Interface-based Design*

Luca de Alfaro[1] and Thomas A. Henzinger[2,3]

[1]UC Santa Cruz [2]UC Berkeley [3]EPFL Switzerland

Abstract. Surveying results from [5] and [6], we motivate and introduce the theory behind formalizing rich interfaces for software and hardware components. Rich interfaces specify the protocol aspects of component interaction. Their formalization, called *interface automata*, permits a compiler to check the compatibility of component interaction protocols. Interface automata support incremental design and independent implementability. Incremental design means that the compatibility checking of interfaces can proceed for partial system descriptions, without knowing the interfaces of all components. Independent implementability means that compatible interfaces can be refined separately, while still maintaining compatibility.

1 Introduction

Interfaces play a central role in the component-based design of software and hardware systems. We say that two or more components are *compatible* if they work together properly. Good interface design is based on two principles. First, an interface should expose enough information about a component as to make it possible to predict if two or more components are compatible by looking only at their interfaces. Second, an interface should not expose more information about a component than is required by the first principle.

The technical realization of these principles depends, of course, on what it means for two or more components to "work together properly." A simple interpretation is offered by typed programming languages: a component that implements a function and a component that calls the function are compatible if the function definition and the function call agree on the number, order, and types of the parameters. We discuss richer notions of compatibility, which specify in addition to type information, also protocol information about how a component must be used. For example, the interface of a file server with the two methods `open-file` and `read-file` may stipulate that the method `read-file` must not be called before the method `open-file` has been called. Symmetrically, the interface of a client specifies the possible behaviors of the client in terms of which orderings of `open-file` and `read-file` calls may occur during its execution.

* This research was supported in part by the ONR grant N00014-02-1-0671 and by the NSF grants CCR-0234690, CCR-9988172, and CCR-0225610. A version of this paper appeared in *Engineering Theories of Software-Intensive Systems* (edited by Manfred Broy et al.), NATO Science Series, Volume 195, Springer-Verlag, 2005.

Given such server and client interfaces, a compiler can check statically if the server and the client fit together.

Interfaces that expose protocol information about component interaction can be specified naturally in an automaton-based language [5]. In this article, we give a tutorial introduction to such *interface automata*.

2 Interface Languages

We begin by introducing two requirements on interface languages. An interface language should support incremental design and independent implementability. With each interface language we present, we will verify that both of these requirements are met.

2.1 Incremental design

A component is typically an open system, i.e., it has some free inputs, which are provided by other components. Incremental design is supported if we can check the compatibility of two or more component interfaces without specifying interfaces for *all* components, i.e., without closing the system. The unspecified component interfaces may later be added one by one, as long as throughout the process, the set of specified interfaces stays compatible. More precisely, the property of incremental design requires that if the interfaces in a set $\mathcal{F}$ (representing the complete, closed design) are compatible, then the interfaces in every subset $\mathcal{G} \subseteq \mathcal{F}$ (representing a partial, open design) are compatible. This yields an *existential* interpretation of interface compatibility: the interfaces in an open set $\mathcal{G}$ of interfaces (i.e., a set with free inputs) are compatible if there exists an interface E (representing an environment that provides all free inputs to the interfaces in $\mathcal{G}$) such that the interfaces in the closed set $\mathcal{G} \cup \{E\}$ (without free inputs) are compatible.[1]

Incremental design suggests that we model *compatibility* as a symmetric binary relation $\sim$ between interfaces, and *composition* as a binary partial function $||$ on interfaces. If two interfaces F and G are compatible, that is, $F \sim G$, then $F||G$ is defined and denotes the resulting composite interface. Now the property of *incremental design* reads as follows:

> *For all interfaces F, G, H, and I, if $F \sim G$ and $H \sim I$ and $F||G \sim H||I$, then $F \sim H$ and $G \sim I$ and $F||H \sim G||I$.*

This property ensures that the compatible components of a system can be put together in any order.[2]

[1] It is important to emphasize the existential interpretation of the free inputs of interfaces, because this deviates from the standard, universal interpretation of free inputs in specifications [5]. While a specification of an open system is *well-formed* if it can be realized for all input values, an interface is well-formed if it is compatible to some environment. In other words, for interfaces, the environment is helpful, not adversarial.

