Interface Input/Output Automata: Splitting Assumptions from Guarantees

Larsen, Kim Guldstrand; Nyman, Ulrik; Wasowski, Andrzej

Publication date:
2006

Document Version
Early version, also known as pre-print

Interface Input/Output Automata: Splitting Assumptions from Guarantees

Kim G. Larsen [1] Ulrik Nyman [1] Andrzej Wąsowski [1]

Center for Embedded Software Systems (CISS), Aalborg University, Denmark

Abstract

We propose a new look at one of the most fundamental types of behavioral interfaces: discrete time specifications of communication—directly related to the work of de Alfaro and Henzinger [2]. Our framework is concerned with distributed non-blocking asynchronous systems in the style of Lynch's I/O-automata [11], relying on a context dependent notion of refinement based on relativized language inclusion.

There are two main contributions of the work. First, we explicitly separate assumptions from guarantees, increasing the modeling power of the specification language and demonstrating an interesting relation between blocking and non-blocking interface frameworks. Second, our composition operator is systematically and formally derived from the requirements stated as a system of inequalities. The derived composed interfaces are maximal in the sense of behavior, or equivalently are the weakest in the sense of assumptions. Finally, we present a method for solving systems of inequalities as used in our setup.

Key words: software design, programming languages, components

1 Introduction

Interfaces play an essential role in any software engineering methodology supporting component based system development. Traditionally static type information, type checking, annotations and inference have been used to decide whether or not two components are compatible. Static types, or static interfaces, are necessarily very conservative and usually very weak. Most typically type correctness implies only simple safety properties and rejects many pairs of components as incompatible, even if they could effectively cooperate in practice. The goal of the work presented in this article is grossly the same as that of de Alfaro's and Henzinger's in [2,3]: to provide a more expressive way of describing component interfaces, especially for model driven development processes, when most typically finite state models can be used.

[1] Email: {kgl,ulrik,wasowski}@cs.aau.dk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

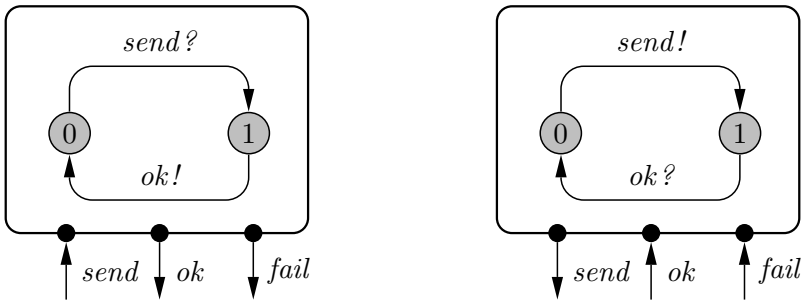
Following our predecessors, we believe that enriching interfaces with descriptions of behaviors gives enough power to meet the typical requirements. Thus we choose to model behavior of the components using automata. We explicitly split the assumptions and guarantees about a given component into two different automata though. Each interface consists of an environment and a specification. An *environment* automaton describes the assumptions that the component makes about the behavior of its surroundings. A *specification* automaton describes the guarantees that the component gives about the output that it will deliver.

A significant advantage of composite interfaces is that one of the parts can be changed without changing the other part. Thus the same assumptions can be used for multiple interfaces. It has been argued before [8] that such a setup is useful for modeling software product lines: a family of component variants may be specified using a single specification (guarantee) and multiple environmental restrictions. A clever compiler may use such a restricted interface to derive specialized versions of components from a single source code.

An interesting theoretical side effect of our exposition, is an implicit correspondence drawn between blocking and non-blocking interface theories. A single blocking interface automaton of [2] expresses both the assumptions of a component and the guarantees that it provides. When an interface is not able to accept some input, it effectively assumes that any compatible environment will never provide it. Once we make our systems non-blocking, the same effect is achieved by enriching interfaces with an explicit description of permissible behavior of the surroundings (the environment component of the interface in our case). We demonstrate that in this way a very similar kind of theory is obtained for non-blocking systems. Rightly so, the theory is obtained with solely use of standard notions such as I/O-automata, relativized refinement, modal transition systems that were available long before research in the interface theories emerged. There seems to be no indication that a blocking theory could not be obtained in the same way.

Composition of interfaces is a central construction in any interface theory: what is the interface implemented by any two components implementing two given interfaces? A new achievement of our work is that the composition is derived systematically. Instead of proposing the operator and proving properties about it, we state requirements for it in form of a system of inequalities, and derive a result of the composition systematically as a maximal solution of this system. As a result properties of composition hold by construction.

