Interface Input/Output Automata

Kim G. Larsen[1], Ulrik Nyman[1], and Andrzej Wąsowski[2⋆]

[1] Department of Computer Science, Aalborg University
{kgl,ulrik}@cs.aau.dk
[2] Computational Logic and Algorithms Group, IT University of Copenhagen
wasowski@itu.dk

Abstract. Building on the theory of interface automata by de Alfaro and Henzinger we design an interface language for Lynch's I/O automata, a popular formalism used in the development of distributed asynchronous systems, not addressed by previous interface research. We introduce an explicit separation of assumptions from guarantees not yet seen in other behavioral interface theories. Moreover we derive the composition operator systematically and formally, guaranteeing that the resulting compositions are always the weakest in the sense of assumptions, and the strongest in the sense of guarantees. We also present a method for solving systems of relativized behavioral inequalities as used in our setup and draw a formal correspondence between our work and interface automata.

1 Introduction

A suitably expressive interface language lies at the very center of any component-oriented development framework. Interfaces are abstractions of components, carrying all essential information necessary to establish cross-component compatibility. Instead of reasoning about components directly, one typically examines compatibility of their interfaces, while the adherence of a particular implementation to its interface is tested separately. This, not only allows for independent development of components, but also by introducing compositionality helps to combat the state space explosion problem in various automatic analyses.

Type annotations, type checking, and type inference have traditionally been used to decide compatibility of components soundly with respect to memory safety. However, static type correctness in this traditional sense fails to guarantee more elaborate properties, like correctness of communication, or deadlock freeness. This observation has inspired a long line of research on behavioral type systems and behavioral interface languages suitable for specification of highly trusted computer systems (see [1–4] and references therein for examples).

We follow de Alfaro and Henzinger [5,6] in studying an automata based interface language, or *interface automata*. Unlike them however, we explicitly separate, in the interface description, the assumptions that a component may make about its use from the guarantees that it needs to commit to. Assumptions describe the possible behaviors of the component's external environment, while guarantees describe the possible behaviors of the component itself.

⋆ Partly supported by Center for Embedded Software Systems (CISS) in Aalborg.

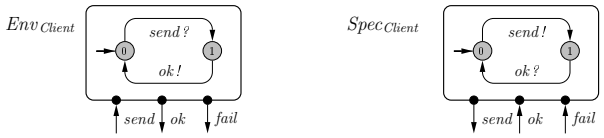

Fig. 1. $Client = (Env_{Client}, Spec_{Client})$

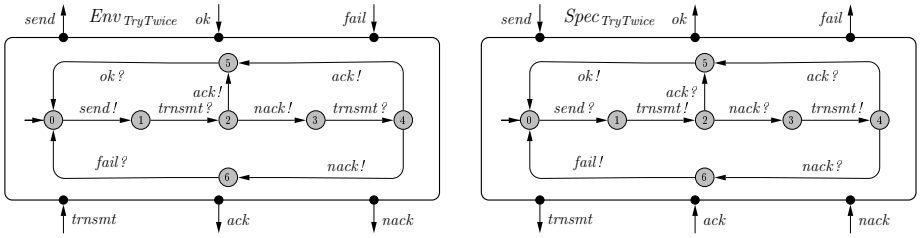

Fig. 2. $TryTwice = (Env_{TryTwice}, Spec_{TryTwice})$

Each interface in our theory consists of two I/O automata. The first, called the *environment*, represents assumptions. The second, called the *specification*, describes guarantees. Figure 1 shows an interface for a *Client* component consisting of the automata Env_{Client} and $Spec_{Client}$. The arrows incoming to or outgoing from the box surrounding each of the automata visualize their static types, or *signatures*. The environment Env_{Client} specifies that even though the static type does allow a *fail* action, the emission of this action is disallowed for all compliant execution environments. The only legal input is *send*. One can still use the *Client* component in a context that syntactically permits *fail*, but the behavior of the *Client* is only guaranteed in environments that do not fail.

Alfaro and Henzinger model assumptions about the use of a component by the interface's inabilities to receive inputs. The output transitions of the very same interface automaton describe its guarantees. Since we separate the two, we alleviate the need for blocking. Our automata are *input enabled*—accepting any input from their signature in every state. In order to avoid clutter we usually do not draw loop transitions, which correspond to ignoring an input. There is one such implicit transition $1 \xrightarrow{send?} 1$ in Env_{Client} and three in $Spec_{Client}$.

Two interfaces can be combined into a composite interface, describing a new set of assumptions and guarantees. Interface *TryTwice*, presented in Fig. 2 can be composed with *Client*. The two components do not form a closed system, but are intended for use together with a further unspecified *LinkLayer* component.

Composition of interfaces is a central construction in any interface theory. One of our contributions is that the composition is derived systematically: we formally state requirements for it in the form of a system of inequalities, and derive a result of the composition as a maximal solution of this system. Consequently properties of the composition hold by construction.

Figure 3 shows the interface resulting from composing *Client* and *TryTwice*. Later we shall explain how it has been computed. Now observe that any component legally interacting with this new interface may not send a *nack* twice

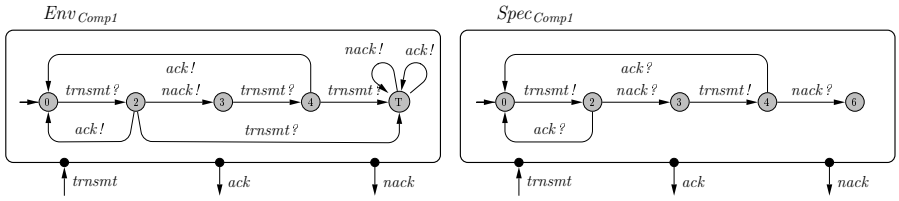

Fig. 3. $(Env_{TryTwice}, Spec_{TryTwice})|(Env_{Client}, Spec_{Client}) = Comp1$

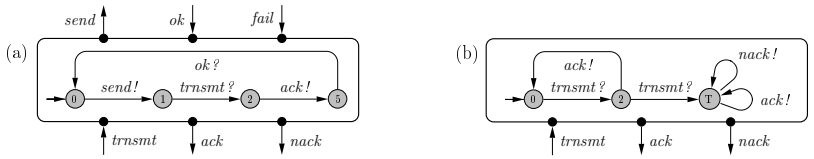

Fig. 4. (a) The environment Env_{NoNack} and (b) the environment Env_{Comp2}.

in response to the *transmt* request—a simple consequence of the fact that this would make *TryTwice* respond with a *fail* to *Client*, violating the assumptions of the latter. The additional state T manifests the fact that the computed environment expresses the weakest assumptions. It allows receiving arbitrary behavior after a second *transmt* in a row, because any compliant implementation would never send it, and thus would never be affected by the subsequent behaviour.