[2] We could formalize the property of incremental design, instead, as associativity of interface composition [5]. We chose our formalization, because it does not require an

2.2 Independent implementability

Recall the first principle of interface design, namely, that the information contained in interfaces should suffice to check if two or more components are compatible. This principle can be formalized as follows: if F and G are compatible interfaces, and F' is a component that conforms to interface F, and G' is a component that conforms to interface G, then F' and G' are compatible components, and moreover, the composition $F'||G'$ of the two components conforms to the composite interface $F||G$. We call this the property of independent implementability, because it enables the outsourcing of the implementation of the components F' and G' to two different vendors: as long as the vendors conform to the provided interfaces F and G, respectively, their products will fit together, even if the vendors do not communicate with each other.

For simplicity, in this article we gloss over the differences between interfaces and components, and express both in the same language; that is, we consider components to be simply more detailed interfaces.[3] For this purpose, we use a *refinement* preorder between interfaces: if $F \succeq F'$, then the interface F' refines the interface F. An interface may be refined into an implementation in several steps. As the refinement relation is a preorder, it is transitive. The property of *independent implementability* reads as follows:

> *For all interfaces F, F', G, and G', if $F \sim G$ and $F \succeq F'$ and $G \succeq G'$, then $F' \sim G'$ and $F||G \succeq F'||G'$.*

This property ensures that compatible interfaces can always be refined separately.[4]

3 Assume/Guarantee Interfaces

We illustrate the properties of incremental design and independent implementability through a simple, stateless interface language called assume/guarantee (A/G, for short) interfaces [6]. Assume/guarantee interfaces have input and output variables. An A/G interface puts a constraint on the environment through a predicate ϕ^I on its input variables: the environment is expected to provide inputs that satisfy ϕ^I. In return, the interface communicates to the environment

explicit notion of equality or equivalence between interfaces. Implicitly, according to our formalization, two interfaces F and G are *equivalent* if they are compatible with same interfaces, that is, if for all interfaces H, we have $F \sim H$ iff $G \sim H$. It can be shown that if the property of incremental design holds, then for all interfaces F, G, and H, if $F \sim G$ and $F||G \sim H$, then $G \sim H$ and $F \sim G||H$ and the two interfaces $(F||G)||H$ and $F||(G||H)$ are equivalent in the specified sense.

[3] A discussion about interfaces versus components can be found in [6].

[4] The property of independent implementability is a compositionality property [6]. It should be noted that the "direction" of interface compositionality is top-down, from more abstract to more refined interfaces: if $F \sim G$ and $F \succeq F'$ and $G \succeq G'$, then $F' \sim G'$. This is in contrast to the bottom-up compositionality of many other formalisms: if $F' \sim G'$ and $F' \preceq F$ and $G' \preceq G$, then $F \sim G$.

a constraint ϕ^O on its output variables: it vouches to provide only outputs that satisfy ϕ^O. In other words, the input assumption ϕ^I represents a precondition, and the output guarantee ϕ^O a postcondition.

Definition 1. *An* A/G interface $F = \langle X^I, X^O, \phi^I, \phi^O \rangle$ *consists of*

- *two disjoint sets* X^I *and* X^O *of* input *and* output *variables;*
- *a satisfiable predicate* ϕ^I *over* X^I *called* input assumption*;*
- *a satisfiable predicate* ϕ^O *over* X^O *called* output guarantee*.*

Note that input assumptions, like output guarantees, are required to be satisfiable, not valid. An input assumption is satisfiable if it can be met by some environment. Hence, for every A/G interface there is a context in which it can be used. On the other hand, in general not all environments will satisfy the input assumption; that is, the interface puts a constraint on the environment.

