4. The L* Teacher for Rule IA-Compat

Example 7. Our teacher for Rule IA-COMPAT receives a total of four conjectures when M_1 and M_2 are given by $\mathcal{A}_{\text{sys}}$ and $\mathcal{A}_{\text{env}}$ (Figs. 1d, 1c), respectively. The first and second conjectures are the automata $\mathcal{C}_1$ and $\mathcal{E}_{\text{bad}}$ (Figs. 2a, 2b), respectively, which Example 6 shows violate Premises 2 and 1 of Rule IA-COMPAT. The third conjecture $\mathcal{C}_3$ (Fig. 2c) is also incompatible with $\mathcal{A}_{\text{sys}}$, since the cycle formed between states 1 and 2 allow an arbitrary number of consecutive `nack`s. The final conjecture, $\mathcal{E}_{\text{good}}$ (Fig. 2d) is refined by $\mathcal{A}_{\text{env}}$ and adequate enough to prove compatibility with $\mathcal{A}_{\text{sys}}$.

Algorithm for property safety. Although the LTS safety checking algorithm of [8] can soundly be applied to interface automata, the knowledge about compatibility between automata allows us to develop an optimized algorithm for checking property safety. To do so, we must first consider controllability.

Definition 10. *Let A be an interface automaton, and $t \in \alpha A$ a word. The controllable prefix of t (w.r.t. A), written* $\text{ControlPref}_A(t)$ *is the longest prefix of t ending in an output or internal action of A.*

Intuitively, the controllable prefix corresponds to the period of time a particular component is dictating a trace. In our optimized safety checking algorithm, we consider incompatibilities arising from any externally controlled sequence appended to the end of the control prefix, not just the particular uncontrollable suffix of the trace. We extend `simulate` to account for this behavior.

`ext_simulate(M,t)` extends `simulate(M,t)` by additionally returning `ERROR` together with the shortest trace to π, when such an error can be reached via a prefix of `t` followed by a sequence of uncontrollable actions.

The key difference between our algorithm (that uses the procedures in Fig. 5 for queries and conjectures) and previous work [8] is that queries here replace the standard automata simulation with `ext_simulate`. The extension accounts for the fact that error states are propagated along output transitions in addition to internal ones in $M_1 \| P^\pi$ (recall, these actions correspond to ones that A cannot control). Moreover, when an assumption must be refined, the teacher returns to L* the controllable prefix of the counterexample that is obtained from reachability analysis (see line 4 in ORACLE 1 and line 5 in ORACLE 2).

Correctness. Granting Lemma 3, we are guaranteed that either L* terminates with the weakest assumption, or that $\mathcal{R}$ does not hold. We omit the proof in the interest of space.

Lemma 3. *Let $\mathcal{R}$ be an assume-guarantee rule in the context of interface automata M_1 and M_2 (and if applicable the safety LTS P), and let A_W be the weakest assumption for $\mathcal{R}$. Then*

(i) QUERY-$\mathcal{R}$`(t)` *returns* `YES` *iff* `t` *is a trace of A_W, and*
(ii) CONJECTURE-$\mathcal{R}$`(A)` *returns*
  (a) `OK` *iff the conclusion of $\mathcal{R}$ holds,*
  (b) `INCOMPATIBLE` *or* `PROPERTY_VIOLATION` *if the conclusion of $\mathcal{R}$ does not hold, and otherwise*
  (c) a trace in the symmetric difference of $\mathcal{L}$(`A`) and $\mathcal{L}(A_W)$.

```
procedure QUERY-IA-PROP(t)
 1: (states,_) :=
      ext_simulate(M1‖P^π,
        ControlPref_A(t))
 2: if π ∈ states then
 3:    return NO
 4: end if
 5: return YES

procedure ORACLE1-IA-PROP(A)
 1: (result,t) := analyze(M1‖P^π‖A)
 2: t := t ↓αA
 3: if result = ERROR then
 4:    return REFINE(ControlPref_A(t))
 5: else
 6:    return OK
 7: end if

procedure ORACLE2-IA-PROP(A)
 1: (result,t) := analyze(M2‖A^π)
 2: t := t ↓αA
 3: if result = ERROR then
 4:    if QUERY-IA-PROP(t) = YES then
 5:       return REFINE(ControlPref_A(t))
 6:    else
 7:       return PROPERTY VIOLATION
 8:    end if
 9: else
10:    return OK
11: end if
```

Fig. 5. The L* Teacher for Rule IA-PROP

7 Experience

The ARD Case Study. Autonomous Rendezvous and Docking (ARD) describes a spacecraft's capability of locating and docking with another spacecraft without direct human guidance. In the context of a NASA project, we were given UML statecharts describing an ARD protocol at a high level of abstraction, along with some required properties in natural language, for example: *"good values from at least two sensors are required to proceed to the next mode."*

The model consists of sensors (GPS, StarPlanetTracker, InertialNavigation), used to estimate the current position, velocity, etc. of the spacecraft, and "modes" that constitute a typical ARD system (see Figure 6). The ARD software moves sequentially through the modes, exhibiting different behavior in each. The OrbitalState component takes sensor readings and reports to the mode-related software whether it obtained good readings from enough sensors to calculate a reasonable spacecraft state estimate for ARD. The ARD software enables or disables the orbital state (via `enableNavigation` and `disableNavigation` actions, respectively), reads the current estimate (via the `read` action), and requests for an update of the state estimation (via `refresh`). The sensors may also fail, as observed through the `failed` actions.

