Basic Research in Computer Science

An Interface Theory for Input/Output Automata

Kim G. Larsen
Ulrik Nyman
Andrzej Wasowski

BRICS Report Series **RS-06-11**

ISSN 0909-0878 **June 2006**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark

Telephone: +45 8942 9300
Telefax: +45 8942 5601
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/06/11/`

An Interface Theory for Input/Output Automata

Kim G. Larsen[a,b] Ulrik Nyman[b]
Andrzej Wąsowski[c*]

[a]BRICS[†], Department of Computer Science,
Aalborg University,
Fr. Bajersvej 7B, DK-9220 Aalborg Ø, Denmark

[b]CISS, Center for Embedded Software Systems,
Aalborg University,
Fr. Bajersvej 7B, DK-9220 Aalborg Ø, Denmark
{kgl,ulrik}@cs.aau.dk

[a]Computational Logic and Algorithms Group,
IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 København S, Denmark
wasowski@itu.dk

December 17, 2006

Abstract

Building on the theory of interface automata by de Alfaro and Henzinger we design an interface language for Lynch's Input/Output Automata, a popular formalism used in the development of distributed asynchronous systems, not addressed by

*Partly supported by Center for Embedded Software Systems (CISS) in Aalborg.
†Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

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. Proofs are provided in an appendix.

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 [9, 22, 18, 19] and references therein for examples).

We follow de Alfaro and Henzinger [2, 3] 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.

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

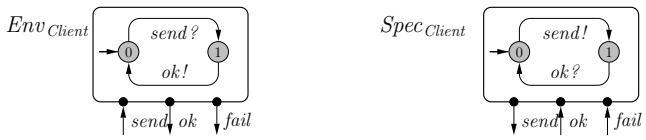


Figure 1: $Client = (Env_{Client}, Spec_{Client})$

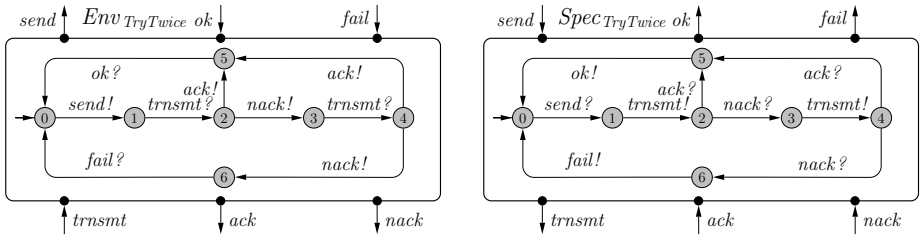


Figure 2: $TryTwice = (Env_{TryTwice}, Spec_{TryTwice})$

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 solu-

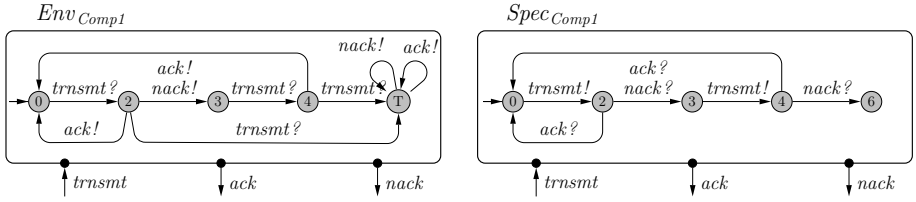


Figure 3: $(Env_{TryTwice}, Spec_{TryTwice})|(Env_{Client}, Spec_{Client}) = Comp1$

tion 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 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 [12] 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.

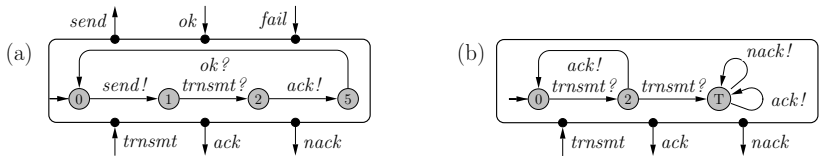


Figure 4: (a) The environment Env_{NoNack} and (b) the environment Env_{Comp2}.

As we have also argued in [12] 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 [2] 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. The appendix A contains the proofs of all our claims.

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 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 [7], 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:*

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

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* [3], *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:

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

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:

$$\begin{aligned}\text{For all } I_1, I_2.E_1 \models I_1 \leqslant S_1 \text{ and } E_2 \models I_2 \leqslant S_2 \\ \text{implies } I_1 \models E|I_2 \leqslant E_1 \text{ and } I_2 \models E|I_1 \leqslant E_2\ .\end{aligned} \tag{2}$$

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

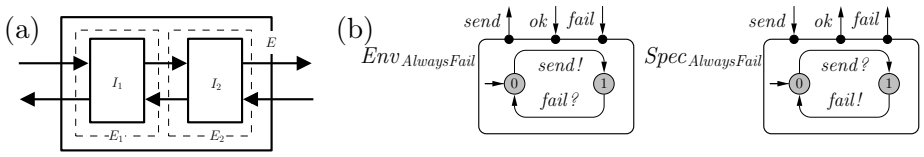


Figure 5: (a) Flowgraph for a composition of (E_1, S_1) and (E_2, S_2). (b) *AlwaysFail*

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 [21] 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* [16] 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 $F \xrightarrow{a!}\!\!\!\!\!\!/\;\;$ and $\exists P'.\, P \xrightarrow{a?} P'$ then 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* 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 $\mathcal{E}$,*

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

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 the specification $Spec_{Comp1}$ from Fig. 3 has been obtained by minimizing $Spec_{TryTwice} \mid Spec_{Client}$.

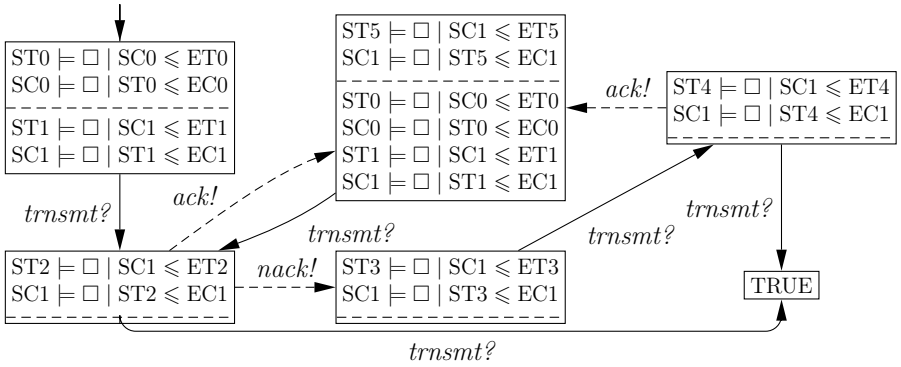


Figure 6: The resulting modal transition system for the computation of Env_{Comp1}.

6 Interface Automata

The relation of our theory to interface automata [2, 3] 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 [3] 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 [2, 3] 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 [3]). *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* $steps_S \subseteq states_S \times act_S \times states_S$ *is an input-deterministic transition relation, where* $act_S = in_S \cup out_S \cup int_S$

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. 6. 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:

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

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

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 [8, 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 $steps_A = \{((s,e),a,(s',e'))|s \xrightarrow{a} s' \text{ and } e \xrightarrow{a} e'\}$.

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 [4].

Alfaro and Henzinger choose alternating simulation to support contravariant treatment of inputs and outputs. We stress that our choice of 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:

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

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 [2, 3], which was later extended with time and resource information in [1] and [5]. To strengthen the case, we have used some examples from [3] adapting them to our framework, and aligned the terminology with [2, 3] as much as possible. Another approach to compatibility for blocking-services is taken by Rajamani and Rehof in [22] targeting compatibility of web services. We work in the input-enabled asynchronous setting of I/O-automata [20], 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 [10, 11]. It was so far applied in the setting of protocol verification [14], automatic testing [13] and modeling software product lines [12]. 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 [17]. 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 [15] 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] L. de Alfaro, T. A. 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, pages 108–122. Springer, 2002.

[2] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120, Vienna, Austria, September 2001. ACM Press.

[3] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In *In Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School*. Kluwer Academic Publishers, 2004.