An advantage of separating assumptions from guarantees is that one of the automata can be changed without affecting the other. Thus the same guarantees can be used for multiple interfaces. In [7] we have argued that this is useful for modeling software product lines: a family of component variants may be specified using a single specification (guarantee) and multiple environmental restrictions (assumptions). An advanced compiler may use the assumptions to derive specialized versions of the component from the same source code. Let us illustrate this with an example. Figure 4a gives an alternative environment Env_{NoNack} for the $Spec_{TryTwice}$ specification. This environment disallows the sending of a *nack* as a response to a *trnsmt* request. Any implementation of *TryTwice* is also an implementation of $(Env_{NoNack}, Spec_{TryTwice})$. If it is only used in Env_{NoNack}, then it could be automatically specialized to these specific circumstances. The error handling code could be removed as it is not needed in such a context. The composition $Comp2 = (Env_{NoNack}, Spec_{TryTwice})|(Env_{Client}, Spec_{Client})$ has exactly the same specification part as the *Comp1* composition. The resulting environment Env_{Comp2} (Fig. 4b) disallows the generation of the *nack* input even though the static type permits this.

As we have also argued in [7] the separation supports a simple declarative style of modeling assumptions: simple properties can be modeled as standalone automata and combined using the process algebraic operators of sum and product, corresponding to disjunction and conjunction of properties respectively.

An interesting theoretical side effect of our exposition, is an informal correspondence drawn between blocking and non-blocking interface theories. A single

blocking interface automaton of [5] expresses both the assumptions of a component and its commitments. When a blocking interface automaton is unable to accept an input, it effectively assumes that any compatible environment will never provide it. In the theory for non-blocking systems the interfaces are composed of two non-blocking automata, and the same effect is achieved by explicitly using one of the automata for describing the permissible behavior of the surroundings.

The paper develops as follows. Section 2 defines I/O automata and interfaces. Section 3 discusses refinement of interfaces. The most central section, Section 4, is devoted to composition, while a more technical section, Section 5, is devoted to systems of inequalities used in section 4 and is a contribution in itself. But reading it is not essential for appreciating our interface theory. Section 6 draws a correspondence between interface automata and our interfaces, while section 7 discusses other related work. We conclude in section 8. A particularly interested reader can find the proofs of all our claims in an upcoming BRICS report.

2 I/O Automata and Their Interfaces

Definition 1. *An I/O automaton* $S{=}(states_S, start_S, in_S, out_S, int_S, steps_S)$ *is a 6-tuple, where* $states_S$ *is a set of states,* $start_S \in states_S$ *is an initial state,* in_S *is a set of input actions,* out_S *a set of output actions, and* int_S *is a set of internal actions. All of the action sets are mutually disjoint. We abbreviate* $ext_S = in_S \cup out_S$ *and* $act_S = ext_S \cup int_S$*. Then* $steps_S \subseteq states_S \times act_S \times states_S$ *is the set of transitions. I/O automata are input enabled: for every state* s *and any action* $i \in in_S$ *there exists a state* s' *and a transition* $(s, i, s') \in steps_S$*.*

We write $q \xrightarrow{a}_S q'$ if $(q, a, q') \in steps_S$. We often explicitly suffix external actions with direction of communication writing $q \xrightarrow{a!}_S q'$ if $a \in out_S$, and $q \xrightarrow{a?}_S q'$ if $a \in in_S$. Notice that the labels $a!$ and $a?$ still denote exactly the same action, and we can drop the suffixes whenever the direction of communication is irrelevant. We write $q \xrightarrow{a}\!\!\!\!\!\!/\;\;$, meaning that there is no q' such that $q \xrightarrow{a} q'$.

Definition 2. *An execution of an I/O-automaton* S *starting in a state* q^0 *is a finite sequence of labels* $q^0, a_0, q^1, a_1, q^2, a_2, \ldots, q^{n-1}, a_{n-1}, q^n$ *such that all* q^i*'s are members of* $states_S$*, all* a_i*'s are members of* act_S *and for every* $k = 0 \ldots n-1$ *it is the case that* $q^k \xrightarrow{a_k}_S q^{k+1}$*. A trace* σ *of* S *is an execution* ψ *of* S *starting in the initial state, with all the states and internal actions deleted:* $\sigma = \psi \restriction ext_S$*, where* $\psi \restriction X$ *denotes a sequence created from* ψ *by removing symbols that are not in set* X*. The set of all traces of automaton* S *is denoted* Tr_S*.*

Two I/O-automata S_1 and S_2 are *syntactically composable* if their input and output sets do not overlap and their internal actions are not shared: $in_{S_1} \cap in_{S_2} = out_{S_1} \cap out_{S_2} = int_{S_1} \cap act_{S_2} = act_{S_1} \cap int_{S_2} = \emptyset$. Two syntactically composable automata $S_1 = (states_{S_1}, start_{S_1}, in_{S_1}, out_{S_1}, int_{S_1}, steps_{S_1})$ and $S_2 = (states_{S_2}, start_{S_2}, in_{S_2}, out_{S_2}, int_{S_2}, steps_{S_2})$ can be composed into a single product automaton $S = S_1|S_2$, where $S{=}(states_S, start_S, in_S, out_S, int_S, steps_S)$ and $states_S = states_{S_1} \times states_{S_2}$, $start_S = (start_{S_1}, start_{S_2})$, $in_S = in_{S_1} \cup in_{S_2} \setminus out_{S_1} \setminus out_{S_2}$,