Example 1. A division component with two inputs x and y, and an output z, might have an A/G interface with the input assumption $y \neq 0$ and the output guarantee TRUE (which is trivially satisfied by all output values). The input assumption $y \neq 0$ ensures that the component is used only in contexts that provide non-zero divisors. □

In the following, when referring to the components of several interfaces, we use the interface name as subscript to identify ownership. For example, the input assumption of an interface F is denoted by ϕ^I_F.

3.1 Compatibility and composition

We define the composition of A/G interfaces in several steps. First, two A/G interfaces are syntactically composable if their output variables are disjoint. In general, some outputs of one interface will provide inputs to the other interface, and some inputs will remain free in the composition. Second, two A/G interfaces F and G are semantically compatible if whenever one interface provides inputs to the other interface, then the output guarantee of the former implies the input assumption of the latter. Consider first the closed case, that all inputs of F are outputs of G, and vice versa. Then F and G are compatible if the closed formula

$$(\forall X^O_F \cup X^O_G)(\phi^O_F \wedge \phi^O_G \;\Rightarrow\; \phi^I_F \wedge \phi^I_G) \qquad (\psi)$$

is true. In the open case, where some inputs of F and G are left free, the formula (ψ) has free input variables. As discussed above, to support incremental design, the two interfaces F and G are compatible if they can be used together in *some* context, i.e., if there is a environment that makes (ψ) true by providing helpful input values. Thus, in the open case, the A/G interfaces F and G are compatible if the formula (ψ) is satisfiable. Then, the formula (ψ) is the input assumption of the composite interface $F||G$, because it encodes the weakest condition on the environment of $F||G$ that makes F and G work together.

Definition 2. *Two A/G interfaces F and G are* composable *if $X_F^O \cap X_G^O = \emptyset$. Two A/G interfaces F and G are* compatible*, written $F \sim G$, if they are composable and the formula*

$$(\forall X_F^O \cup X_G^O)(\phi_F^O \wedge \phi_G^O \;\Rightarrow\; \phi_F^I \wedge \phi_G^I) \qquad (\psi)$$

is satisfiable. The composition $F||G$ of two compatible A/G interfaces F and G is the A/G interface with

- $X_{F||G}^I = (X_F^I \cup X_G^I) \backslash X_{F||G}^O$;
- $X_{F||G}^O = X_F^O \cup X_G^O$;
- $\phi_{F||G}^I = \psi$;
- $\phi_{F||G}^O = \phi_F^O \wedge \phi_G^O$.

Note that the compatibility relation $\sim$ is symmetric.

Example 2. Let F be the A/G interface without input variables, the single output variable x, and the output guarantee TRUE. Let G be the A/G interface with the two input variables x and y, the input assumption $x = 0 \;\Rightarrow\; y = 0$, and no output variables. Then F and G are compatible, because the formula

$$(\forall x)(\text{TRUE} \;\Rightarrow\; (x = 0 \;\Rightarrow\; y = 0))$$

simplifies to $y = 0$, which is satisfiable. Note that the predicate $y = 0$ expresses the weakest input assumption that the composite interface needs to make in order to ensure that the input assumption $x = 0 \;\Rightarrow\; y = 0$ of G is satisfied. This is because F makes no guarantees about x; in particular, it might provide outputs x that are 0, and it might provide outputs x that are different from 0.

The composition $F||G$ has the input variable y, the input assumption $y = 0$, the output variable x, and the output guarantee TRUE. □

The following theorem shows that the A/G interfaces support incremental design.