The next section introduces the framework by means of an adapted version of an example originating from [3]. In sections 3 and 4 we define the Input/Output automata and interfaces. Section 5 defines the composition of interfaces, and section 6 gives a method for solving systems of inequality used in earlier sections. Sections 7 and 8 discuss the related work and conclude.



(a) The environment Env_{Client} describing the assumptions.

(b) The specification $Spec_{Client}$ describing the guarantees.

Fig. 1. The interface *Client* consisting of Env_{Client} and $Spec_{Client}$.

2 Example

An interface of a component is given by two automata: the environment automaton describes the assumptions that are made about the environment in which the component can function, while the specification automaton states the guarantees that the component provides about the output delivered.

Figure 1 gives an interface for a simple client component having two inputs and one output. Figure 1(a) shows the environment part, Env_{Client}, which describes the assumptions that the component designer has made about the environment. The arrows going into or away from the rounded box around the automaton describe the static type (*signature*) of the environment. The specification, which for the *Client* is shown in Figure 1(b), will always have the inverse signature. The environment automaton specifies that even though the static type allows the possibility of the *fail* input, the emission of this input is disallowed by the dynamic type for all compliant execution environments. 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 never actually producing this input.

Both Env_{Client} and $Spec_{Client}$ are input enabled, meaning that they accept any input in any state, as long as the input belongs to the signature. We adopt a notational convention that transitions corresponding to ignoring an input (input self-loops) are not drawn. Thus in the above example there is one implicit transition $1 \xrightarrow{send?} 1$ in the environment Env_{Client}. One of the implicit transitions of $Spec_{Client}$ is $0 \xrightarrow{fail?} 0$.

Fig. 2 gives the environment assumptions of a component *TryTwice* that is composable with the *Client* component. Fig. 3 gives the output guarantees of *TryTwice*. The two components do not form a closed system, but are intended for use together with a *LinkLayer* component, which remains unspecified.

Observe that all the environments, which we have shown until now, are just reflections of their accompanying specifications, in the sense that what is

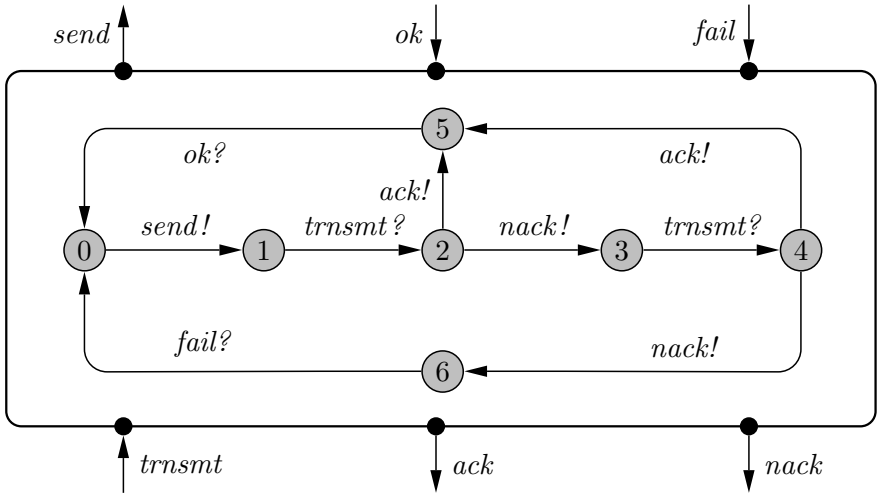


Fig. 2. The environment $Env_{TryTwice}$ describing the assumptions of the *TryTwice* component.

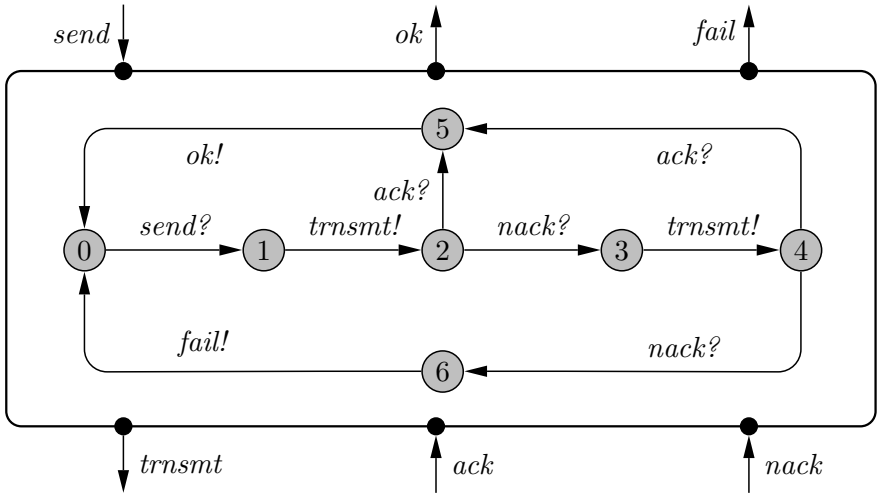


Fig. 3. The specification $Spec_{TryTwice}$ describing the guarantees of the *TryTwice* component.