$out_S = out_{S_1} \cup out_{S_2} \setminus in_{S_1} \setminus in_{S_2}$, $int_S = int_{S_1} \cup int_{S_2} \cup (ext_{S_1} \cap ext_{S_2})$, and $steps_S$ are defined by the following rules:

$$\text{if } q_1 \xrightarrow{a}_{S_1} q_1' \text{ and } a \in act_{S_1} \setminus act_{S_2} \text{ then } (q_1, q_2) \xrightarrow{a}_{S_1|S_2} (q_1', q_2)$$
$$\text{if } q_2 \xrightarrow{a}_{S_2} q_2' \text{ and } a \in act_{S_2} \setminus act_{S_1} \text{ then } (q_1, q_2) \xrightarrow{a}_{S_1|S_2} (q_1, q_2')$$
$$\text{if } q_1 \xrightarrow{a}_{S_1} q_1' \text{ and } q_2 \xrightarrow{a}_{S_2} q_2' \text{ then } (q_1, q_2) \xrightarrow{a}_{S_1|S_2} (q_1', q_2')$$

In practice unreachable states may be removed from the product, without affecting the results presented below.

Our composition (same as in [6]) differs from the standard I/O automata composition in that it applies hiding immediately. It is equivalent with the standard composition as long as each action is only shared by at most two components.

We define an interface model to be a pair (E, S) of I/O automata:

Definition 3. *A pair of I/O automata (E, S) is an interface if $E|S$ is a closed system, i.e. $in_E = out_S$ and $out_E = in_S$.*

The environment automaton E drives the specification automaton S. Any implementation I of S must conform to S as long as it is receiving input that conforms to E. The behavior of I on sequences of inputs that cannot be provided by E is not constrained. We formalize this using relativized refinement:

Definition 4. *An I/O automaton I implements an interface (E, S), written $E \models I \leqslant S$, iff $out_I = out_S$ and $in_I = in_S$ and $Tr_E \cap Tr_I \subseteq Tr_S$.*

3 Refinement of Interfaces

We establish a hierarchy on interfaces in order to quantify their generality.

Definition 5. *Let (E_1, S_1) and (E_2, S_2) be two interfaces with the same signatures. We will say that (E_1, S_1) is a stronger interface than (E_2, S_2), written $(E_1, S_1) \preceq (E_2, S_2)$, if (E_1, S_1) has less implementations than (E_2, S_2), so for any I/O automaton I: $E_1 \models I \leqslant S_1$ implies $E_2 \models I \leqslant S_2$.*

The refinement of interfaces can be seen as a subtyping relation in a behavioral type system for components. In such an interpretation we would say that (E_1, S_1) is a subtype of (E_2, S_2). We propose several simple sound characterizations of the above refinement that are useful in making proofs:

Theorem 6. *Let (E_1, S_1), (E_2, S_2) be interfaces with identical signatures. Then*

1. *$Tr_{E_1} \cap Tr_{S_1} = Tr_{E_2} \cap Tr_{S_2}$ implies $(E_1, S_1) \preceq (E_2, S_2)$ and $(E_2, S_2) \preceq (E_1, S_1)$*
2. *$Tr_{E_2} \subseteq Tr_{E_1} \wedge Tr_{S_1} \subseteq Tr_{S_2}$ implies $(E_1, S_1) \preceq (E_2, S_2)$*
3. *$Tr_{E_1} \setminus Tr_{S_1} \supseteq Tr_{E_2} \setminus Tr_{S_2}$ implies $(E_1, S_1) \preceq (E_2, S_2)$*

The above characterizations are convenient in establishing subtyping relations among interfaces in many concrete cases. However none of them are complete. The refinement of interfaces can be characterized in a sound and complete manner using a notion of tests that resembles failure traces of Hoare [8], but determinized, relativized with respect to the environment, and suffix closed.

Definition 7. *The set of conformance tests of interface* (E, S) *is defined as:*

$$test_{(E,S)} = \{\sigma \cdot a \,|\, \sigma \in Tr_E \cap Tr_S,\ \sigma \cdot a \in Tr_E \setminus Tr_S\} \cdot ext_E^* \ ,$$

where X^* *denotes the set of all finite sequences over alphabet* X*.*

Theorem 8. *Let* (E_1, S_1) *and* (E_2, S_2) *be two interfaces with identical signatures. Then* $test_{(E_1,S_1)} \supseteq test_{(E_2,S_2)}$ *iff* $(E_1, S_1) \preceq (E_2, S_2)$*.*

Without spelling out the details, we remark that a finite automaton, such that $test_{(E,S)}$ is its accepted language, can be computed in quadratic time, and can be used for testing containment in applications of the above theorem.

4 Interface Compositions

We would like to abstract compositions of components by compositions of their interfaces. For any two compatible interfaces (E_1, S_1) and (E_2, S_2) we should be able to derive an interface of their composition (E, S), the one that is implemented flawlessly by any two implementations of (E_1, S_1) and (E_2, S_2).

Two interfaces are *syntactically composable* if the I/O automata comprising them are pointwise syntactically composable. This guarantees that any components I_1 and I_2 implementing syntactically composable interfaces (E_1, S_1) and (E_2, S_2), are also syntactically composable. The question that we want to address is the *dynamic compatibility* of I_1 and I_2: can I_1 violate the environmental assumptions expressed in E_2? Can I_2 violate the assumptions in E_1?

We may be tempted to say that the composite interface is the composition of the interface parts: $(E, S) = (E_1|E_2, S_1|S_2)$. This construction, however, is unsound. It is possible to find two compliant implementations that, when composed together, violate (E, S). In order to arrive at a sound and complete notion of composition, we will state the requirements for the composite interface, and then derive the construction from them. The three requirements are: *independent implementability* [6], *mutal deadlock freeness*, and *associativity*.