Theorem 1. *For all A/G interfaces F, G, H, and I, if $F \sim G$ and $H \sim I$ and $F||G \;\sim\; H||I$, then $F \sim H$ and $G \sim I$ and $F||H \;\sim\; G||I$.*

Proof sketch. Note that from the premises of the theorem it follows that (1) the four sets X_F^O, X_G^O, X_H^O, and X_I^O are pairwise disjoint; and (2) the formula

$$(\forall X_F^O \cup X_G^O \cup X_H^O \cup X_I^O)(\phi_F^O \wedge \phi_G^O \wedge \phi_H^O \wedge \phi_I^O \;\Rightarrow\; \phi_F^I \wedge \phi_G^I \wedge \phi_H^I \wedge \phi_I^I)$$

is satisfiable. To prove from this that, say, the formula

$$(\forall X_F^O \cup X_H^O)(\phi_F^O \wedge \phi_H^O \;\Rightarrow\; \phi_F^I \wedge \phi_H^I)$$

is satisfiable, choose the values for the variables in $X_{F||H}^I \cap X_{G||I}^O$ so that $\phi_G^O \wedge \phi_I^O$ is true. □

3.2 Refinement

Besides composition, the second operation on interfaces is refinement. Refinement between A/G interfaces is, like subtyping for function types, defined in an input/output contravariant fashion: an implementation must accept all inputs that the specification accepts, and it may produce only outputs that the specification allows. Hence, to refine an A/G interface, the input assumption can be weakened, and the output guarantee can be strengthened.

Definition 3. *An A/G interface* F' refines *an A/G interface* F*, written* $F \succeq F'$*, if*

(1) $X_F^I \subseteq X_{F'}^I$ *and* $X_F^O \supseteq X_{F'}^O$*;*
(2) $\phi_F^I \Rightarrow \phi_{F'}^I$ *and* $\phi_F^O \Leftarrow \phi_{F'}^O$*.*

Refinement between A/G interfaces is a preorder (i.e., reflexive and transitive). The following theorem shows that the A/G interfaces support independent implementability.

Theorem 2. *For all A/G interfaces* F*,* G*, and* F'*, if* $F \sim G$ *and* $F \succeq F'$*, then* $F' \sim G$ *and* $F||G \ \succeq \ F'||G$*.*

Proof sketch. From $X_F^O \cap X_G^O = \emptyset$ and $X_F^O \supseteq X_{F'}^O$, it follows that $X_{F'}^O \cap X_G^O = \emptyset$. Choose values for the input variables in $X_{F||G}^I$ so that

$$(\forall X_F^O \cup X_G^O)(\phi_F^O \wedge \phi_G^O \ \Rightarrow \ \phi_F^I \wedge \phi_G^I).$$

From $X_F^I \subseteq X_{F'}^I$ and $X_F^O \supseteq X_{F'}^O$, it follows that $X_{F||G}^I \subseteq X_{F'||G}^I$. Choose arbitrary values for all variables not in $X_{F||G}^I$. Then $\phi_{F'}^O \wedge \phi_G^O \ \Rightarrow \ \phi_{F'}^I \wedge \phi_G^I$ follows from $\phi_{F'}^O \Rightarrow \phi_F^O$ and $\phi_F^O \wedge \phi_G^O \ \Rightarrow \ \phi_F^I \wedge \phi_G^I$ and $\phi_F^I \Rightarrow \phi_{F'}^I$. This proves that $F' \sim G$. The proof that $F||G \ \succeq \ F'||G$ is straight-forward. □

Note that the contravariant definition of refinement is needed for Theorem 7 to hold, as input assumptions and output guarantees occur on two different sides of the implication in the formula (ψ).

We have not fixed the types of variables, nor the theory in which input assumptions or output guarantees are written. Checking the compatibility of A/G interfaces, and checking refinement between A/G interfaces, requires a procedure that decides the satisfiability of universal formulas in that theory. For example, if all variables are boolean, then the input assumptions and output guarantees are quantifier-free boolean formulas. In this case, compatibility checking requires the evaluation of $\exists\forall$ boolean formulas (namely, satisfiability checking of the universal formula (ψ)), and refinement checking requires the evaluation of $\forall$ boolean formulas (namely, validity checking of the two implications of Definition 6).

4 Automaton Interfaces

We now present the stateful interface language called *interface automata* [5]. An interface automaton is an edge-labeled digraph whose vertices represent interface states, whose edges represent interface transitions, and whose labels represent action names. The actions are partitioned into input, output, and internal actions. The internal actions are "hidden": they cannot be observed by the environment. The syntax of interface automata is identical to the syntax of I/O automata [8], but composition will be defined differently.