[4] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *Proceedings of the Ninth International*

Conference on Concurrency Theory (CONCUR'98), volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer-Verlag, 1998.

[5] A. Chakabarti, 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. Springer, 2003.

[6] Holger Hermanns, Jakob Rehof, and Marielle I. A. Stoelinga, editors. *Workshop Procedings FIT 2005: Foundations of Interface Technologies*, ENTCS. Elsevier Science Publishers, 2005.

[7] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[8] John E. Hopcroft, Rejeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2nd edition, 2001.

[9] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *POPL 2001*. ACM Press, 2001.

[10] Kim G. Larsen. *Context Dependent Bisimulation Between Processes*. PhD thesis, Edinburgh University, 1986.

[11] Kim G. Larsen. A context dependent equivalence between processes. *Theoretical Computer Science*, 49:184–215, 1987.

[12] Kim G. Larsen, Ulrik Larsen, and Andrzej Wąsowski. Color-blind specifications for transformations of reactive synchronous programs. In Maura Cerioli, editor, *Proceedings of FASE, Edinburgh, UK, April 2005*, LNCS. Springer-Verlag, 2005.

[13] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. 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.

[14] Kim G. Larsen and Robin Milner. A compositional protocol verification using relativized bisimulation. *Information and Computation*, 99(1):80–108, 1992.

[15] Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. Interface input/output automata: Splitting assumptions from guarantees. In Hermanns et al. [6].

[16] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society, 1988.

[17] Kim Guldstrand Larsen and Liu Xinxin. Equation solving using modal transition systems. In *Fifth Annual IEEE Symposium on Logics in Computer Science (LICS), 4–7 June 1990, Philadelphia, PA, USA*, pages 108–117, 1990.

[18] E. A. Lee and Y. Xiong. 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.

[19] Edward A. Lee, Haiyang Zheng, and Ye Zhou. Causality interfaces and compositional causality analysis. In Hermanns et al. [6].

[20] Nancy Lynch. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38, Princeton University, Princeton, N.J., 1988.

[21] Patrick Maier. Compositional circular assume-guarantee rules cannot be sound and complete. In A.D. Gordon, editor, *Foundations of Software Science and Computational Structures: 6th International Conference, FOSSACS 2003*, volume 2620 of *Lecture Notes in Computer Science*, pages 343–357. Springer-Verlag, 2003.

[22] Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 166–179, Copenhagen, Denmark, July 2002. Springer-Verlag.

A Proofs

This appendix contains proofs of theorems and lemmas, along with some counterexamples for negative claims or one-way implications. *The appendix is not an integral part of the paper, and reading it is not required in order to assess the value of the results.*

Before we continue with arguing for the correctness of the claims expressed in the main matter of the paper, let us introduce several basic technicalities:

Definition 26. *Given an input set I, a set of traces Σ is I-enabled if any trace of Σ can be extended with any input of I and still remain a trace of Σ:*

$$\forall \sigma \in \Sigma.\ \forall j \in 1 \ldots |\sigma|.\ \forall i \in I. \exists \sigma' \in \Sigma. \sigma' = \sigma \cdot i\ .$$

Theorem 27. *For any I/O automaton A, the set of traces Tr_A is in_A-enabled.*

Lemma 28. *For any two syntactically composable I/O automata A and B:*

$$\begin{aligned} Tr_{A|B} = \{\sigma \restriction ext_{A|B} \mid \sigma \restriction ext_A \in Tr_A \\ \text{and } \sigma \restriction ext_B \in Tr_B \\ \text{and } \sigma \in (ext_A \cup ext_B)^*\} \end{aligned}$$

Let us switch back to the main line of the discussion now. We begin the discussion of correctness of our claims with section 3:

Proof of Theorem 6. Consider the three cases separately:

1. Let $E_1 \models I \leqslant S_1$ and $\sigma \in Tr_{E_2} \cap Tr_I$. If $\sigma \in Tr_{E_1}$ then by assumption $\sigma \in Tr_{E_1} \cap Tr_I \subseteq Tr_{E_1} \cap Tr_{S_1} = Tr_{E_2} \cap Tr_{S_2}$ and we are done. So assume that $\sigma \notin Tr_{E_1}$ and take $\sigma' \leqslant \sigma$, such that $\sigma' \in Tr_{E_1}$, and $\sigma' a \leqslant \sigma$, $\sigma' \notin Tr_{E_1}$. Now $\sigma' \in Tr_{E_1} \cap Tr_I \subseteq Tr_{E_1} \cap S_1 = Tr_{E_2} \cap Tr_{S_2}$. So $\sigma' \in Tr_{S_2} \cap Tr_{E_2}$. Due to input-enabledness $a \in out_{E_1} = in_{S_2}$, so $\sigma' a \in Tr_{S_2} \cap Tr_{E_2} = Tr_{S_1} \cap Tr_{E_1}$. A contradiction, as we required that $\sigma' a \notin Tr_{E_1}$. So $\sigma \in Tr_{E_1}$, which ultimately implies $E_2 \models I \leqslant S_2$. The proof of $E_2 \models I \leqslant S_2$ implying $E_1 \models I \leqslant S_1$ is entirely symmetric. Finally a counterexample exists witnessing that the implication of the first case of the theorem does not hold

in the converse direction (completeness). Similarly counterexamples are known that the characterization cannot be weakened, by changing equality into set inclusion and implying refinement one way.

2. The proof of the second case is trivial. It is also easy to show a counterexample for incompleteness.

3. Let $E_1 \models I \leqslant S_1$ and observe that $Tr_{E_2} \cap Tr_I = (Tr_{E_2} \cap Tr_{S_2} \cap Tr_I) \cup (Tr_{E_2} \setminus Tr_{S_2}) \cap Tr_I$. Only the second summand in the above union can violate $E_2 \models I \leqslant S_2$, but fortunately it can be shown that this summand is empty. Observe that $(Tr_{E_2} \setminus Tr_{S_2}) \cap Tr_I \subseteq (Tr_{E_1} \setminus Tr_{S_1}) \cap Tr_I$. If the left hand-side of the inclusion is non-empty, then so is the right hand side, but this contradicts $E_1 \models I \leqslant S_1$, which finishes the proof. A counterexample is known that witnesses the characterization of the third case being incomplete.

□

Proof of Theorem 8. We split the proof of the equivalence into two implication proofs.

($\Rightarrow$) Assume $test_{(E_2,S_2)} \subseteq test_{(E_1,S_1)}$. Instead of proving the implication $E_1 \models I \leqslant S_1 \implies E_2 \models I \leqslant S_2$ directly, show the contrapositive:

$$E_2 \not\models I \leqslant S_2 \text{ implies } E_1 \not\models I \leqslant S_1 \tag{7}$$

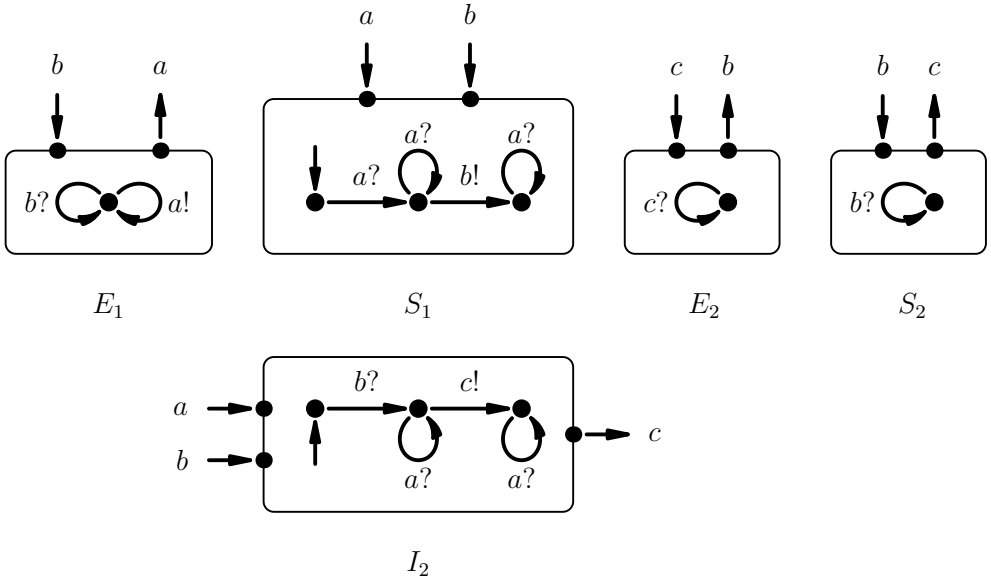
This can be done by considering a shortest trace witnessing the antecedent of (7), and observing that it belongs to $test_{(E_2,S_2)}$, which means that it also belongs to $test_{(E_1,S_1)}$ and witnesses the consequent.