an input in the specification is an output in the environment and vice versa. This is caused by the fact that they have been created directly from a blocking example in [3], which closely links the assumptions to guarantees. However, our motivation behind modeling the environment and the specification separately was that one component could be used in several environments or several components can be used in the same environment. Fig. 4 gives an alternative environment for the $Spec_{TryTwice}$ specification. This environment disallows sending of a *nack* as a response to a *trnsmt* request. It is also unable to send a *fail* event to the *Client* component. Now consider an implementation

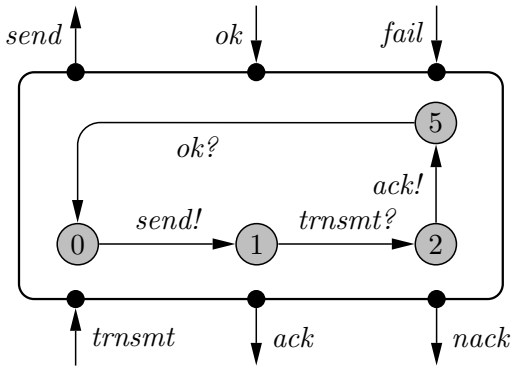


Fig. 4. The environment Env_{NoNack} with different assumptions about the environment for $Spec_{TryTwice}$.

of a communication layer in which it is indicated that the implementation of *TryTwice* will run in the environment Env_{NoNack}. The actual source code of *TryTwice*'s implementation could be automatically specialized to the environment of this specific configuration. In this particular case the error handling code could be removed as the component no longer needs to be able to handle a *nack* reply.

The two environments impose different restrictions on a possible *LinkLayer* component. The original $Env_{TryTwice}$ together with $Spec_{TryTwice}$ requires that *LinkLayer* does not send two *nack* replies in a row, whereas the Env_{NoNack} environment would enforce that the *LinkLayer* never sends a *nack* reply at all. At the same time $Env_{TryTwice}$ composed with the *Client* will never produce two *nack* messages sent in a row, as this may lead to a failure in the *Client* (the environment may produce a disallowed *fail*).

3 Input/Output Automata

Definition 3.1 An I/O automaton $A = (states_A, start_A, in_A, out_A, steps_A)$ is a 5-tuple, where $states_A$ is a set of states, $start_A \subseteq states_A$ is a non empty set of initial states, in_A is a set of input actions, out_A a set of output actions ($in_A \cap out_A = \emptyset$), and $steps_A \subseteq states_A \times (in_A \cup out_A \cup \{\tau\}) \times states_A$ is a set of transitions, where τ is a fixed internal action contained in neither in_A nor out_A. I/O automata are input enabled: for every state s and any input action $i \in in_A$ there exists a state s' and a transition $(s, i, s') \in steps_A$.

It is convenient to abbreviate $in_A \cup out_A$ as ext_A and $ext_A \cup \{\tau\}$ as act_A. In order to increase clarity we often explicitly suffix actions with direction of communication. If $(s, a, s') \in steps_A$, then we write $s \xrightarrow{a!} s'$ if $a \in out_A$, and $s \xrightarrow{a?} s'$ if $a \in in_A$. Notice that labels $a!$ and $a?$ still denote exactly the same action, and we can drop the suffixes whenever the direction of communication is clear from the context or irrelevant.

Definition 3.2 An execution of an I/O-automaton A starting in a state s_0 is a finite sequence of labels $s_0, a_0, s_1, a_1, s_2, a_2, \ldots, s_{n-1}, a_{n-1}, s_n$ such that all s_i's are members of $states_A$, all a_i's are members of act_A and for every $k = 0 \ldots n-1$ it is the case that $s_k \xrightarrow{a_k}_A s_{k+1}$.

Definition 3.3 A trace σ of an I/O-automaton A is an execution ψ of A starting in a state $s_0 \in start_A$, with all the states and internal τ actions deleted:

$$\sigma = \psi \upharpoonright ext_A$$

where $\sigma \upharpoonright X$ denotes the trace created from σ by removing all symbols that are not in the set X. The set of all traces of automaton A is denoted Tr_A.