Definition 4. *An* interface automaton $F = \langle Q, q^0, A^I, A^O, A^H, \delta \rangle$ *consists of*

- *a finite set* Q *of* states*;*
- *an* initial *state* $q^0 \in Q$*;*
- *three pairwise disjoint sets* A^I*,* A^O*, and* A^H *of* input*,* output*, and* hidden *actions;*
- *a set* $\delta \subseteq Q \times A \times Q$ *of* transitions*, where* $A = A^I \cup A^O \cup A^H$ *is the set of all* actions*.*

We require that the automaton F *be* input-deterministic*, that is, for all states* $q, q', q'' \in Q$ *and all input actions* $a \in A^I$*, if* $(q, a, q') \in \delta$ *and* $(q, a, q'') \in \delta$*, then* $q' = q''$*.*

An action $a \in A$ is *enabled* at a state $q \in Q$ if there exists a state $q' \in Q$ such that $(q, a, q') \in \delta$. Given a state $q \in Q$, we write $A^I(q)$ (resp. $A^O(q)$; $A^H(q)$) for the set of input (resp. output; hidden) actions that are enabled at q. Unlike I/O automata, an interface automaton is not required to be input-enabled; that is, we do not require that $A^I(q) = A^I$ for all states $q \in Q$. Rather, we use the set $A^I(q)$ to specify the input actions that are accepted at state q; that is, an interface automaton encodes the assumption that, when F is in state q, the environment does not provide an input action that is not enabled at q.

Example 3. We model a software component that implements a message-transmission service. The component has a method "*send*" for sending messages. When this method is called, the component returns either "*ok*" or "*fail*." To perform this service, the component relies on a communication channel that provides the method "*trnsmt*" for transmitting messages. The two possible return values are "*ack*," which indicates a successful transmission, and "*nack*," which indicates a failure. When the method "*send*" is called, the component tries to transmit the message, and if the first transmission fails, it tries again. If both transmissions fail, the component reports failure. The interface automaton modeling this component is called *TryTwice* and shown in Figure 1.

The interface automaton *TryTwice* has the three input actions "*send*," "*ack*," and "*nack*"; the three output actions "*trnsmt*," "*ok*," and "*fail*"; and no hidden actions. It has seven states, with state 0 being initial (marked by an arrow without source). On the transitions, we append to the name of the action label the symbol "?" (resp. "!"; ";") to indicate that the transition is input (resp.

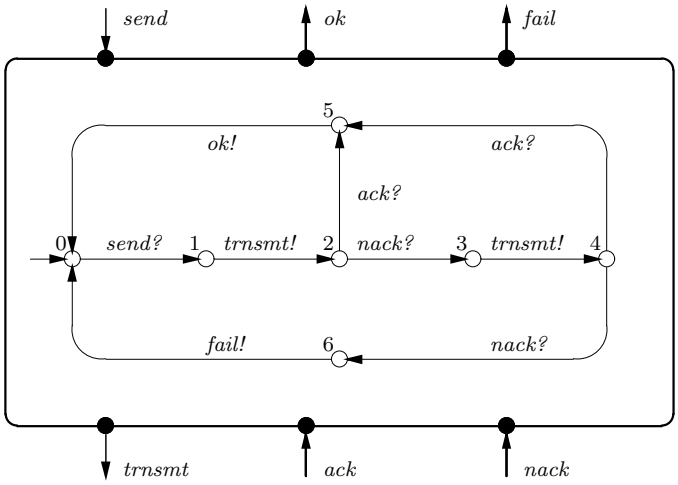

Fig. 1. The interface automaton *TryTwice*.