($\Leftarrow$) The proof in the opposite direction proceeds by the contrapositive of the main implication:

$$test_{(E_2,S_2)} \not\subseteq test_{(E_1,S_1)} \text{ implies } \text{exists } I \text{ such that } E_1 \models I \leqslant S_1 \text{ and } E_2 \not\models I \leqslant S_2 \tag{8}$$

The counterexample I can be constructed by building an automaton around the trace witnessing the antecedent—a test of (E_2, S_2), which is not a test of (E_1, S_1)—and input-enabling it. The traces of such automaton can only be prefixes of the backbone trace σ, perhaps suffixed with finite sequences of inputs (the construction is often used see Fig. for example). After constructing I in this way it is easy to conclude that it implements (E_1, S_1) and does not implement (E_2, S_2). □

In section 4, p. 8 we claimed that for given two syntactically composable interfaces (E_1, S_1), (E_2, S_2) the interface composed of their pointwise compositions $(E_1|E_2, S_1|S_2)$ is unsound, or more precisely it violates the independent implementability rule (1). The counterexample supporting this claim is:



Observe that $E_1 \models S_1 \leqslant S_1$ and $E_2 \models I_2 \leqslant S_2$, but $E_1|E_2 \not\models S_1|I_2 \leqslant S_1|S_2$.

The following quite technical lemma states how a trace of three cooperating components can be split into traces on the respective local interfaces. Its nature is rather technical, but we rely on it in the proof of Theorem 9 and some subsequent results.

Lemma 29. *Let* $ext_C = ext_{A|B}$ *and* $\sigma \in Tr_{A|B} \cap Tr_C$. *There exists* $\sigma' \in (ext_A \cup ext_B \cup ext_C)^*$ *such that the following three properties hold:*

1. $\sigma = \sigma' \restriction ext_C \in Tr_{A|B} \cap Tr_C$
2. $\sigma' \restriction ext_B \in Tr_{A|C} \cap Tr_B$
3. $\sigma' \restriction ext_A \in Tr_{B|C} \cap Tr_A$

Moreover each of the traces constructed by restriction in the above three cases, is a result of a composition of the traces constructed in the two other cases.

Proof of the lemma 29. The first case basically repeats the assumption. The remaining cases are symmetric, so let us look just at the first of them.

By lemma 28 there exists $\sigma' \in (ext_A \cup ext_B)^*$ such that $\sigma' \restriction ext_{A|B} = \sigma$, $\sigma' \restriction ext_A \in Tr_A$, and $\sigma' \restriction ext_B \in Tr_B$. With this information and $ext_{A|B} = ext_C$ we can apply lemma 29, using the same σ' but a different configuration of restrictions, to show that $\sigma' \restriction ext_B \in Tr_{A|C}$.

The fact that the trace of each case is a composition of the traces of two other cases, follows from the way we apply lemma 28 in each of the three proofs. Incidentally the two traces that we compose while applying the lemma are always the same traces that are used in the proofs of other cases (it is essential the we always use the same σ'). □

The following lemma gives a more algebraic way of using the former observation. Intuitively it means that if we are using an automaton C in a context of A_1 and A_2, then we may soundly substitute an automaton that is smaller in the same context.

Lemma 30. *Let A_1, A_2, B, C and D be I/O automata. Then the following proof rule is sound:*

$$\frac{A_1 \models C|A_2 \leqslant D \qquad A_1|A_2 \models B \leqslant C}{A_1 \models B|A_2 \leqslant D}$$

Proof of Lemma 30. Let $\sigma \in Tr_{A_1} \cap Tr_{B|A_2}$. We need to show that $\sigma \in Tr_D$. By Lemma 29 we know that there exists $\sigma' \in (ext_B \cup ext_{A_2})^*$ such that:

1. $\sigma' \restriction ext_B \in Tr_B \cap Tr_{A_1|A_2}$,
2. $\sigma = \sigma' \restriction ext_{A_1} \in Tr_{A_1} \cap Tr_{B|A_2}$
3. $\sigma' \restriction ext_{A_2} \in Tr_{A_2} \cap Tr_{B|A_1}$

The first one of the above together with the second premise of the lemma's rule imply that $\sigma' \restriction ext_C = \sigma' \restriction ext_{A_1|A_2} \in Tr_C$. Also $\sigma' \restriction ext_{C|A_2} = \sigma' \restriction ext_{A_1} = \sigma$.

Summarizing we get that: $\sigma' \restriction ext_{C|A_2} = \sigma$, $\sigma' \restriction ext_C \in Tr_C$, and $\sigma' \restriction ext_{A_2} \in Tr_{A_2}$. By Lemma 28 we get $\sigma \in Tr_{C|A_2}$. Since also $\sigma \in Tr_{A_1}$, we conclude that $\sigma \in Tr_D$ by the first premise of the rule. □

We prove the Theorem 9 with a series of simpler claims.

Lemma 31. *Let (E_1, S_1), (E_2, S_2) be syntactically composable interfaces, E an I/O automaton such that $in_E = out_{S_1|S_2}$, $out_E = in_{S_1|S_2}$. Let I_2 have the same signature as S_2. Then $I_2 \not\models E|S_1 \leqslant E_2$ implies that either $S_2 \not\models E|S_1 \leqslant E_2$ or $E_2 \not\models I_2 \leqslant S_2$.*

Proof. Show that the trace witnessing $I_2 \not\models E|S_1 \leqslant E_2$ witnesses $S_2 \not\models E|S_1 \leqslant E_2$ or it contains a prefix witnessing $E_2 \not\models I_2 \leqslant S_2$. □

Taking the contrapositive of the above lemma (applied twice) leads us to the following corollary:

Corollary 32. *Let (E_1, S_1), and (E_2, S_2) be syntactically composable interfaces and let E be an I/O automaton with the same signature as $E_1|E_2$. Then $S_2 \models E|S_1 \leqslant E_2$ and $E_2 \models I_2 \leqslant S_2$ imply that $I_2 \models E|S_1 \leqslant E_2$. Similarly $S_1 \models E|S_2 \leqslant E_1$ and $E_1 \models I_1 \leqslant S_1$ imply that $I_1 \models E|S_2 \leqslant E_1$.*

Lemma 33. *Let (E_1, S_1), (E_2, S_2) be syntactically composable interfaces, E an I/O automaton such that $in_E = out_{S_1|S_2}$, $out_E = in_{S_1|S_2}$, and $E_1 \models I_1 \leqslant S_1$, $E_2 \models I_2 \leqslant S_2$. Then $S_2 \not\models E|I_1 \leqslant E_2$ implies that either $S_2 \not\models E|S_1 \leqslant E_2$ or $S_1 \not\models E|S_2 \leqslant E_1$.*

Proof. Take any $\sigma \in Tr_{S_2} \cap Tr_{E|I_1}$ and $\sigma \notin Tr_{E_2}$. By lemma 29 there exists $\sigma' \in (ext_E \cup ext_{I_1})^*$ such that $\sigma = \sigma' \restriction ext_{S_2} \in Tr_{S_2} \cap Tr_{E|I_1}$, $\sigma' \restriction ext_E \in Tr_E \cap Tr_{I_1|S_1}$, and $\sigma' \restriction ext_{I_1} \in Tr_{I_1} \cap Tr_{E|S_2}$.