Two I/O-automata A and B are *syntactically composable* if their input and output sorts do not overlap: $in_A \cap in_B = \emptyset$ and $out_A \cap out_B = \emptyset$. Two syntactically composable automata $A = (states_A, start_A, in_A, out_A, steps_A)$ and $B = (states_B, start_B, in_B, out_B, steps_B)$ can be composed into a single automaton $A|B = (states_{A|B}, start_{A|B}, in_{A|B}, out_{A|B}, steps_{A|B})$, where $states_{A|B} = states_A \times states_B$, $start_{A|B} = start_A \times start_B$, $in_{A|B} = in_A \cup in_B \setminus out_A \setminus out_B$, $out_{A|B} = out_A \cup out_B \setminus in_A \setminus in_B$, and $steps_{A|B}$ are all the transitions defined according to the following rules:

$$\frac{s_1 \xrightarrow{a} s_1' \quad a \in ext_A \setminus ext_B}{(s_1, s_2) \xrightarrow{a} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{a} s_2' \quad a \in ext_B \setminus ext_A}{(s_1, s_2) \xrightarrow{a} (s_1, s_2')} \qquad \frac{s_1 \xrightarrow{a} s_1' \quad s_2 \xrightarrow{a} s_2'}{(s_1, s_2) \xrightarrow{\tau} (s_1', s_2')}$$

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

The standard notion of refinement for I/O automata is based on language inclusion:

Definition 3.4 An I/O-automaton A refines (implements) an I/O-automaton B, written $A \leqslant B$, if they have the same signatures, $in_A = in_B$ and $out_A = out_B$, and the language of A is included in the language of B: $Tr_A \subseteq Tr_B$.

4 Interfaces

An interface is a specification of services *provided* (or *guaranteed*) by a component and a specification of *assumptions* under which these services will be provided. To emphasize this dual nature of interfaces, we define an interface model to be a pair (E, S) of I/O-automata: E modeling the environmental assumptions, and S modeling the provided guarantees.

Definition 4.1 A pair of I/O-automata (E, S) is an interface if E models an environment closing a specification S, 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 may only, if it is communicating with a component conforming to E, provide the behaviors of S. The behavior of I on sequences

of inputs that cannot be provided by E is not constrained. The notion of language inclusion (Def. 3.4) is not suitable for expressing this kind of context-dependent refinement. To remedy this deficiency we introduce a relativized form of refinement that directly treats the two parts of interfaces:

Definition 4.2 An I/O-automaton I refines an I/O-automaton S in the environment E, written $E \models I \leqslant S$, iff both I and S are syntactically composable with E ($in_I = in_S = out_E$ and $out_I = out_S = in_E$) and

$$Tr_E \cap Tr_I \subseteq Tr_E \cap Tr_S$$

We say that I implements the interface (E, S), if I refines S in E ($E \models I \leqslant S$).

If (E, S) is an interface, then (F, S) is an interface with *stronger assumptions* if F is an environment that can see more differences between potential implementations than E can see. In other words F allows a smaller set of implementations than E. We will also say that (E, S) has *weaker assumptions* than (F, S) (allows a bigger set of implementations). We formalize this intuition by introducing an information ordering on environments:

Definition 4.3 An I/O-automaton F is more discriminating than an I/O-automaton E, written $E \sqsubseteq F$, if F can distinguish more systems than E (so F-relativized refinement is a stronger relation than E-relativized refinement):

$$E \sqsubseteq F \quad \text{iff} \quad \forall S_1, S_2. F \models S_1 \leqslant S_2 \Rightarrow E \models S_1 \leqslant S_2$$

There exists a very powerful and simple characterization of discrimination based on trace inclusion:

Theorem 4.4 *Let E, F be two I/O-automata such that $in_E = in_F$, $out_E = out_F$, and in_E is nonempty. Then $E \sqsubseteq F$ if and only if $E \leqslant F$.*

Note that while the definition of discrimination is intentional and abstract (quantifies over all possible systems), the characterization given by Theorem 4.4 is reduced to a machine checkable procedure like model checking. Moreover the proofs involving discrimination can now be reduced to language inclusion proofs, which greatly simplifies further developments.

The notion of discrimination and its simple characterization will soon prove fundamental. If $E \sqsubseteq F$ and some implementation I implements (F, S) it is also known to implement (E, S). So we can reason about the classes of environments that I is suitable for. Moreover such strenghtening of the environment part of the interface allows performing optimizations on implementations: if I implements (E, S) in general, but in the specific network of distributed systems I interacts with an environment that is not more discriminating than F, the compiler is allowed to optimize the implementation with respect to this specific environment, for example removing unnecessary behavior.

Perhaps most importantly for the current work, we will use discrimination as the order on the space of automata, which will be instrumental to

characterize the power of the interfaces synthesized in the framework.