Independent implementability means that (E, S) is such, that the implementations of (E_1, S_1) and (E_2, S_2) can be developed independently of each other, and their composition will implement the composition of their interfaces:

$$\text{For all } I_1, I_2.\ E_1 \models I_1 \leqslant S_1 \text{ and } E_2 \models I_2 \leqslant S_2 \text{ implies } E \models I_1|I_2 \leqslant S \ . \quad (1)$$

Mutual deadlock freeness means that any two correct implementations, when composed and embedded in an environment that obeys the assumptions of E, will not violate each other's assumptions:

$$\text{For all } I_1, I_2. E_1 \models I_1 \leqslant S_1 \text{ and } E_2 \models I_2 \leqslant S_2 \text{ implies } I_1 \models E|I_2 \leqslant E_1 \text{ and } I_2 \models E|I_1 \leqslant E_2 \ . \quad (2)$$

You may find it useful to refer to the flowgraph on Fig. 5a, while studying the above rule. Observe that in the composed system I_1 is indeed the environment

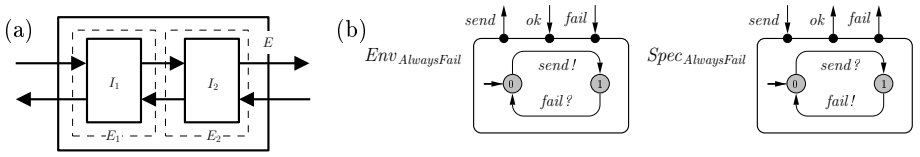

Fig. 5. (a) Flowgraph for a composition of (E_1, S_1) and (E_2, S_2). (b) *AlwaysFail*

in which $E|I_2$ operates. The composition $E|I_2$ is also the environment for I_1 and it is supposed not to violate any of the assumptions expressed in E_1.

Finally, associativity means that in whatever order compositions are applied, they give rise to equivalent interfaces:

$$\begin{aligned}&((E_1, S_1) \,|\, (E_2, S_2)) \,|\, (E_3, S_3) \preceq (E_1, S_1) \,|\, ((E_2, S_2) \,|\, (E_3, S_3))\\ &(E_1, S_1) \,|\, ((E_2, S_2) \,|\, (E_3, S_3)) \preceq ((E_1, S_1) \,|\, (E_2, S_2)) \,|\, (E_3, S_3)\ .\end{aligned} \tag{3}$$

A disadvantage of the above requirements is that they are not constructive. They rely on quantification over all implementations, which makes them useless for computing the composition. Fortunately the quantification can be eliminated. The following theorem reduces the property of mutual deadlock freeness of all implementations to mutual deadlock freeness of the interfaces being composed:

Theorem 9. *Any environment E fulfills the requirement (2) iff it fulfills the following condition:*

$$S_1 \models E|S_2 \leqslant E_1 \text{ and } S_2 \models E|S_1 \leqslant E_2\ . \tag{4}$$

The above reduction is very fortunate, as (4) also implies independent implementability with the choice of the guarantees component to be $S_1|S_2$:

Theorem 10. *Let (E_1, S_1) and (E_2, S_2) be syntactically composable interfaces, and E be an environment I/O automaton satisfying property (4). Then for all I_1 and I_2 such that $E_1 \models I_1 \leqslant S_1$ and $E_2 \models I_2 \leqslant S_2$ we have $E \models I_1|I_2 \leqslant S_1|S_2$.*

Consequently if we were able to find an environment E satisfying (4), then the interface $(E, S_1|S_2)$ would satisfy mutual deadlock freeness and independent implementability—a good candidate for the composition of environments. However, the environment satisfying (4) may not always exist. This is the case, if S_1 unconditionally, independently of E's behavior, violates the assumptions of S_2 expressed in E_2. In this case (E_1, S_1) and (E_2, S_2) are said to be *incompatible*.

Definition 11. *Interfaces (E_1, S_1), (E_2, S_2) are incompatible if there exists no I/O automaton E such that: $S_1 \models E|S_2 \leqslant E_1$ and $S_2 \models E|S_1 \leqslant E_2$.*

Figure 5b shows an interface *AlwaysFail*, which has a signature compatible with the signature of *Client*. Nevertheless the dynamic types of *Client* and *AlwaysFail* are incompatible in that they share only one nonempty trace, consisting of one step, and this trace ends in a deadlock.

In fact there typically exist many pairs (E, S) that satisfy all our requirements. For example an interface (M, U), consisting of a mute environment M never producing any outputs and a universal system specification U generating all possible traces, would satisfy the composition requirements of any two compatible interfaces. The interface (M, U) allows any implementation—it says that its implementations will behave in an arbitrary fashion (U), not allowing any external stimulation (M). Clearly, as a component interface, (M, U) is useless.

We should ensure that our composition operator produces the interface that carries over all the information available from its components. It must have the smallest possible set of implementations, while still satisfying all our requirements. Similarly, it must maximize the set of components compatible with it (as opposed to the set of components implementing it). We shall call this optimal interface *the most general*. Intuitively to achieve this optimality we need an environment E satisfying the requirements such that it is maximal with respect to trace inclusion. By increasing the set Tr_E we make it easier for components to be compatible with our interface. Similarly we make it harder to implement the composite interface, as increasing the set of traces of E decreases the assumptions that an implementation can make. The following theorem says that such a maximal E always exists for compatible interfaces:

Theorem 12. *Let (E_1, S_1) and (E_2, S_2) be two syntactically composable interfaces. If there exists an I/O automaton E enjoying property (4) then there also exists a maximal such environment with respect to trace inclusion.*

Theorem 13. *The composition operator mapping interfaces (E_1, S_1) and (E_2, S_2) to $(E, S_1|S_2)$, where E is the maximal solution of (4), is associative.*

Theorems 12–13 together with our earlier observations suggest that the interface $(E, S_1|S_2)$, where E is this maximal solution of equations (4), is even more likely to be the most general interface that we are searching for. A maximal solution of (4) can be found algorithmically for finite state interfaces. Section 5 describes a method that can be used for this purpose.