If $\sigma' \restriction ext_{I_1} \in Tr_{S_1}$ then by lemma 28 and the above memberships we get that $\sigma' \restriction ext_{S_2} \in Tr_{E|S_1}$ (or more precisely we get $\sigma' \restriction ext_{S_2} \in Tr_{E|I_1}$ and the former follows from $\sigma' \restriction ext_{I_1} \in Tr_{S_1}$). Observe that now $\sigma' \restriction ext_{S_2} = \sigma$ witnesses $S_2 \not\models E|S_1 \leqslant E_2$.

Otherwise if $\sigma' \restriction ext_{I_1} \notin Tr_{S_1}$ then by assumptions $\sigma' \restriction ext_{I_1} \notin Tr_{E_1}$. Take σ_{E_1} to be the longest prefix of $\sigma' \restriction ext_{I_1}$ such that $\sigma_{E_1} \in Tr_{E_1}$. Since $E_1 \models I_1 \leqslant S_1$ then $\sigma_{E_1} \in Tr_{S_1}$. Let a be the next action following σ_{E_1} in $\sigma' \restriction ext_{I_1}$. Due to input-enabledness of E_1 it must be that $a \in out_{E_1} = in_{S_1}$. Due to the input-enabledness of S_1: $\sigma_{E_1} a \in Tr_{S_1}$. Observe that $\sigma_{E_1} a$ witnesses that $S_1 \not\models E|S_2 \leqslant E_1$. □

A contrapositive of the above lemma (applied twice) leads us to the following corollary:

Corollary 34. *Let (E_1, S_1), and (E_2, S_2) be syntactically composable interfaces and let E be an I/O automaton with the same signature as $E_1|E_2$ such that $S_2 \models E|S_1 \leqslant E_2$ and $S_1 \models E|S_2 \leqslant E_1$. Then $E_1 \models I_1 \leqslant S_1$ and $E_2 \models I_2 \leqslant S_2$ implies $S_2 \models E|I_1 \leqslant E_2$ and $S_1 \models E|I_2 \leqslant E_1$.*

Proof of Theorem 9. Observe that the theorem holds trivially for the left-to-right direction (completeness of the simplified characterization with respect to the one containing universal quantification). This is because $E_1 \models S_1 \leqslant S_1$ and $E_2 \models S_2 \leqslant S_2$. The simplified characterization turns out to be just a special case of the general one.

As far as the right-to-left direction (soundness) is concerned observe that by way of corollary 34 we obtain that: $S_2 \models E|I_1 \leqslant E_2$ and $S_1 \models E|I_2 \leqslant E_1$. From the former, by way of corollary 32 taking interfaces (E_1, I_1) and (E_2, S_2), we obtain $I_2 \models E|I_1 \leqslant E_2$. From the latter, by way of corollary 32 taking (E_1, S_1) and (E_2, I_2), we get $I_1 \models E|I_2 \leqslant E_1$, which finishes the proof. □

Proof of Theorem 10. Consider the contrapositive instead. For any two automata I_1 and I_2 it says that:

$$E \not\models I_1|I_2 \leqslant S_1|S_2 \implies E_1 \not\models I_1 \leqslant S_1 \vee E_2 \not\models I_2 \leqslant S_2\,.$$

So consider I_1, I_2 satisfying the antecedent. There exists a trace $\sigma \in Tr_E \cap Tr_{I_1|I_2}$ such that $\sigma \notin Tr_{S_1|S_2}$. By lemma 28 there exists $\sigma' \in (ext_{I_1} \cup ext_{I_2})^*$ such that:

1. $\sigma' \restriction ext_{I_1|I_2} = \sigma$
2. $\sigma_{I_1} := \sigma' \restriction ext_{I_1} \in Tr_{I_1}$
3. $\sigma_{I_2} := \sigma' \restriction ext_{I_2} \in Tr_{I_2}$

Now we want to execute σ_{I_1} on S_1 and σ_{I_2} on S_2. One of them must fail as otherwise σ would succeed on $S_1|S_2$ contradicting our earlier assumption.

Without loss of generality assume that σ_{I_1} fails on S_1 ($\sigma_{I_1} \notin Tr_{S_1}$) pointwise earlier than σ_{I_2} fails on S_2 (which may not fail at all). By pointwise, we do not mean that σ_{I_1} fails in fewer steps than σ_{I_2}, but that the symbol which makes σ_{I_1} fail comes earlier in the merged trace σ' than the possible failing symbol of σ_{I_2}. In other words, it is S_1 that makes $S_1|S_2$ fail on σ.

Consider a prefix $\xi = a_1 \ldots a_k$ of σ_{I_1} such that all strict prefixes of ξ are traces of S_1, $a_k \in out_{S_1}$ and $\xi \notin Tr_{S_1}$ (a_k cannot be matched by S_1).

We will show that $E_1 \not\models I_1 \leqslant S_1$. We already know that $\xi \in Tr_{I_1}$ and $\xi \notin Tr_{S_1}$. We still need to argue that $\xi \in Tr_{E_1}$. Intuitively we should use the assumption $S_1 \models E|S_2 \leqslant E_1$ in showing this, as this is the only inequality we have that can be directly used for proving that something is a trace of E_1. Unfortunately $\xi \notin Tr_{S_1}$, so we cannot use it directly.

Instead take a prefix ξ' of ξ such that $\xi = \xi' a_k$. We want to show that $\xi' \in Tr_{E|S_2}$. Consider ξ'', a prefix of σ' corresponding to ξ' (σ' embeds σ_{I_1}, ξ is a prefix of σ_{I_1}, and ξ' is a prefix of ξ). In other words: $\xi'' \restriction ext_{I_1} = \xi'$. (Note that $ext_{I_1} = ext_{S_1}$).

Then we make the following basic observations:

4. $\xi'' \restriction ext_{E|S_2} = \xi'$, because $ext_{E|S_2} = ext_{I_1}$.

5. $\xi'' \restriction ext_E \in Tr_E$, because $ext_{I_1|I_2} = ext_E$ and $\sigma \in Tr_E$, ξ'' is a prefix of σ' and $\sigma' \restriction ext_{I_1|I_2} = \sigma$.

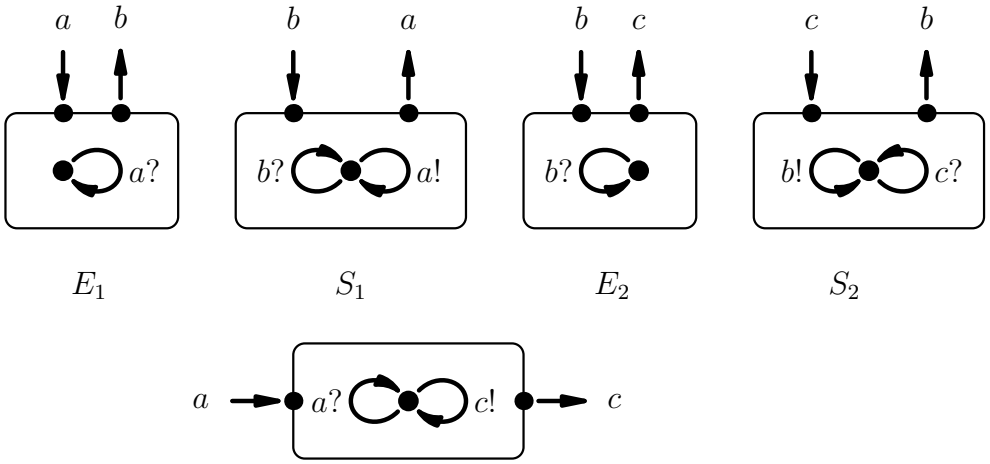
6. $\xi'' \restriction ext_{S_2} \in Tr_{S_2}$, because $\xi'' \restriction ext_{S_2}$ is a prefix of σ_{I_2}, which is also a trace of S_2, because σ_{I_2} fails later on S_2 than σ_{I_1} fails on S_1.

7. $\xi'' \in (ext_E \cup ext_{S_2})^*$, because ξ'' is a prefix of σ' and $\sigma' \in (ext_{I_1} \cup ext_{I_2})^*$ and $ext_E \cup ext_{S_2} = ext_{I_1} \cup ext_{I_2}$.