5 Interface Compositions

We would like to approximate compositions of components by compositions of their interfaces. For any two compatible interfaces we should be able to derive an interface of their composition, or more precisely the one that is implemented by any two implementations of the respective interfaces.

We say that two interfaces are *syntactically composable* if the I/O-automata comprising them are pointwise syntactically composable (see p. 6). This guarantees that any two components I_1 and I_2 implementing two syntactically composable interfaces (E_1, S_1) and (E_2, S_2) respectively, are also syntactically composable. The question that we want to address is whether I_1 and I_2 are also *dynamically compatible*, so that I_1 does not violate the environmental assumptions of E_2 and that I_2 does not violate the environmental assumptions of E_1.

Moreover we shall be interested in establishing what is the interface implemented by $I_1|I_2$, if they are indeed compatible. We may be tempted to say that the interface of the composed system is the composition of interfaces: $(E_1|E_2, S_1|S_2)$. Unfortunately the construction proposed in such an ad hoc, unmotivated way, will likely be unsound. [2] Instead we state the requirements for the resulting composition intentionally, in general terms. We want to require that the result of composition of (E_1, S_1), with (E_2, S_2) is an interface $(E, S_1|S_2)$, such that composition of *any two* implementations I_1, I_2 of (E_1, S_1) and (E_2, S_2), implements $(E, S_1|S_2)$:

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

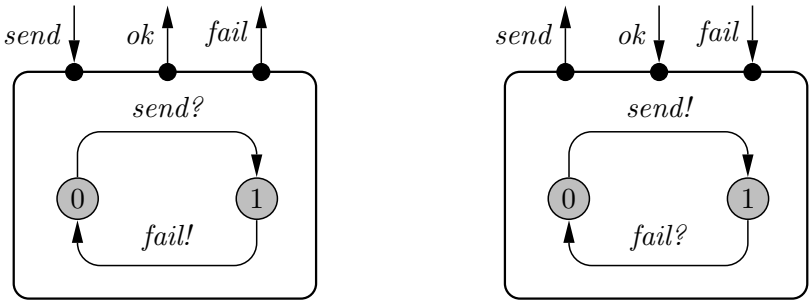
We shall hope that there exists a maximal such environment E with respect to $\sqsubseteq$. In such a case no other environment F satisfying (1) could be more critical about comparing the implementations to their specifications. This corresponds to E being the *weakest assumption* possible for the composed interface, which is a highly desirable property of any composition operator.

Finally the environment satisfying requirement (1) may not always exist. For example 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*.

Figure 5 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.

Let us now continue more formally with a theorem stating that the requirement (1) given above is well defined, i.e: we can find a unique solution

[2] Indeed the construction mentioned here uses too weak assumptions to be generally correct.



(a) $Env_{AlwaysFail}$ (b) $Spec_{AlwaysFail}$

Fig. 5. The interface *AlwaysFail*.

to it, where uniqueness is understood in terms of the discriminating power:

Theorem 5.1 *For any two interfaces* (E_1, S_1) *and* (E_2, S_2)*, if there exists an environment* E *such that*

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

then there also exists a greatest such E *with respect to* $\sqsubseteq$*.*

The above theorem can be treated as a sanity check on our requirements for interface composition. Still, although sane, the requirements themselves seem very general. They include a universal quantification over all possible implementations. Even if the requirement is known to have a solution with a unique set of traces in the general case, the existence proof is not constructive, or frankly, it is completely useless if one wanted to implement the framework. For that reason we propose a conservative characterization of the composition operator, and then develop it formally, showing a constructive algorithm computing it.

Intuitively we should require that any environment E satisfying (1) should not force S_1 into behavior towards S_2 that would violate S_2's environmental assumptions. Similarly E should not drive S_2 into behavior that would violate S_1's environmental assumptions expressed in E_1. This intuition brings us to the following system of behavioral inequalities:

$$\begin{cases} E|S_1 \sqsubseteq E_2 \\ E|S_2 \sqsubseteq E_1 \end{cases} \qquad (3)$$

It turns out that this characterization subsumes the first one. Any solution of (3) also satisfies (1), or alternatively the set of solutions of (3) is embedded in the lattice of solutions of (1):

Theorem 5.2 (Soundness) *Consider any two interfaces* (E_1, S_1)*,* (E_2, S_2)

and an environment E *such that*

$$\begin{cases} E|S_1 \sqsubseteq E_2 \\ E|S_2 \sqsubseteq E_1 \end{cases}$$

For any two I/O-automata I_1 *and* I_2 *if*

$$E_1 \models I_1 \leqslant S_1 \text{ and } E_2 \models I_2 \leqslant S_2$$

then also

$$E \models I_1|I_2 \leqslant S_1|S_2 \ .$$

Now briefly consider the theorem in the opposite direction. Can we indeed prove that the set of solutions of (1) is the same as the set of solutions to (3)? In fact this is not possible in the general case, as E_1 and E_2 can be made artificially small: in such a case there will be many E's that satisfy (1), but do not satisfy (3). Consider the following counter example.