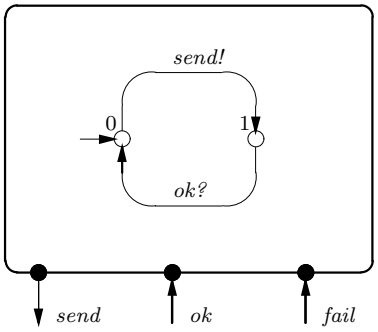

Fig. 2. The interface automaton *Client*.

output; hidden). Note how the automaton expresses in a straight-forward manner the above informal description of the message-passing service. The input "*send*" is accepted only in state 0; that is, the component expects a client to send a second message only after it has received an "*ok*" or "*fail*" response.

The interface automaton *Client* of Figure 2 shows a possible client of the message-passing service. It has the input actions "*ok*" and "*fail*," the output action "*send*," and again no hidden actions. This particular client expects messages to be sent successfully, and makes no provisions for handling failures: after calling the method "*send*," it accepts the return value "*ok*," but does not accept the return value "*fail*." The expectation that the return value is always "*ok*" is an assumption by the component *Client* about its environment; that is, the component *Client* is designed to be used only with message-transmission services that cannot fail. □

An interface automaton F is *closed* if it has no input and output actions; that is, if $A^I = A^O = \emptyset$. Closed interface automata do not interact with the environment.

An *execution* of the interface automaton F is a finite alternating sequence $q_0, a_0, q_1, a_1, \ldots, q_n$ of states and actions such that $(q_i, a_i, q_{i+1}) \in \delta$ for all $0 \leq i < n$. The execution is *autonomous* if all its actions are output or hidden actions; that is, if $a_i \in A^O \cup A^H$ for all $0 \leq i < n$. Autonomous executions do not depend on input actions. The execution is *invisible* if all its actions are hidden; that is, if $a_i \in A^H$ for all $0 \leq i < n$. A state $q' \in Q$ is (*autonomously*; *invisibly*) *reachable from* a state $q \in Q$ if there exists an (autonomous; invisible) execution whose first state is q, and whose last state is q'. The state q' is *reachable in* F if q' is reachable from the initial state q^0. In the definition of interface automata, it is not required that all states be reachable. However, one is generally not interested in states that are not reachable, and they can be removed.

4.1 Compatibility and composition

We define the composition of two interface automata only if their actions are disjoint, except that an input action of one automaton may coincide with an output action of the other automaton.

Definition 5. *Two interface automata F and G are* composable *if*

(1) $A_F^H \cap A_G = \emptyset$ *and* $A_F \cap A_G^H = \emptyset$*;*
(2) $A_F^I \cap A_G^I = \emptyset$*;*
(3) $A_F^O \cap A_G^O = \emptyset$*.*

For two interface automata F and G, we let $\mathit{shared}(F, G) = A_F \cap A_G$ be the set of common actions. If F and G are composable, then $\mathit{shared}(F, G) = (A_G^I \cap A_F^O) \cup (A_F^I \cap A_G^O)$. We define the composition of interface automata in stages, first defining the product automaton $F \otimes G$. The two automata synchronize on the actions in $\mathit{shared}(F, G)$, and asynchronously interleave all other actions. Shared actions become hidden in the product.

Definition 6. *For two composable interface automata F and G, the* product $F \otimes G$ *is the interface automaton with*

- $Q_{F\otimes G} = Q_F \times Q_G$;
- $q^0_{F\otimes G} = (q^0_F, q^0_G)$;
- $A^I_{F\otimes G} = (A^I_F \cup A^I_G) \backslash shared(F,G)$;
- $A^O_{F\otimes G} = (A^O_F \cup A^O_G) \backslash shared(F,G)$;
- $A^H_{F\otimes G} = A^H_F \cup A^H_G \cup shared(F,G)$;
- $((q,r),a,(q',r')) \in \delta_{F\otimes G}$ *iff*
  $a \notin shared(F,G)$ *and* $(q,a,q') \in \delta_F$ *and* $r = r'$, *or*
  $a \notin shared(F,G)$ *and* $q = q'$ *and* $(r,a,r') \in \delta_G$, *or*
  $a \in shared(F,G)$ *and* $(q,a,q') \in \delta_F$ *and* $(r,a,r') \in \delta_G$.