Properties 4–7 mean that ξ'' is a witness of the fact that $\xi' \in Tr_{E|S_2}$ (lemma 28). Also $\xi' \in Tr_{S_1}$ because a_k was the first action failing on S_1. From these two facts we conclude that $\xi' \in Tr_{E_1}$ by assumption that $S_1 \models E|S_2 \leqslant E_1$. Moreover since $a_k \in out_{S_1} = in_{E_1}$ and E_1 is input-enabled we have that $\xi \in Tr_{E_1}$.

Summing up: $\xi \in Tr_{E_1}$, $\xi \in Tr_{I_1}$ and $\xi \notin Tr_{S_1}$, which means that: $E_1 \not\models I_1 \leqslant S_1$, which finishes the proof (except for the dual case if σ_{I_2} fails on S_2, before σ_{I_1} fails on S_1, which we leave out due to its symmetry). □

A counterexample showing that theorem 10 cannot be extended to hold in the opposite direction:



Automata E_1, and E_2 are mute (they cannot produce any outputs), while S_1 and S_2 are universal (they generate complete languages over their alphabets). Also E is universal over its own alphabet. For this reason it is not hard to conclude that for any two implementations I_1, I_2

of the same signatures as S_1, S_2 respectively the following three implementation relations hold:

$$E_1 \models I_1 \leqslant S_1 \quad E_2 \models I_2 \leqslant S_2 \quad E \models I_1|I_2 \leqslant S_1|S_2$$

So E exhibits the independent implementability property with specification $S_1|S_2$. Nevertheless $S_1 \not\models E|S_2 \leqslant E_1$ (witnessed by a trace containing a single action b).

Theorem 35. *The set of all input enabled languages (trace sets) of a given signature forms a complete lattice, ordered by inclusion of trace set, with set intersection being the greatest lower bound operator, and set union being the least upper bound operator.*

The above lattice induces a quotient lattice on I/O automata of a given signature. Computing a product of two automata gives an automaton that belongs to the greatest lower bound class, while taking a sum of two automata gives an automaton that belongs to the least upper bound class.

Proof of Theorem 12. We can view the process of solving (4) compositionally: each of the two equations can be solved separately and then the environment being the greatest lower bound of the two solutions is a solution to the entire system of equations. So it suffices to prove that each of the equations separately has a greatest solution. This argument boils down to showing that for any two solutions E' and E'' of given equation, an automaton E from their greatest lower bound class is also a solution (E will be such that $Tr_E = Tr_{E'} \cup Tr_{E''}$).

Fortunately this is actually the case as "$|$" distributes over union, which can be easily shown using lemma 28. So for the first equation we get: $Tr_{S_1} \cap Tr_{E'|S_2} \subseteq Tr_{E_1}$ and $Tr_{S_1} \cap Tr_{E''|S_2} \subseteq Tr_{E_1}$ and $Tr_{E|S_1} = Tr_{E'|S_1} \cup Tr_{E''|S_1}$ implies that $Tr_{S_1} \cap Tr_{E|S_2} \subseteq Tr_{E_1}$. Similarly for the second equation. □

Proof. (Proof of Theorem 13) It is fairly easy to see that if

$$(E, S) = ((E_1, S_1) \,|\, (E_2, S_2)) \,|\, (E_3, S_3) \ ,$$

then $S \leqslant S_1|S_2|S_3$ and $S_1|S_2|S_3 \leqslant S$, and similarly for the dual parenthesizing of this composition (follows from associativity of parallel composition of I/O automata). So if we combine three interfaces, we are guaranteed that they have equivalent specification component. We should now argue that they also have equivalent environment components.

Observe that by definition of composition E must fulfill the following inequalities (together with some maximal E_{12} that has to exist):

$$S_1|S_2 \models E|S_3 \leqslant E_{12} \tag{9}$$
$$S_3 \models E|S_1|S_2 \leqslant E_3 \tag{10}$$
$$S_2 \models E_{12}|S_1 \leqslant E_2 \tag{11}$$
$$S_1 \models E_{12}|S_2 \leqslant E_1 \tag{12}$$

By Lemma 36 then E must be a maximal solution of

$$S_1 \models E|S_2|S_3 \leqslant E_1 \tag{13}$$
$$S_2 \models E|S_1|S_3 \leqslant E_2 \tag{14}$$
$$S_3 \models E|S_1|S_2 \leqslant E_3 \tag{15}$$

Similarly by Lemma 37 any maximal solution of inequalities (13)–(15) is also a maximal solution of inequalities (9)–(12) for some existing E_{12}. All this means that the two characterizations are equivalent. Similarly we can reduce the alternative parenthesizing of the composition of three interfaces to the same characterization (13)–(15), meaning that both parenthesizing yield equivalent environment components E, which finishes the proof. □

Lemma 36. *Any E satisfying (9)–(12) for some choice of E_{12}, also satisfies inequalities (13)–(15).*

Proof. So assume that E satisfying (9)–(12). Observe that by (12), (9), Lemma 30, and commutativity of "|", we get (13).

$$\frac{S_1 \models E_{12}|S_2 \leqslant E_1 \qquad S_1|S_2 \models E|S_3 \leqslant E_{12}}{S_1 \models E|S_2|S_3 \leqslant E_1} \tag{16}$$

Similarly by (11), (9) and Lemma 30, we get (14).

$$\frac{S_2 \models E_{12}|S_1 \leqslant E_2 \qquad S_1|S_2 \models E|S_3 \leqslant E_{12}}{S_2 \models E|S_1|S_3 \leqslant E_2} \tag{17}$$

Finally (15) is directly contain in our assumptions as (10). □

Lemma 37. *If E satisfies (13)–(15) then there exists E_{12} such that both satisfy (9)–(11).*

Proof. Assume that E satisfies (13)–(15). Observe that (13) and (14) mean that $E|S_3$ satisfies mutual deadlock freeness requirements (4) for composition of (E_1, S_1) and (E_2, S_2):

$$S_1 \models (E|S_3)|S_2 \leqslant E_1 \tag{18}$$
$$S_2 \models (E|S_3)|S_1 \leqslant E_2 \tag{19}$$

But this means, by Thm. 12 that there exist a maximal such E_{12} satisfying these equations—we conclude that inequalities (11) and (12) are satisfied. Further observe that due to maximality of E_{12} we get $E|S_3 \leqslant E_{12}$ which implies in particular that:

$$S_1|S_2 \models E|S_3 \leqslant E_{12} \ , \tag{20}$$

so (9) is satisfied. Finally (10) is directly contained in our assumptions as (15). □

Proof. (Proof of Theorem 14) We prove the contrapositive of theorem's claim:

$$\begin{aligned}(E'', S'') \text{ incompatible with } (E, S_1|S_2) \text{ implies} \\ (E'', S'') \text{ incompatible with } (E', S')\end{aligned} \tag{21}$$

Let M be a mute environment (not producing any outputs) of the same signature as $E''|E$. Then (E'', S'') incompatible with $(E, S_1|S_2)$ means either (a) $S_1|S_2 \not\models M|S'' \leqslant E$, or (b) $S'' \not\models M|S_1|S_2 \leqslant E''$, because if the mute environment M cannot satisfy the mutual deadlock freeness then certainly any bigger environment cannot.

(a) There exists a trace $\sigma \cdot a$ such that $\sigma \cdot a \in Tr_{M|S''} \cap Tr_{S_1|S_2}$ and $\sigma \cdot a \notin Tr_E$, while $\sigma \in Tr_E$. Take $\sigma_{E'} \cdot b$ a prefix of $\sigma \cdot a$ such that $\sigma_{E'} \in Tr_{E'}$ and $\sigma_{E'} \cdot b \notin Tr_{E'}$. Such a prefix always exists as $Tr_{E'} \subset Tr_E$ — both E and E' are solutions of the same mutual deadlock freeness inequalities, and E is maximal, so. Since (E', S') satisfies independent implementability, and S_1, S_2 are legal implementations themselves we get that $E' \models S_1|S_2 \leqslant S'$. Since $\sigma_{E'} \in Tr_{E'} \cap Tr_{S_1|S_2}$ then $\sigma_{E'} \in Tr_{S'}$. As $b \in in_{S'}$ then due to input-enabledness $\sigma_{E'} \cdot b \in Tr_{S'}$. Summing up $\sigma_{E'} \cdot b \in Tr_{S'} \cap Tr_{M|S''}$ and $\sigma_{E'} \cdot b \notin Tr_{E'}$, which means that $S' \not\models M|S'' \leqslant E'$ and consequently (E'', S'') incompatible with (E', S').