Let E_1 and E_2 both be *mute* environments, i.e. they do not produce any outputs, despite their signatures containing some. A *mute* environment M consists of a single state with self-loop transitions for all the inputs contained in its signature and no other transitions. At the same time let S_1, S_2 be *universal* specifications, i.e. they can produce all input-enabled traces allowed by their signatures. A *universal* specification U consists of a single state with self-loops for all inputs and outputs contained in the signature.

We can easily conclude that $E_1 \models I_1 \leqslant S_1$ for any I_1 (because S_1 has all possible traces and thus implies $Tr_{I_1} \cap Tr_{E_1} \subseteq Tr_{S_1} \cap Tr_{E_1}$ for all I_1). Similarly $E_2 \models I_2 \leqslant S_2$ for any I_2. Also $E \models I_1|I_2 \leqslant S_1|S_2$ for any statically suitable E because of the maximality of S_1 and S_2. Since the latter holds for any E it also holds for the universal (the maximal) one. But then $E|S_1 \not\leqslant E_2$ because the left hand side of the inequality is universal while the right hand side is mute. By Theorem 4.4 we arrive at $E|S_1 \not\sqsubseteq E_2$. So a universal E is a valid set of assumptions for the interface $(E, S_1|S_2)$ according to (1), but it cannot be derived as a solution of (3) due to unnecessary restrictiveness of E_1 and E_2.

We conjecture that it is possible to prove a restricted form of the completeness theorem (the reverse of Thm. 5.2), where some assumptions about the strength of the environmental parts of interfaces are made (namely that E_1 and E_2 are not too strong with respect to S_1 and S_2). Nevertheless we envision that this result would only be of theoretical interest. In practice the designer wants to have the freedom to produce his interfaces by combining variants of environments with the same system specification and vice versa, without worrying about their relative strength. Thus, it is necessary to allow non-maximal versions of environment components.

Consider the above example once again. The problem with E being universal is that it obeys our intended property (1), but still is able to drive I_1 and

I_2 into some states that environments satisfying (3) would never be able to force them into. This cannot possibly cause any correctness problems because of the generality of S_1 and S_2 that indeed allow all possible behavior. Nevertheless it may affect the product line derivation process. A clever compiler can specialize I_1 and I_2 with respect to E_1 and E_2 so that the size of the generated code (or other objective function) is minimized, without E_1 and E_2 noticing any difference. If we then use such specialized implementations in the composition $(I_1|I_2)$, the synthesized general environment E will potentially see the difference between their original versions and the specialized ones. For that reason it is better for the interface derivation framework to respect the instructions of the designer and avoid synthesizing environment components of interfaces that are more general (weaker) than the environment components of interfaces being composed. So ultimately we recommend adhering to the stronger characterization (3). After all the designer might have strengthened the assumptions in E_1 and E_2 intentionally.

Last but not least we have to consider the case where the interfaces are incompatible. If no E exists that can ensure that the assumptions of the components are met then we term the two components incompatible, as given by the following definition:

Definition 5.3 Interfaces (E_1, S_1), (E_2, S_2) are incompatible if there is no I/O-automaton E such that:

$$E|S_1 \sqsubseteq E_2 \text{ and } E|S_2 \sqsubseteq E_1 \ .$$

The method for solving systems of inequalities presented in the next section will also detect this situation.

6 Systems of Behavioral Inequalities

As we have just argued, computing compositions of interfaces requires a method for finding solutions of systems of linear inequalities. In particular we are interested in systems of inequalities of the following form:

$$\mathcal{C}(E) \ : \ \begin{cases} E|S_1 \sqsubseteq F_1 \\ \quad \vdots \\ E|S_m \sqsubseteq F_m \end{cases} \tag{4}$$

where $\{S_i\}_{i=1..m}$ and $\{F_i\}_{i=1..m}$ are constants (fixed known I/O-automata) and E is a single unknown. We are interested in finding one of the greatest such E's with respect to $\sqsubseteq$. If no solutions exist then incompatibility between components should be reported.

Notice that for all i such that $in_{E|S_i}$ is empty (and correspondence between refinement and simulation breaks) the ith constraint can be removed from the system. It does not restrict E's behavior in any way. Ultimately, thanks

to Theorem 4.4, we can restrict our problem to solving inequalities of the following kind:

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

where all $in_{E|S_i}$ sets are nonempty. 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. It should be emphasized though, that this assumption is not inherent to the method.