As increasing the environment E makes the interfaces more general, so does decreasing the specification S (within the limits set by the requirements). For any particular selection of E satisfying (1), no S can be smaller (relative to E) than $S_1|S_2$, because S_1 and S_2 themselves are valid implementations. So $S_1|S_2$ is the smallest possible specification of the composite interface with respect to any particular choice of E. This observation can be generalized to a claim that $(E, S_1|S_2)$ is the most general interface possible:

Theorem 14. *Let (E_1, S_1), (E_2, S_2) be interfaces. Let E be the maximal solution to (4) and let (E', S') satisfy independent implementability and mutual deadlock freeness. If (E', S') is compatible with (E'', S'') then also $(E, S_1|S_2)$ is compatible with (E'', S'').*

Having concluded that $(E, S_1|S_2)$, where E is a maximal solution of (4), is well defined and the most general, we can use it as a definition of the composition operator. We will denote this composite interface by $(E_1, S_1)|(E_2, S_2)$.

Furthermore our composition of interfaces is complete in the following sense

Theorem 15. *For compatible interfaces (E_1, S_1), (E_2, S_2) and any (E', S') satisfying independent implementability and mutual deadlock freeness:*

$$(E_1, S_1)|(E_2, S_2) \preceq (E', S') \ .$$

We remark that our composition would not be complete if we only required independent implementability. It seems likely from the work presented in [9] that it is indeed impossible, for our setting, to be complete in the above sense using only independent implementability. Similarly we would not be complete if we only required mutual deadlock freeness, simply because it does not restrict the S component, which can then be taken to be mute, likely yielding a smaller interface than ours. Still our composition is sound and complete with respect to both requirements combined. Requirements (2) and (3) have been introduced solely for their inherent usefulness. Their interplay guaranteeing soundness and completeness is a pleasant side effect.

Definition 16. *Let (E_1, S_1), (E_2, S_2) be syntactically composable interfaces. Their composition, denoted $(E_1, S_1)|(E_2, S_2)$, is an interface $(E, S_1|S_2)$, where E has the same signature as $E_1|E_2$, and is a maximal solution of (4).*

The operator of Def. 16 is associative, supports independent implementability and mutual deadlock freeness, and produces the most general interfaces.

5 Solving Behavioral Inequalities

Computing compositions of interfaces requires a method for finding solutions of systems of relativized linear inequalities. In particular we are interested in systems of inequalities of the following form:

$$\mathcal{C}(E) \ : \ \begin{cases} P_1 \models E|S_1 \leqslant F_1 \\ \quad \vdots \\ P_m \models E|S_m \leqslant F_m \end{cases} \tag{5}$$

where $\{P_i\}_{i=1..m}$, $\{S_i\}_{i=1..m}$ and $\{F_i\}_{i=1..m}$ are states of the three I/O automata P, S and F and E is a single unknown automaton. We are interested in finding a greatest such E with respect to $\leqslant$, or in reporting incompatibility between components, if no solutions exist. Since in (4) various components of inequalities come from separate automata, in order to apply the method below we need to construct three automata P, S and F as the disjoint unions of the automata that appear in the given place of the constraints in (4). We introduce three convenient mapping functions *in*, *out* and *ext* which from a state of the two automata F and S return respectively the set of input, output or external actions of the automata that this state originates from in the disjoint union computation. We will use them in the algorithm below to recover some of the signature information lost by making the disjoint union.

For simplicity of exposition we shall also assume that all I/O automata involved in the systems are deterministic. Otherwise they can be determinized

without loss of information, as long as our refinement criterion is based on language inclusion. This assumption is not inherent to the method, though.

We should now state a property similar to Theorem 12, but formulated for systems of inequalities in general. We expand it to any number of constraints and do not require that all the I/O automata come from the same interfaces.

Theorem 17. *Let $\mathcal{C}(E)$ be a finite system of relativized inequalities:*

$$\mathcal{C}(E) \;:\; \begin{cases} P_1 \models E|S_1 \leqslant F_1 \\ \quad\vdots \\ P_m \models E|S_m \leqslant F_m \end{cases}$$

If $\mathcal{C}(E)$ has a solution (an I/O automaton satisfying all the constraints), then $\mathcal{C}(E)$ also has a greatest solution with respect to trace set inclusion.

We begin with constructing a *modal transition system* [10] corresponding to $\mathcal{C}(E)$, and then choose a maximal solution from its states and transitions. From our perspective modal transition systems are automata with two transition relations $\rightarrow_{may}$ and $\rightarrow_{must}$.

Definition 18. *A modal transition system is a quadruple $\mathcal{S} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$, where Q is a set of systems of constraints (states), A is a set of actions, $\rightarrow_{may} \subseteq Q \times A \times Q$ is the* may *transition relation, and $\rightarrow_{must} \subseteq Q \times A \times Q$ is the* must *transition relation, $\rightarrow_{must} \subseteq \rightarrow_{may}$.*

Systems of relativized inequalities can be seen as sets of constraint triples $\{(P_1, S_1, F_1), \ldots, (P_m, S_m, F_m)\}$ over the solution E. The constraints evolve when any of their components, including the unknown E, takes an action. This evolution comprises not only state changes of the I/O automata, but also removing and introducing constraints. Legal actions of the unknown component E in any of its states are dependent on the states of the constraints—on what all the P_i's, S_i's and all the F_i's can do. This is why we label states of our modal transition systems with systems of inequalities (sets of constraints). All the steps that are allowed by the constraints, but are not strictly required (like a possibility to produce an output) should give rise to *may* transitions in the modal transition system. While all the steps that are strictly required (like input actions enforced by input-enabledness) give rise to corresponding *must* transitions.