(b) From assumption there exists a trace σ such that $\sigma \in Tr_{M|S_1|S_2} \cap Tr_{S''}$, $\sigma \notin Tr_{E''}$. By Lemma 29 there exists σ' such that $\sigma' \restriction ext_{S_1|S_2} \in Tr_{S_1|S_2} \cap Tr_{M|S''}$. If $\sigma' \restriction ext_{S_1|S_2} \in Tr_{E'}$ then by $E' \models S_1|S_2 \leqslant S'$ we get

$\sigma' \restriction ext_{S_1|S_2} \in Tr_{S'}$, and consequently by lemma 28 $\sigma \in Tr_{M|S'} \cap Tr_{S''}$ and $\sigma \notin Tr_{E''}$, so $S'' \not\models M|S' \leqslant E''$, effectively implying incompatibility of (E', S') and (E'', S'').

What if $\sigma' \restriction ext_{S_1|S_2} \notin Tr_{E'}$? Then consider a prefix $\sigma_{E'} \cdot b$ of $\sigma' \restriction ext_{S_1|S_2}$ such that $\sigma_{E'} \in Tr_{E'}$ and $\sigma_{E'} \cdot b \notin Tr_{E'}$. We get that $\sigma_{E'} \cdot b \in Tr_{M|S''}$ as a prefix, $\sigma_{E'} \cdot b \in Tr_{S'}$ by $E' \models S_1|S_2 \leqslant S'$ and b input of S', and $\sigma_{E'} \cdot b \notin Tr_{E'}$. This effectively means that $S' \not\models M|S'' \leqslant E'$, so (E', S') and (E'', S'') are incompatible. □

Proof of Theorem 15. By independent implementability $Tr_{E'} \cap Tr_{S_1|S_2} \subseteq Tr_{E'} \cap Tr_{S'}$, which implies $Tr_{E'} \setminus Tr_{S'} \subseteq Tr_{E'} \setminus Tr_{S_1|S_2} \subseteq Tr_E \setminus Tr_{S_1|S_2}$, the second inclusion by maximality of E: $Tr_{E'} \subset Tr_E$. By the third case of Theorem 6 we get $(E, S_1|S_2) \preceq (E', S')$. □

Proof of Theorem 17. This proof is a simple generalization of the argument in the proof of Theorem 12. □

We are now interested in proving Theorem 21 arguing the correctness of our method for solving systems of inequalities. Let us start with an auxiliary well-foundedness lemma.

Lemma 38. *Let $\mathcal{C}(E)$ be a system of inequalities*

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

and $\mathcal{E} = (Q, A, \rightarrow_{may}, \rightarrow_{must})$ be a modal transition system induced by $\mathcal{C}$. If E is an I/O automaton such that its set of states $states_E$ is a maximal consistency relation over $\mathcal{E}$ and

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

and its set of transitions $step_E$ is a maximal consistent set of transitions of $\mathcal{E}$ with respect to $states_E$ and the maximal state consistency relation of $\mathcal{E}$ is not empty, then

1. *The signature of E is the same as a signature of solutions of $\mathcal{C}(E)$*

2. *E is input enabled.*

Proof. Ad (i). By simple inspection of the rules and Definition 19 one can convince herself that that actions in the modal transition systems are only taken from in_E and out_E. Ad (ii) Note that the rules for generating transitions are input enabled. If an input labeled transition is missing it must be because it was removed. However all input transitions are *must* transitions, so if it was removed also its source state should be removed, which contradicts with existence of a non-input enabled state. □

Sketch of Proof of Theorem 21. The correctness of algorithm is largerly by construction: assume a solution found using this method does not satisfy one of the inequalities and get a witnessing traces. Find a contradiction, as any witnessing trace leads to a conclusion that an inconsistent state or a transition is in the solution.

Maximality follows from choice of the maximal consistency relation. □

Below we present a pseudocode SOLVE of a natural (non-optimized) algorithm implementing the method. We start with the set of states Q equal to the initial set of inequalities. After exhaustively adding alls states reachable by must transitions, we remove those that are reachable backwards from error states. Once this is done we explore possible may transitions. The three parts of the algorithm are iterated until no more states can be added. The algorithm relies on three functions δ (computing a closure of a set of constraints), ϕ (computing a must step), and ψ (computing a may step). The functions, also presented below, are simple reformulations of the rules presented above.

$$\begin{aligned}\delta(E) = \{(P', S', F') \mid (P,S,F) \in E, a \in ext_S \cap ext_F,\\ S \xrightarrow{a} S', F \xrightarrow{a} F', P \xrightarrow{a} P'\}\\ \cup \{\textsc{False} \mid (P,S,F) \in E, a \in ext_S \cap ext_F,\\ S \xrightarrow{a?} S', F' \xrightarrow{a!}\!\!\!\!\!\!\rightarrow, P \xrightarrow{a?} P'\}\end{aligned}$$

$$\begin{aligned}\phi_a(E) = \\ \{(P', S, F') \mid (P,S,F) \in E, a \in in_E \setminus out_S, F \xrightarrow{a?} F', P \xrightarrow{a!} P'\}\\ \cup \{(P, S', F) \mid (P,S,F) \in E, a \in int_E \cap out_S, S \xrightarrow{a!} S'\}\end{aligned}$$

$$\phi(Q) = \{(E, a, \delta(\phi_a(E))) \mid E \in Q, a \in in_E\}$$

$$
\begin{aligned}
\psi_a(E) = & \\
& \{(P', S, F') \mid (P, S, F) \in E, a \in out_E \setminus in_S, P \xrightarrow{a?} P', F \xrightarrow{a!} F'\} \\
& \cup \{\textsc{False} \mid a \in out_E \setminus in_S, P \xrightarrow{a?} P', F \xrightarrow{a!} \not\rightarrow\} \\
& \cup \{(P, S', F) \mid (P, S, F) \in E, a \in out_E \cap in_S, S \xrightarrow{a?} S'\} \\
\psi(Q) = & \{(E, a, \delta(\psi_a(E)) \mid E \in Q, a \in out_E\}
\end{aligned}
$$

$\textsc{Solve}(\{(P_1, S_1, F_1), \ldots, (P_m, S_m, F_m)\}\})$

```
                    ▷ explored states
1   Q ← {{(P_1, S_1, F_1), ..., (P_m, S_m, F_m)}}
2   X ← ∅           ▷ explored error states
3   T ← ∅           ▷ explored transitions
4   do      Q' ← Q

            ▷ Take May transitions
            ▷ perform steps forwards (fixpoint)
5           do      T' ← T
6                   T ← T ∪ φ(Q)
7                   Q ← Q ∪ {E' | (_, _, E') ∈ T} \ X
8           while   T ≠ T'

             ▷ prune all error states backwards (fixpoint)
 9           X ← X ∪ {E ∈ Q | False ∈ E}
10           do      X' ← X
11                   X ← X ∪ {E | (E, a, E') ∈ T, E' ∈ X, a ∈ in_E}
12           while X ≠ X'
13           Q ← Q \ X

             ▷ Take May transitions (fixpoint)
14           do      T' ← T
15                   T ← T ∪ ψ(Q)
16                   Q ← Q ∪ {(_, _, E') | False ∉ E'}
17           while   T' ≠ T

18  while   Q ≠ Q'
19  T ← {(E, a, E') ∈ T | E ∈ Q, E' ∈ Q}
20  return Q,T
```

In the following we will use Det_A to denote a determinized version of interface automaton A. Determinization of interface automata can be achieved by using the classical algorithm for determinizing NFAs.

In the following we use $Prune$ to mean the process of removing incompatible states from two composed interface automata. It is defined exactly as the $||$ composition operator in [2], such that the result is empty exactly when the two interface automata are incompatible.