We should now state a property similar to Theorem 5.1, but formulated for systems of inequalities instead of our intentional requirements for composition. This time we expand it to any number of constraints and do not require that all the I/O-automata come from the same interfaces.

Theorem 6.1 *Let $\{(E_i, S_i)\}_{i\in 1..m}$ be a finite set of interfaces, and $\{F_i\}_{i\in 1..m}$ be a finite set of environments, such that for each $i \in 1..m$. $in_{E_i} = in_{F_i} \neq \emptyset$ and $out_{E_i} = out_{F_i}$. Let $\mathcal{C}(E)$ be a system of inequalities containing a constraint $E|S_i \leqslant F_i$ for each $i \in 1..m$:*

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

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 $\sqsubseteq$.

We solve the systems of inequalities by first constructing a corresponding *modal transition system*, and then choosing a maximal solution from its states and transitions. Intuitively modal transition systems can be seen as I/O-automata with transition set partitioned into two *may* and *must* classes.

Definition 6.2 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.

Systems of inequalities can be seen as sets of constraint pairs $\{(S_1, F_1), \ldots, (S_m, F_m)\}$ over the solution E. The constraints evolve over time, when any of their components, including the unknown E, take some actions. This evolution comprises not only state changes of the I/O-automata, but also removing and introducing constraints. Clearly, legal actions of the unknown component E in any of its states are dependent on the states of the constraints—on what all the 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 should give rise to *may* transitions in the modal transition system. While all the steps that are strictly required give rise to corresponding *must* transitions.

Formally a set of constraints $\{(S_1, F_1), \ldots, (S_m, F_m)\}$ induces a modal transition system $\mathcal{E} = (Q, A_0, \rightarrow_{may}, \rightarrow_{must})$ where Q is the set of constraints over states of S_i's and F_i's enriched with a distinct primitive constraint FALSE, A_0 is equal to the set $\{(S_1, F_1), \ldots, (S_m, F_m)\}$ 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:

(i) For all $(S, F) \in E$ such that $a \in out_E \setminus in_S$: if there exists F' such that $F \xrightarrow{a!} F'$ then $(S, F') \in E'$ else FALSE $\in E'$

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

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

(i) For all $(S, F) \in E$ such that $a \in in_E \setminus out_S$ and $F \xrightarrow{a?} F'$ also $(S, F') \in E'$

(ii) For all $(S, F) \in E$ such that $a \in in_E \cap out_S$ if $S \xrightarrow{a!} S'$ then $(S', F) \in E'$

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

(i) whenever $(S, F) \in E$ and $a \in ext_S \cap ext_F$ and $S \xrightarrow{a} S'$ and $F \xrightarrow{a} F'$ then also $(S', F') \in E$.

(ii) whenever $(S, F) \in E$ and $a \in ext_S \cap ext_F$ and $S \xrightarrow{a!} S'$ and there is no F' such that $F \xrightarrow{a!} F'$ then also FALSE $\in E$.

Definition 6.3 A state consistency relation $\mathcal{S}$ over a modal transition system $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ is a subset of Q such that if $E \in \mathcal{S}$ then FALSE $\notin E$ and whenever $E \xrightarrow{a}_{must} E'$ then $E' \in \mathcal{S}$.

Definition 6.4 A consistent set of transitions $\mathcal{T}$ of a modal transition system $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ with respect to a state consistency relation $\mathcal{S}$ is a maximal subset of $\rightarrow_{may} \cup \rightarrow_{must}$, where whenever $(s, a, s') \in \mathcal{T}$ then $s \in \mathcal{S}$ and $s' \in \mathcal{S}$.

Theorem 6.5 (Soundness & Completness) *Let $\mathcal{C}(E)$ be a system of inequalities*

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

and $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ *be a 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* $\mathcal{E}$*,*

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

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.*

In practice a greatest solution can be found without constructing the entire modal transition system induced by $\mathcal{C}(E)$. Instead a fixpoint computation is applied, which only constructs the consistent part. We start with the set of states Q equal to the initial set of inequalities and then add transitions and new states applying the transition rules and the closure rules described above. If a state containing FALSE is found then the exploration is not pursued beyond that point. Also all *may* transitions targeting this state, so far and in the future, should be dropped. If there are any *must* transitions targeting a FALSE state then not only the transitions leading to it, but also their source states (transitively backwards over must transitions) are dropped. The computation reaches a fixpoint on a greatest solution of $\mathcal{C}(E)$, or on the empty sets of states and transitions, when the system of inequalities has no solutions at all.