Let $\delta^I_F = \{(q,a,q') \in \delta \mid a \in A^I_F\}$ denote the set of input transitions of an interface automaton F, and let δ^O_F and δ^H_F be defined similarly as the output and hidden transitions of F. Then according to the definition of product automata, each input transition of $F \otimes G$ is an input transition of either F or G; that is, $((q,r),a,(q',r')) \in \delta^I_{F\otimes G}$ iff

$(q,a,q') \in \delta^I_F$ and $a \notin A^O_G$ and $r = r'$; or
$a \notin A^O_F$ and $q = q'$ and $(r,a,r') \in \delta^I_G$.

Each output transitions of $F \otimes G$ is an output transition of F or G; that is, $((q,r),a,(q',r')) \in \delta^O_{F\otimes G}$ iff

$(q,a,q') \in \delta^O_F$ and $a \notin A^I_G$ and $r = r'$; or
$a \notin A^I_F$ and $q = q'$ and $(r,a,r') \in \delta^O_G$.

Each hidden transition of $F \otimes G$ is either an input transition of F that is an output transition of G, or vice versa, or it is a hidden transition of F or G; that is, $((q,r),a,(q',r')) \in \delta^H_{F\otimes G}$ iff

$(q,a,q') \in \delta^I_F$ and $(r,a,r') \in \delta^O_G$; or
$(q,a,q') \in \delta^O_F$ and $(r,a,r') \in \delta^I_G$; or
$(q,a,q') \in \delta^H_F$ and $r = r'$; or
$q = q'$ and $(r,a,r') \in \delta^H_G$.

Example 4. The product *TryTwice* $\otimes$ *Client* of the interface automata *TryTwice* and *Client* from Figures 1 and 2 is shown in Figure 3. Each state of the product consists of a state of *TryTwice* together with a state of *Client*. Only the reachable states of the product automaton are shown. Each transition of the product is either a joint "*send*" transition, which represents the call of the method "*send*" by *Client*, or a joint "*ok*" transition, which represents the termination of the method "*send*" with return value "*ok*," or a transition of *TryTwice* calling the method "*trnsmt*" of the (unspecified) communication channel, or a transition of *TryTwice* receiving the return value "*ack*" or "*nack*" from the channel.

Consider the following sequence of events. The component *Client* calls the method "*send*"; then *TryTwice* calls twice the method "*trnsmt*" and receives

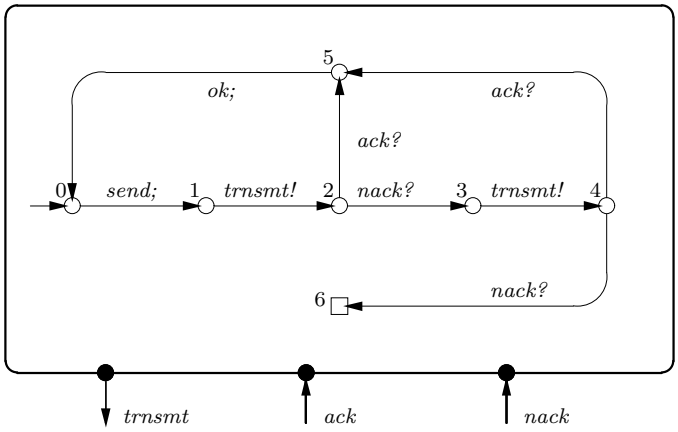

Fig. 3. The product automaton $TryTwice \otimes Client$.