Lemma 39. *If two interface automata A_1, A_2 are compatible, then their determinizations Det_{A_1}, Det_{A_2} are compatible.*

Proof of Lemma 39. The lemma can be rewritten as:

$$Prune_{(A_1 \otimes A_2)} \text{ is non-empty } \implies Prune_{(Det_{A_1} \otimes Det_{A_2})} \text{ is non-empty}$$

We will prove the contrapositive of this.

$$Prune_{(Det_{A_1} \otimes Det_{A_2})} \text{ is empty } \implies Prune_{(A_1 \otimes A_2)} \text{ is empty}$$

If $Prune_{(Det_{A_1} \otimes Det_{A_2})}$ is empty then it means that there exists a trace $\sigma \in Tr_{Det_{A_1} \otimes Det_{A_2}}$ consisting of internally controllably transitions $(out_{A_1 \otimes A_2} \cup int_{A_1 \otimes A_2})$ such that after executing it from the initial state of $Det_{A_1} \otimes Det_{A_2}$ a state (s_1, s_2) is reached, which is illegal. We assume without loss of generality that Det_{A_1} in s_1 produces an output a and that Det_{A_2} in s_2 is unable to receive a and thus deadlocks. We know that there exists a trace σ' such that $\sigma = \sigma' \restriction ext_{A_1 \otimes A_2}$, $\sigma_1 = \sigma' \restriction ext_{A_1} \in Tr_{Det_{A_1}}$ and $\sigma_2 = \sigma' \restriction ext_{A_2} \in Tr_{Det_{A_2}}$. The traces σ_1 and σ_2 are also traces of A_1 and A_2 respectively, but in A_1 and A_2 they may lead to several state due to non-determinism. Because determinization can increase the possible transitions from a state, and never decrease them, there will be at least one state reachable by σ_1 in A_1 in which it can produce a and at least one state reachable by σ_2 in A_2 in which it deadlocks on the input a. This pair of states, which together form an illegal state in $A_1 \otimes A_2$ is reachable from the initial state of $A_1 \otimes A_2$ through internally controllable transitions $(out_{A_1 \otimes A_2} \cup int_{A_1 \otimes A_2})$. □

Lemma 40. *If A is deterministic and $unzip_{(A)} = (E, S)$ then*

$$Tr_E \cap Tr_S = Tr_A \tag{22}$$

$$\textit{and} \quad \sigma \in Tr_E \setminus Tr_S \implies \sigma \notin Tr_A \tag{23}$$

$$\textit{and} \quad \sigma \in Tr_S \setminus Tr_E \implies \sigma \notin Tr_A \tag{24}$$

$$\textit{and} \quad Tr_E \supseteq Tr_A \tag{25}$$

$$\textit{and} \quad Tr_S \supseteq Tr_A \tag{26}$$

Proof of Lemma 40. In order to prove (22) we look at the transition relations of E and S as defined on page 17. These transition relations are extended versions of the transition relation of A. The two transition relations are extend on two non-overlapping sets, namely in_E and in_S. Thus no new traces are added to the set $Tr_E \cap Tr_S$ because a new trace of either Tr_E or Tr_S will have to include a symbol on which the other transition relation does not differ from the transition relation of A. All the other points follow easily from (22). □

Proof of Theorem 24. The theorem can be rewritten as:

$$\begin{aligned} &Prune_{(A_1 \otimes A_2)} \text{ is non-empty } \Longrightarrow \\ &\quad \text{if } (E_1, S_1) = unzip_{(A_1)} \wedge (E_2, S_2) = unzip_{(A_2)} \\ &\quad \text{then } \exists E \text{ such that } S_1 \models E|S_2 \leqslant E_1 \wedge S_2 \models E|S_1 \leqslant E_2 \end{aligned}$$

We choose E to be the mute environment. We now claim that E satisfies $S_1 \models E|S_2 \leqslant E_1 \wedge S_2 \models E|S_1 \leqslant E_2$.

This is proven by contradiction, assuming that the conclusion does not hold. Based on symmetry, we will only look at one case of negation of our final conclusion: $S_1 \not\models E|S_2 \leqslant E_1$. We will prove that if $S_1 \not\models E|S_2 \leqslant E_1$ then $Prune_{(A_1 \otimes A_2)}$ will have to be empty. Given lemma 39 we can assume that A_1 and A_2 are deterministic.

Take the shortest trace σa witnessing that $S_1 \not\models E|S_2 \leqslant E_1$. This gives that $\sigma a \in S_1$, $\sigma a \in E|S_2$ and $\sigma a \notin E_1$. We also have that $\sigma \in S_1$, $\sigma \in E|S_2$ and $\sigma \in E_1$. Also $a \in out_{E_1}$ since E_1 is input enabled. Given that A_1 is deterministic we can conclude by lemma 40 that $Tr_{E_1} \cap Tr_{S_1} = Tr_{A_1}$ and thus we know that $\sigma \in Tr_{A_1}$. We can also conclude that $\sigma a \notin Tr_{A_1}$ since $\sigma a \notin Tr_{E_1}$.

Since we have that $\sigma a \in E|S_2$ we know that there must exist a trace $\sigma' \in Tr_{S_2}$ such that $\sigma = \sigma' \restriction ext_{S_1}$. The only extra symbols that might be in σ' are outputs from S_2 not directed towards S_1. This can be concluded because S_2 is composed with the mute environment E. Thus σ' consists of symbols that are either inputs from S_1 to S_2, internal transitions of S_2 or outputs of S_2 to S_1. All of these symbols are internally controllable actions of $A_1 \otimes A_2$. Because we have that $\sigma \in A_1$ and that the extra symbols in σ' are inputs not coming from A_1 we can conclude that $\sigma' \in A_2$. Given this we have that σ' will lead $A_1 \otimes A_2$ via internally controllable actions to a state in which A_2 is ready to output a but in which A_1 is blocking on a as an input.

Thus we have proved that $Prune_{(A_1 \otimes A_2)}$ must be empty because an illegal state can be reached from the initial state by internally controllable

transitions. □

Proof of Theorem 25. Prove a simplified contrapositive instead: if the initial state of $zip_{(E_1,S_1)} \times zip_{(E_2,S_2)}$ contains illegal states then $S_2 \not\models M|S_1 \leqslant E_2$ or $S_1 \not\models M|S_2 \leqslant E_1$. If the mute automaton M does not satisfy the equations, then it must be that no other E can solve them, and the two interfaces are incompatible in our framework.

Take $A_1 = zip_{(E_1,S_1)}$, $A_2 = zip_{(E_2,S_2)}$ and observe that $(s_1^0, s_2^0) = start_{A_1 \times A_2}$ is incompatible. This means that there exists a trace $\sigma \in (int_{A_1|A_2} \cup out_{A_1|A_2})^*$ such that $(s_1^0, s_2^0) \xrightarrow{\sigma} (d_1, d_2)$ and (d_1, d_2) is an incompatible state. So there exist σ_1, σ_2 such that $s_1^0 \xrightarrow{\sigma_1} d_1$ and $s_2^0 \xrightarrow{\sigma_2} d_2$. Also $\sigma_1 = \sigma \restriction ext_{A_1}$ and $\sigma_2 = \sigma \restriction ext_{A_2}$. Since (d_1, d_2) is an error state there must exist an action $a \in int_{A_1|A_2}$ such that either $d_1 \xrightarrow{a!}$ and $d_2 \xrightarrow{a?} \not\rightarrow$ or $d_2 \xrightarrow{a!}$ and $d_1 \xrightarrow{a?} \not\rightarrow$. In the first case (the second case is entirely symmetric) we get: $\sigma_1 a \in Tr_{S_1} \cap Tr_{E_1}$ and $\sigma_2 \in Tr_{S_2} \cap Tr_{E_2}$. Due to input-enabledness $\sigma_2 a \in Tr_{S_2}$ so it must be that $\sigma_2 a \notin Tr_{E_2}$ (as E_2 is deterministic). We will argue that $\sigma_2 a$ witnesses the following:

$$S_2 \not\models M|S_1 \leqslant E_2$$

We already know that $\sigma_2 a \in Tr_{S_2}$ and $\sigma_2 a \notin Tr_{E_2}$ it remains to argue that $\sigma_2 a \in Tr_{M|S_1}$, which we will argue by using σa as a witness:

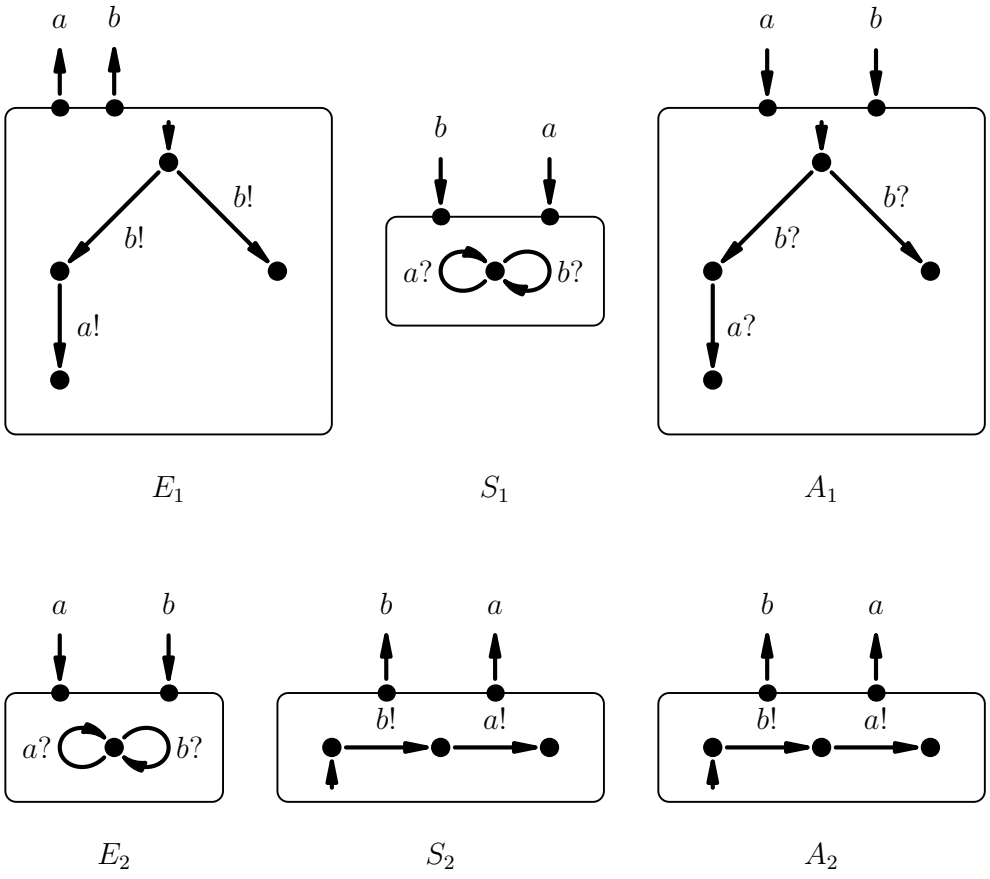
1. First $\sigma a \restriction ext_M \in Tr_M$ because any symbol in σa is from $out_{S_1|S_2} \cup int_{S_1|S_2}$. The latter of the two sets is disjoint from ext_M, so all its elements are filtered out from $\sigma a \restriction ext_M$. The former set is equal to in_M. So all actions of $\sigma a \restriction ext_M$ are inputs of M so necessarily $\sigma a \restriction ext_M$ is a trace of M due to input enabledness.

2. Then $\sigma a \restriction ext_{S_1} \in Tr_{S_1}$, as $\sigma a \restriction ext_{S_1} = \sigma_1 a \in Tr_{S_1}$.

3. Finally $\sigma a \in (ext_M \cup ext_{S_1})$, as this is a set that includes all actions that are in the configuration.

□

Theorem 25 only holds for deterministic systems. A counter example consists of two interfaces that are compatible, but their zippings give raise to incompatible interface automata.



Above E_1, S_1, E_2, S_2 are I/O automata, while A_1, A_2 are interface automata. Also we have that $zip_{(E_1,S_1)} = A_1$ and $zip_{(E_2,S_2)} = A_2$. It is clear that A_1 and A_2 are incompatible. The error state is reached if A_1 non-deterministically chooses the right branch.

We want to argue that (E_1, S_1) is compatible with (E_2, S_2). Since these two components constitute a closed system, any environment E for them (if it existed) would have an empty transition relation and Tr_E contains only the empty trace. Let us argue that such E actually satisfies the requirements:

1. Since $Tr_E = \{\epsilon\}$ we have that $Tr_{E|S_1} = Tr_{S_1}$. Then $Tr_{S_2} \cap Tr_{S_1} = Tr_{S_2} \subseteq Tr_{E_2}$.

2. Similarly $Tr_{E|S_2} = Tr_{S_2}$, $Tr_{S_1} \cap Tr_{S_2} = Tr_{S_2} \subseteq Tr_{E_1}$.

So the single state empty environment E is a legal environment for $(E_1, S_1)|(E_2, S_2)$, meaning that the two are compatible and the counterexample is valid.

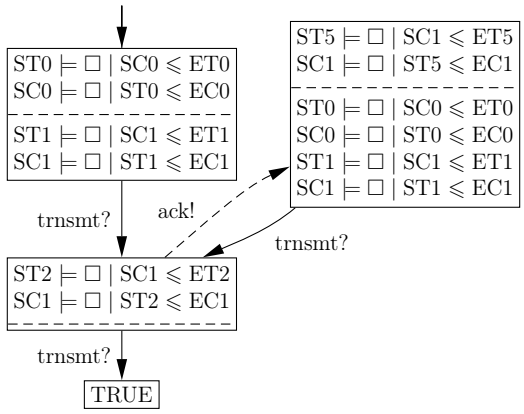


Figure 7: Resulting modal transition system for *Comp2*.

Recent BRICS Report Series Publications

RS-06-11 **Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski.** ***An Interface Theory for Input/Output Automata.*** **June 2006. 40 pp. Appears in Misra, Nipkow and Sekerinski, editors,** ***Formal Methods: 14th International Symposium,*** **FM '06 Proceedings, LNCS 4085, 2006, pages 82–97.**

RS-06-10 **Christian Kirkegaard and Anders Møller.** ***Static Analysis for Java Servlets and JSP.*** **June 2006. 23 pp. Full version of paper presented at SAS '06.**

RS-06-9 **Claus Brabrand, Robert Giegerich, and Anders Møller.** ***Analyzing Ambiguity of Context-Free Grammars.*** **April 2006. 19 pp.**

RS-06-8 **Christian Kirkegaard and Anders Møller.** ***Static Analysis for Java Servlets and JSP.*** **April 2006. 22 pp.**

RS-06-7 **Petr Jančar and Jiří Srba.** ***Undecidability Results for Bisimilarity on Prefix Rewrite Systems.*** **April 2006. 20 pp. Presented at** ***FoSSaCS 2006,*** **LNCS 3921:277–291.**

RS-06-6 **Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Bas Luttik.** ***A Finite Equational Base for CCS with Left Merge and Communication Merge.*** **March 2006. 22 pp.**

RS-06-5 **Kristian Støvring.** ***Extending the Extensional Lambda Calculus with Surjective Pairing is Conservative.*** **March 2006. 18 pp. To appear in** ***Logical Methods in Computer Science.*** **Supersedes RS-05-35.**

RS-06-4 **Olivier Danvy and Kevin Millikin.** ***A Rational Deconstruction of Landin's J Operator.*** **February 2006. ii+26 pp. To appear in the post-reviewed proceedings of the 17th International Workshop on the** ***Implementation and Application of Functional Languages*** **(IFL'05), Dublin, Ireland, September 2005.**

RS-06-3 **Małgorzata Biernacka and Olivier Danvy.** ***A Concrete Framework for Environment Machines.*** **February 2006. ii+29 pp. To appear in the** ***ACM Transactions on Computational Logic.*** **Supersedes BRICS RS-05-15.**

RS-06-2 **Mikkel Baun Kjærgaard and Jonathan Bunde-Pedersen.** ***A Formal Model for Context-Awareness.*** **February 2006. 26 pp.**

RS-06-1 **Luca Aceto, Taolue Chen, Willem Jan Fokkink, and Anna Ingólfsdóttir.** ***On the Axiomatizability of Priority.*** **January 2006. 25 pp.**