Formally three I/O automata P, S, F induce a modal transition system $\mathcal{E} = (Q, A_0, \rightarrow_{may}, \rightarrow_{must})$, where elements of Q are sets of constraints over states of P, S and F, enriched with a distinct primitive constraint FALSE denoting an empty set of solutions. The initial state A_0 is equal to the set $\{(P_1, S_1, F_1), \ldots, (P_m, S_m, F_m)\}$ of initial constraints, and the transition relations are defined according to the following rules:

$E \xrightarrow{a!}_{may} E'$ if and only if both of the following rules are satisfied:

For all $(P, S, F) \in E$ such that $a \in out_E \setminus in_S$
If $\exists F'.\, F \xrightarrow{a!} F'$ and $\exists P'.\, P \xrightarrow{a} P'$ then $(P', S, F') \in E'$
Else if $\exists P'.P \xrightarrow{a?} P'$ and $F \xrightarrow{a!}\!\!\!\!\!\!/\;\;$ then FALSE $\in E'$

For all $(P, S, F) \in E$ and all S' such that $a \in out_E \cap in_S$
If $S \xrightarrow{a?} S'$ also $(P, S', F) \in E'$

$E \xrightarrow{a?}_{must} E'$ and $E \xrightarrow{a?}_{may} E'$ iff both of the following rules are satisfied:

For all $(P, S, F) \in E$ and all F' such that $a \in in_E \setminus out_S$
If $F \xrightarrow{a?} F'$ and $P \xrightarrow{a!} P'$ then $(P', S, F') \in E'$
For all $(P, S, F) \in E$ such that $a \in in_E \cap out_S$
If $S \xrightarrow{a!} S'$ then $(P, S', F) \in E'$

Each state $E \in Q$ of $\mathcal{E}$ is minimal such that it satisfies the above transition rules *and* the following *closure rules*:

For all $(P, S, F) \in E$ and $a \in ext_S \cap ext_F$
If $\exists S'.\, S \xrightarrow{a} S'$ and $\exists F'.\, F \xrightarrow{a} F'$ and $\exists P'.\, P \xrightarrow{a} P'$
then also $(P', S', F') \in E$.
For all $(P, S, F) \in E$ and $a \in ext_S \cap ext_F$
If $S \xrightarrow{a!} S'$ and $F \xrightarrow{a!} \not\rightarrow$ and $\exists P'.\, P \xrightarrow{a?} P'$ then $\text{FALSE} \in E$.

The two *may* rules discuss E making an output transition concerning an external output, or an internal communication with S respectively. The *must* rules state that E needs to accept all the inputs from the outside and from S respectively. Finally the closure rules allow S to advance without any interference with E on its own external actions. Whenever there is a possibility of violation of the relativized trace inclusion, we add false to the target state of E, hinting that E should not be allowed to make that step.

Definition 19. *The state consistency relation $\mathcal{S}$ over a modal transition system $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ is the maximal subset of Q such that if $E \in \mathcal{S}$ then $\text{FALSE} \notin E$ and whenever $E \xrightarrow{a}_{must} E'$ then $E' \in \mathcal{S}$.*

Definition 20. *A consistent set of transitions $\mathcal{T}$ of a modal transition system $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ with respect to consistency relation $\mathcal{S}$ is a maximal subset of $\rightarrow_{may}$, where whenever $(s, a, s') \in \mathcal{T}$ then $s \in \mathcal{S}$ and $s' \in \mathcal{S}$.*

Theorem 21. *Let $\mathcal{C}(E)$ be a system of inequalities as required above, and $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ be the modal transition system induced by $\mathcal{C}$. Then the maximal solution of $\mathcal{C}(E)$ is an I/O automaton E such that its set of states $states_E$ is a maximal consistency relation over $\mathcal{E}$,*

$$\begin{aligned} start_E &= \{(F_1, S_1), ..., (F_m, S_m)\}, \\ in_E &= \bigcup_{i=1}^{m} (in_{F_i} \setminus in_{S_i}) \cup \bigcup_{i=1}^{m} (out_{S_i} \setminus out_{F_i}) \\ out_E &= \bigcup_{i=1}^{m} (out_{F_i} \setminus out_{S_i}) \cup \bigcup_{i=1}^{m} (in_{S_i} \setminus in_{F_i}), \end{aligned}$$

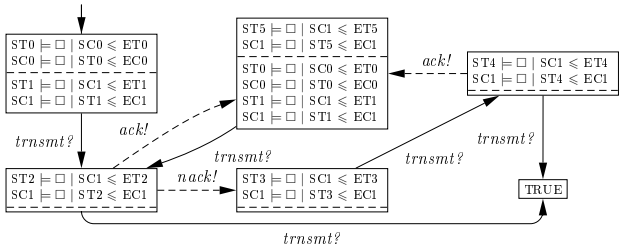

Fig. 6. The resulting modal transition system for the computation of Env_{Comp1}.

and its set of transitions $step_E$ *is a maximal consistent set of transitions of* $\mathcal{E}$ *with respect to* $states_E$*. If the maximal state consistency relation of* $\mathcal{E}$ *is empty then* $\mathcal{C}$ *has no solutions.*

The set $\mathcal{S}$ can be found by a simple maximal fixpoint computation. In practice the consistency of the initial state may be decided in a local fashion without constructing the entire modal transition system.

Figure 6 shows the consistent part of the modal transition system induced by $(Env_{TryTwice}, Spec_{TryTwice})|(Env_{Client}, Spec_{Client})$. It can then be minimized in order to obtain Env_{Comp1}, shown in Fig. 3. Similarly $Spec_{Comp1}$ from Fig. 3 has been obtained by minimizing $Spec_{TryTwice} \mid Spec_{Client}$.

6 Interface Automata

The relation of our theory to interface automata [5, 6] requires special attention, as we address several issues of that work; most importantly the representation of assumptions and guarantees within a single automaton. We clearly separate assumptions from guarantees, and the pairs of assumptions and guarantees can be constructed independently. In [6] Alfaro and Henzinger discuss static Assume/Guarantee interfaces featuring a similar split, however they do not persue the idea to the dynamic case.