7 Related Work

Interface automata have been originally proposed by de Alfaro and Henzinger in [2,3], and then extended with time and resource information in [1] and [4]. Our work relates directly to the original, discrete version of [2]. It can be seen as an alternative, more systematic approach of reaching similar results. To strengthen the case, we have used some examples from [3] in this presentation, adapting them to our framework. We have also aligned our terminology with [2] as much as possible.

Alfaro's and Henzinger's setting considers blocking systems, natural in modeling of transaction services (like web services). A similar approach is taken in [12], also targeting compatibility of web services. We work in the input-enabled asynchronous setting of I/O-automata [11], which is closer to implementation of embedded systems, and preferred in design of distributed algorithms. Interestingly we reach very similar results to [2], as we exploit models of environments to encode information similar to blocking.

Another advantage of our work is that we explicitly guarantee maximality

of the interface composition, and that our composition operator is formally derived as a solution to a system of constraints, while in [2] it is proposed in a more arbitrary fashion. Last, but not least, the explicit separation of assumptions (environments) and guarantees (specifications), absent in [2] increases the modeling capabilities allowing modeling variants of similar components, by embedding them in various environments [8]. Interestingly the authors of [3] discuss stateless interfaces with explicit assume/guarantee separation, but they do not take this division all the way into the behavioral framework.

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 (more precisely simulation and bisimulation) is due to Larsen [5,6]. It was also applied in the setting of protocol verification [10], automatic program testing [9] and modeling software product lines [8]. Here we have adapted it to a relativized 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 [7]. The method presented in section 6 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.

8 Conclusion & Future Work

We have proposed an interface theory for networks of distributed asynchronous components modeled as input-enabled I/O automata. The very characteristic feature of our interfaces was an explicit separation of assumptions from guarantees. Apart from the usual software engineering advantages offered by such separation, here it allows a possibility of describing families of related interfaces that model families of related components (product lines).

Our deriviation of interface composition has been systematic: we have stated requirements for composition and reduced the problem to finding a solution of a corresponding system of behavioral inequalities. We have described an automatic method for solving systems of inequalities arising in our setup.

In the future we would also like to investigate systematic deriviation of the specification part of an interface composed from (E_1, S_1) and (E_2, S_2). In the present work we have somewhat arbitrarily decided to set it to $S_1|S_2$, whereas it would be interesting to arrive at this result from general requirements in a more systematic manner. The second awaited extension is the support for contravariant refinement: namely one that would allow refinement between specification of various signatures in the spirit of [2]. Finally it may prove interesting to weaken the power of environment components by introducing concepts similar to color-blindness [8], in order to increase the modeling power for defining software product lines.

References

[1] Alfaro, L., T. Henzinger and M. I. A. Stoelinga, *Timed interfaces*, in: A. Sangiovanni-Vincentelli and J. Sifakis, editors, *EMSOFT 02: Proc. of 2nd Intl. Workshop on Embedded Software*, Lecture Notes in Computer Science (2002), pp. 108–122.

[2] Alfaro, L. and T. A. Henzinger, *Interface automata*, in: *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)* (2001), pp. 109–120.

[3] Alfaro, L. and T. A. Henzinger, *Interface-based design*, in: *In Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School* (2004).

[4] Chakabarti, A., L. de Alfaro, T. A. Henzinger and M. I. A. Stoelinga, *Resource interfaces*, in: R. Alur and I. Lee, editors, *EMSOFT 03: 3rd Intl. Workshop on Embedded Software*, Lecture Notes in Computer Science (2003).

[5] Larsen, K. G., "Context Dependent Bisimulation Between Processes," Ph.D. thesis, Edinburgh University (1986).

[6] Larsen, K. G., *A context dependent equivalence between processes*, Theoretical Computer Science **49** (1987), pp. 184–215.

[7] Larsen, K. G., *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, pp. 108–117.

[8] Larsen, K. G., U. Larsen and A. Wąsowski, *Color-blind specifications for transformations of reactive synchronous programs*, in: M. Cerioli, editor, *Proceedings of FASE, Edinburgh, UK, April 2005*, Lecture Notes in Computer Science (2005), accepted.

[9] Larsen, K. G., M. Mikucionis and B. Nielsen, *Online testing of real-time systems using* UPPAAL, in: *Formal Approaches to Testing of Software (FATES), Linz, Austria. September 21, 2004*, Lecture Notes in Computer Science **1644** (2005).

[10] Larsen, K. G. and R. Milner, *A compositional protocol verification using relativized bisimulation*, Information and Computation **99** (1992), pp. 80–108.

[11] 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, pp. 29–38.
URL http://theory.lcs.mit.edu/tds/papers/Lynch/princeton88.pdf

[12] Rajamani, S. K. and J. Rehof, *Conformance checking for models of asynchronous message passing software*, in: E. Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science **2404** (2002), pp. 166–179.