Study Set-Up. We have extended the LTSA tool [18] to provide support for expressing interface automata and also with algorithms for (1) compatibility and refinement checking and (2) learning-based compositional verification for interface automata for Rules IA-COMPAT and IA-PROP. In an initial study of the ARD system we translated the UML statecharts and their expected properties into LTSs for LTSA; for the current study we refined these LTSs into Interface Automata, resulting in approximately 1000 lines of input code for the LTSA.

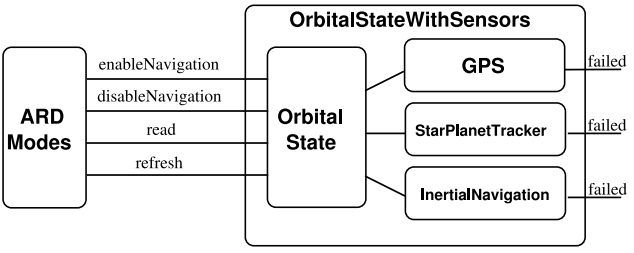


Fig. 6. Architecture of ARD protocol

Model Incompatibilities. Checking compatibility in the ARD system uncovered subtle errors that were undetected in our original study using simple LTS analysis. One incompatibility concerned the OrbitalState sub-system, which is made up of an `Estimator` and three `Counter`s; the counters record the number of good readings obtained for position, velocity and attitude; estimates are updated through the `refresh` action. Checking compatibility between the `Estimator` and the `Counter`s uncovered an error that prevented a significant portion of the system from being exercised.

We illustrate this error with a simplified OrbitalState (see Figures 7 and 8) that estimates position only with a single sensor. When the estimator refreshes, it gets a reading from the sensor. If the reading is `sensor_good`, the estimator `increment`s the counter. The estimator then gets the counter value `get_val`; the estimate becomes `est_poor` when this value is 0, and is otherwise `est_good`. In intermediate states the estimate is `est_updating`. Incompatibility between the two components is illustrated by trace: `<refresh, sensor_good, increment, get_val, return.1, refresh, sensor_good, increment>`, where the estimator tries to increment the counter at its max value. The error occurs because the counter is not reset when it should be (i.e., after each refresh operation). If LTSs were used instead of interface automata, incrementing a counter at its max value would simply not be possible in the composition. Despite this fact, the system would not deadlock, because of the self-loops in the estimator. One could write liveness properties to ensure that certain behaviors are always possible in the system, but it is much simpler and less costly to check for a pre-defined notion of compatibility that does not require specification.

Results and Discussion. After correcting the incompatibilities in the OrbitalState component, we applied our compositional algorithms system-wide level. We divided the ARD model into two components: ARDModes (representing M_1) and OrbitalStateWithSensors (representing M_2); we checked that compatibility and a safety property (the sensor quality property, mentioned earlier) hold. The results in Table 1 compare running LTSA for non-compositional verification with learning-based assume-guarantee reasoning. For each of the runs we report the maximum number of states and transitions explored (separated by the corresponding premise), the analysis time, and the generated assumption's number

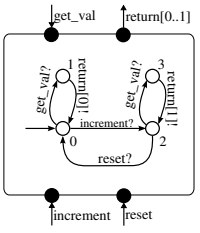


Fig. 7. The ARD counter

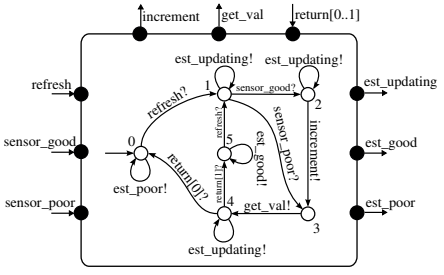


Fig. 8. The ARD position estimator

Table 1. Analysis results for a NASA ARD model

Check	Non-compositional			Compositional				
	States	Transitions	Time	Rule	States	Transitions	$\|A\|$	Time
Compatibility	2434K	16612K	37s	IA-COMPAT			35	445s
				Prem. 1	5K	34K		
				Prem. 2	**182K**	**864K**		
Property	2438K	16634K	36s	IA-PROP			74	483s
				Prem. 1	20K	113K		
				Prem. 2	**433K**	**3393K**		

of states. The experiments were run on a 64-bit Sun machine running Windows and a 1GB Java virtual machine.