In a larger perspective our work can be seen as a study of building interface theories as such: starting with a selection of the building blocks, going through requirements analysis, deriving the composition operator, and studying its generality. Let us review this process briefly. We begin with selecting important ingredients such as a component model, an interface model, an implementation relation and a refinement relation. The particular choice of input-enabled systems and (relativized) trace inclusion is not crucial for our developments. In fact we believe that a similar theory can be built using (relativized) simulation, or for timed automata. We choose I/O automata and trace inclusion because they are very different from Alfaro and Henzinger's interface automata, so we incidentally provide a component theory for a different community—the I/O automata community. At the same time our choice challenges some opinions expressed in [5, 6] that building such a theory, especially supporting contravariant refinement, is impossible using language inclusion criteria or in a non-blocking setting.

Furthermore we show how the composition operator can be derived from requirements (by analysis, reduction and automated solving), while Alfaro and Henzinger introduce this operator in a rather ad hoc manner. After having derived our operator we discuss its generality, and conclude that it is indeed the most general operator possible, meeting our requirements with respect to trace inclusion, with respect to the $\preceq$ refinement, and with respect to compatibility with other components. We conjecture that the operator of our predecessors is also the most general in their setting, however they never make that claim.

Let us now draw a formal correspondance between the two interface theories.

Definition 22 (after [6]). *An interface automaton is a six-tuple* $S = (states_S, start_S, in_S, out_S, int_S, steps_S)$*, where* $states_S$ *is a finite set of states,* $start_S \in states_S$ *is an initial state,* in_S*,* out_S*, and* int_S *are three pairwise disjoint sets of input, output, and internal actions respectively, and* $steps_S \subseteq states_S \times acts_S \times states_S$ *is an input-deterministic transition relation, with* $acts_S = in_S \cup out_S \cup int_S$

Notice that the transition relation of interface automata may be non input-enabled. Syntactic composability of interface automata is governed by the same rule as the composability of I/O automata, defined on p. 4. The composed interface is computed by taking a product of the two automata, and removing from it all *incompatible states*. A state of the product is an *error state* if one of its components can produce a shared output, that the other is unable to receive. A state of the product is *incompatible* if it can reach an error state by an execution over internally controllable transitions (transitions labeled with actions from: $int_{S_1|S_2} \cup out_{S_1|S_2}$).

Definition 23. *Two syntactically composable interface automata* S_1 *and* S_2 *are compatible iff removing all incompatible states from their product leaves an interface automaton with a non-empty set of reachable states.*

The function *unzip* defined below translates an interface automaton to an I/O automaton interface. If A is an interface automaton then $unzip_A := (E, S)$, where $states_S = states_E = states_A \cup \{T\}$, $start_S = start_E = start_A$, $in_S = out_E = in_A$, $out_S = in_E = out_A$, $int_S = int_E = int_A$. The transition relations of E and S are created from the transition relation of A by making it input-enabled on the respective input sets:

$$steps_E = steps_A \cup \{(s, a, T) | s \in states_A, a \in in_E, s \overset{a}{\not\longrightarrow}_A\}$$

$$steps_S = steps_A \cup \{(s, a, T) | s \in states_A, a \in in_S, s \overset{a}{\not\longrightarrow}_A\}$$

Theorem 24. *If* A_1 *and* A_2 *are two compatible interface automata, then* $unzip_{A_1}$ *and* $unzip_{A_2}$ *are compatible I/O automata interfaces.*

The *zip* function is a reverse of *unzip*: it translates an I/O automata interface into a single interface automaton, by computing the product of the two parts using the classic algorithm [11, chpt. 4.2] from automata theory: $zip_{(E,S)} := A$, where $states_A = states_E \times states_S$, $start_A = (start_E, start_S)$, $in_A = in_S$, $out_A = out_S$, $int_A = int_S \cup int_E$, and $steps_A = \{((s, e), a, (s', e')) | s \overset{a}{\longrightarrow} s' \text{ and } e \overset{a}{\longrightarrow} e'\}$.

Theorem 25. *If (E_1, S_1), (E_2, S_2) are compatible deterministic I/O automata interfaces, then $zip_{(E_1,S_1)}$, $zip_{(E_2,S_2)}$ are compatible interface automata.*

The fact that our compatibility only implies compatibility in the interface automata sense for unzippings of deterministic interfaces is not surprising. It is actually expected, due to the very different nature of the refinement relations used in the two theories: trace inclusion and alternating simulation [12].

Alfaro and Henzinger choose alternating simulation to support contravariant treatment of inputs and outputs. We stress that input-enabledness and relativized trace inclusion already guarantee contravariant treatment of behaviors in a very similar spirit. Still our theory somewhat strictly requires that implementations of an interface have precisely the same sort as their interfaces, so it is technically not possible to substitute a richer component in place of a simpler one, if they are the same on shared functionality. We stress that this deficiency is not inherent, while it simplifies the presentation. Contravariant signature extensions can be easily realized with relativized trace inclusion in the input-enabled setting. Instead of requiring $in_I = in_S$ and $out_I = out_S$ in Def. 3, insist on $in_S \subseteq in_I$ and $out_I \subseteq out_S$. In fact the only significant change required in later developments is the addition of a side condition to the independent implementability rule:

$$\begin{aligned}&\forall I_1, I_2.E_1 \models I_1 \leqslant S_1 \text{ and } E_2 \models I_2 \leqslant S_2 \text{ and} \\ &\quad in_{I_1} \cap out_{S_2} \subseteq in_{S_1} \text{ and } in_{I_2} \cap out_{S_1} \subseteq in_{S_2} \text{ implies } E \models I_1|I_2 \leqslant S \ . \quad (6)\end{aligned}$$

This is the very same side condition that Alfaro and Henzinger add to independent implementability in order to support contravariant signature extensions. It ensures that even though the implementation allows additional inputs, it will only be used as described in this interface. The other components will not communicate with it on these additional inputs.

7 Other Related Work

Our work relates directly to the original version of interface automata [5, 6], which was later extended with time and resource information in [13] and [14]. To strengthen the case, we have used some examples from [6] adapting them to our framework, and aligned the terminology with [5, 6] as much as possible. Another approach to compatibility for blocking-services is taken by Rajamani and Rehof in [2] targeting compatibility of web services. We work in the input-enabled asynchronous setting of I/O-automata [15], which is semantically closer to implementations of embedded systems. To the best of our knowledge similar properties have not been studied in the I/O automata community yet.

The notion of relativized refinement and equivalence, or more precisely simulation and bisimulation, is due to Larsen [16, 17]. It was so far applied in the setting of protocol verification [18], automatic testing [19] and modeling software product lines [7]. Here we adapt it to a language inclusion based refinement.

The general method of solving systems of behavioral equations using disjunctive modal transition systems and bisimulation as a requirement was published in [20]. The method presented in section 5 is an adaptation of this earlier work to an input-enabled setting and language-inclusion based refinement. The original method does not assume determinism of processes in the system of constraints.

The preliminary version of this paper [21] featured a stronger definition of mutual deadlock freeness: $E|S_1 \leqslant E_2$ and $E|S_2 \leqslant E_1$. Being stronger, this formulation also implies independent-implementability, but it rules out many useful compositions as incompatible. The relativized version proposed here (2) is weaker, but still strong enough to imply independent implementability. As we have seen in the previous section, it behaves reasonably allowing roughly the same kind of compatible interfaces as interface automata. The present paper, completely rewritten, reworks the theory with this new characterization, adding associativity, refinement of interfaces, a new method for solving systems of inequalities, contravariant signature extension, and the correspondence to interface automata.

8 Conclusion

We have proposed an interface theory for distributed networks of asynchronous components modeled as I/O automata. The characteristic feature of our interfaces is an explicit separation of assumptions from guarantees. Apart from the usual engineering advantages offered by such a separation of concerns, it also allows modeling of families of interfaces implemented by software product lines.

We demonstrated that it is possible to build a reasonably behaved interface theory in an input-enabled setting, with language inclusion as refinement. We emphasize that our derivation of interface composition is systematic: we state requirements for composition and reduce the problem to finding a solution of a corresponding system of behavioral inequalities. We also discuss the generality of the constructed interface, concluding that it exhibits the weakest assumptions and the strongest guarantees that are possible with our requirements. Finally we describe a method for solving systems of inequalities arising in our setup and draw a formal correspondence between the present work and interface automata.

References

1. Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. In: POPL 2001, ACM Press (2001)
2. Rajamani, S.K., Rehof, J.: Conformance checking for models of asynchronous message passing software. In Brinksma, E., Larsen, K.G., eds.: 14th International Conference on Computer Aided Verification (CAV). Volume 2404 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 166–179
3. Lee, E.A., Xiong, Y.: A behavioral type system and its application in Ptolemy II. Formal Aspects of Computing Journal (2004) Special issue on Semantic Foundations of Engineering Design Languages.
4. Lee, E.A., Zheng, H., Zhou, Y.: Causality interfaces and compositional causality analysis. [22]

5. Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), Vienna, Austria, ACM Press (2001) 109–120
6. Alfaro, L., Henzinger, T.A.: Interface-based design. In: In Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School, Kluwer Academic Publishers (2004)
7. Larsen, K.G., Larsen, U., Wąsowski, A.: Color-blind specifications for transformations of reactive synchronous programs. In Cerioli, M., ed.: Proceedings of FASE, Edinburgh, UK, April 2005. LNCS, Springer-Verlag (2005)
8. Hoare, C.: Communicating Sequential Processes. International Series in Computer Science. Prentice Hall (1985)
9. Maier, P.: Compositional circular assume-guarantee rules cannot be sound and complete. In Gordon, A., ed.: Foundations of Software Science and Computational Structures: 6th International Conference, FOSSACS 2003. Volume 2620 of Lecture Notes in Computer Science., Springer-Verlag (2003) 343–357
10. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, IEEE Computer Society (1988) 203–210
11. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. 2nd edn. Addison-Wesley (2001)
12. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.: Alternating refinement relations. In Sangiorgi, D., de Simone, R., eds.: Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98). Volume 1466 of Lecture Notes in Computer Science., Springer-Verlag (1998) 163–178
13. Alfaro, L., Henzinger, T., Stoelinga, M.I.A.: Timed interfaces. In Sangiovanni-Vincentelli, A., Sifakis, J., eds.: EMSOFT 02: Proc. of 2nd Intl. Workshop on Embedded Software. Lecture Notes in Computer Science, Springer (2002) 108–122
14. Chakabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.I.A.: Resource interfaces. In Alur, R., Lee, I., eds.: EMSOFT 03: 3rd Intl. Workshop on Embedded Software. Lecture Notes in Computer Science, Springer (2003)
15. Lynch, N.: I/O automata: A model for discrete event systems. In: Annual Conference on Information Sciences and Systems, Princeton University, Princeton, N.J. (1988) 29–38
16. Larsen, K.G.: Context Dependent Bisimulation Between Processes. PhD thesis, Edinburgh University (1986)
17. Larsen, K.G.: A context dependent equivalence between processes. Theoretical Computer Science **49** (1987) 184–215
18. Larsen, K.G., Milner, R.: A compositional protocol verification using relativized bisimulation. Information and Computation **99** (1992) 80–108
19. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using UPPAAL. In: Formal Approaches to Testing of Software (FATES), Linz, Austria. September 21, 2004. Volume 1644 of Lecture Notes in Computer Science., Springer-Verlag (2005)
20. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: Fifth Annual IEEE Symposium on Logics in Computer Science (LICS), 4–7 June 1990, Philadelphia, PA, USA. (1990) 108–117
21. Larsen, K.G., Nyman, U., Wąsowski, A.: Interface input/output automata: Splitting assumptions from guarantees. [22]
22. Hermanns, H., Rehof, J., Stoelinga, M.I.A., eds.: Workshop Procedings FIT 2005: Foundations of Interface Technologies. ENTCS, Elsevier Science Publishers (2005)