During compositional verification, the largest state spaces explored were during the second premises for the assume-guarantee rules (highlighted in bold in Table 1). This is approximately one order of magnitude smaller than the state space explored when checking $M_1 \| M_2$ directly. On the other hand, assume-guarantee reasoning took 445s (483s) as compared to 37s (36s) for checking compatibility (resp., safety property) directly. This time penalty is due to the iterative nature of the learning algorithm and to the relatively large number of states in the generated assumption. Previous studies [8, 2] on learning for compositional reasoning in other formalisms showed that the approach does not incur such time penalty when smaller assumptions are obtained (i.e., less than 10 states).

The results reported in Table 1 for checking the safety property use the optimized algorithm presented in Section 6. Application of our algorithm from [8] (that views the two components as LTSs and does not take advantage of compatibility) resulted in an assumption of 77 states. The savings in terms of the assumption size (3 states) are not significant in this case. The reason is that the components in our study exhibit behavior where inputs and outputs strictly

alternate. More pronounced savings may be obtained in systems where occurrences of chains of output and internal actions may lead to the error state. Finally, we have also experimented with an algorithm that combines Rules IA-Compat and IA-Prop: the hybrid algorithm computes an assumption that guarantees both compatibility and property satisfaction. With this algorithm, we obtained an assumption of 94 states (in 3930 seconds). This indicates that checking the two properties separately results in smaller assumptions.

We remark that the largest assumptions built by our frameworks are still much smaller than the components that they represent in the compositional rules (M_1 has over 5K states and M_2 has over 143K states). Therefore, the cost of re-verification, for example, using assume-guarantee reasoning will be much smaller than non-compositional verification.

8 Related Work

Several frameworks have been proposed to support assume-guarantee reasoning [14, 21, 6, 12], but their practical impact has been limited by their need for human input in defining appropriate assumptions. Frameworks using L* to learn assumptions or component interfaces have been developed, for example, in the context of assume-guarantee reasoning of LTSs [11,8], synthesizing interfaces for Java classes [1], and symbolic model checking using NuSMV [2]. Unlike these approaches, we distinguish input actions from output actions to allow stricter property checking (i.e., ensuring compatibility).

Several optimizations to learning for assume-guarantee reasoning have also been studied. Alternative system decompositions [7, 20] and discovering small interface alphabets [10, 4] may positively affect the performance of learning. Our methods are orthogonal to these, as we consider an extended transition system.

Another approach [13] uses predicate abstraction and refinement to synthesize interfaces for software libraries. This work does not use learning, nor does it reuse the resulting interfaces in assume-guarantee verification. Several approaches have been defined to automatically abstract a component's environment to obtain interfaces [15, 5], however these techniques are not incremental and do not differentiate between inputs and outputs.

9 Conclusion

In this work we have developed a theoretical framework for the automated compositional verification of systems modeled with interface automata. We provide sound and complete assume-guarantee rules, and learning-based algorithms targeting the weakest-possible assumption for each rule. An evaluation of our algorithms on an application of a NASA case study is also presented, based on the implementation of the algorithms in a practical model checking tool.

References

1. Alur, R., Cerny, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: Proc. 32nd POPL (2005)
2. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Proc. 17th CAV (2005)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2) (1987)
4. Chaki, S., Strichman, O.: Optimized L* for assume-guarantee reasoning. In: Proc. 13th TACAS (2007)
5. Cheung, S.C., Kramer, J.: Checking safety properties using compositional reachability analysis. TOSEM 8(1) (1999)
6. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: Proc. 4th LICS (1989)
7. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: An investigation of decomposition for assume-guarantee reasoning. In: ISSTA (2006)
8. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Proc. 9th TACAS (2003)
9. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. 8th ESEC/FSE (2001)
10. Gheorghiu, M., Giannakopoulou, D., Păsăreanu, C.S.: Refining interface alphabets for compositional verification. In: Proc. 13th TACAS (2007)
11. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Assumption generation for software component verification. In: Proc. 17th ASE (2002)
12. Grumberg, O., Long, D.E.: Model checking and modular verification. In: Proc. 2nd CONCUR (1991)
13. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: Proc. 10th ESEC/FSE (2005)
14. Jones, C.B.: Specification and design of (parallel) programs. In: Proc. 9th IFIP Congress (1983)
15. Krimm, J.-P., Mounier, L.: Compositional state space generation from Lotos programs. In: Proc. 3rd TACAS (1997)
16. Letier, E., Kramer, J., Magee, J., Uchitel, S.: Monitoring and control in scenario-based requirements analysis. In: Proc. 27th ICSE (2005)
17. Lynch, N., Tuttle, M.: An introduction to input/output automata. Centrum voor Wiskunde en Informatica 2(3) (1989)
18. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs. John Wiley & Sons, Chichester (1999)
19. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour analysis of software architectures. In: Proc. 1st WICSA (1999)
20. Nam, W., Alur, R.: Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In: Proc. 4th ATVA (2006)
21. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logic and Models of Concurrent Systems (1984)
22. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. In: Proc. 21st STOC (1989)
23. Veanes, M., Campbell, C., Schulte, W., Tillmann, N.: Online testing with model programs. In: Proc. 10th ESEC/FSE (2005)