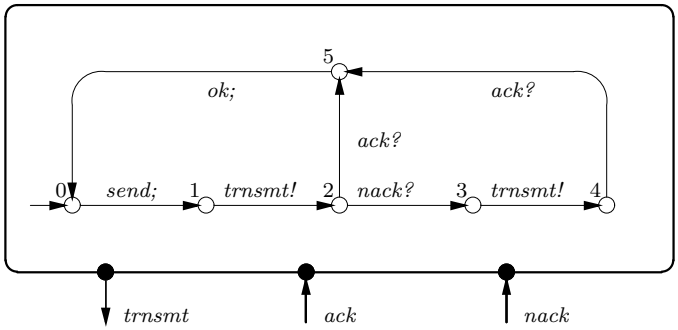

Fig. 4. The composite interface automaton $TryTwice||Client$.

twice the return value "*nack*," indicating transmission failure. This sequence of events brings us to state 6 of the product automaton, which corresponds to state 6 of *TryTwice* and state 1 of *Client*. In state 6, the component *TryTwice* tries to report failure by returning "*fail*," but not expecting failure, the component *Client* does not accept the return value "*fail*" in state 1. Hence the product state 6 has no outgoing edges; it is called an *error* state, because at product state 6, the component *TryTwice* violates the assumption made by the component *Client* about the inputs that *Client* receives. □

This example illustrates that, as interface automata are not necessarily input-enabled, in the product of two interface automata, one of the automata may produce an output action that is in the input alphabet of the other automaton, but is not accepted.

Definition 7. *Given two composable interface automata F and G, a product state $(q, r) \in Q_F \times Q_G$ is an* error *state of the product automaton $F \otimes G$ if there exists an action $a \in shared(F, G)$ such that either $a \in A_F^O(q)$ and $a \notin A_G^I(r)$, or $a \notin A_F^I(q)$ and $a \in A_G^O(r)$. We write $error(F, G)$ for the set of error states of the product automaton $F \otimes G$.*

If the product $F \otimes G$ contains no reachable error states, then the two interface automata F and G satisfy each other's input assumptions and are thus compatible. On the other hand, if $F \otimes G$ is closed and contains a reachable error state, then F and G are incompatible. The interesting case arises when $F \otimes G$ contains reachable error states, but is not closed. The fact that a state in $error(F, G)$ is reachable does not necessarily indicate an incompatibility, because by providing appropriate inputs, the environment of $F \otimes G$ may be able to ensure that no error state is encountered in the product. We therefore define the set of incompatible states of $F \otimes G$ as those states from which no environment can prevent that an error state may be entered. First, the error states of $F \otimes G$ are incompatible. Second, all states from which a sequence of output or hidden actions of $F \otimes G$ leads to an error state are also incompatible, because the product automaton may choose to traverse that sequence in every environment. On the other hand, if an error state is only reachable via an input action, then a helpful environment can choose not to provide that action, thus avoiding the error state.

Definition 8. *Given two composable interface automata F and G, a product state $(q, r) \in Q_F \times Q_G$ is a* compatible *state of the product automaton $F \otimes G$ if there exists no error state $(q', r') \in error(F, G)$ that is autonomously reachable from (q, r). Two interface automata F and G are* compatible*, written $F \sim G$, if they are composable and the initial state of the product automaton $F \otimes G$ is compatible.*

Note that the compatibility relation $\sim$ is symmetric.

If two composable interface automata F and G are compatible, then there is an environment E such that

(1) E is composable with $F \otimes G$;
(2) $(F \otimes G) \otimes E$ is closed;
(3) for all states $((q, r), s) \in (Q_F \times Q_G) \times Q_E$ that are reachable in $(F \otimes G) \otimes E$, we have $(q, r) \notin \mathit{error}(F, G)$ and $((q, r), s) \notin \mathit{error}(F \otimes G, E)$.

The third condition ensures that (3a) E prevents the error states of $F \otimes G$ from being entered, and (3b) E accepts all outputs of $F \otimes G$ and does not provide inputs that are not accepted by $F \otimes G$. An interface automaton that satisfies the conditions (1)–(3) is called a *legal environment* for $F \otimes G$. The existence of a legal environment shows that two compatible interfaces can be used together in *some* context.

For compatible interface automata F and G, there is always a trivial legal environment, which provides no inputs to $F \otimes G$. Formally, *empty closure* of F and G is the interface automaton $\mathit{close}(F, G)$ with

$$
\begin{aligned}
Q_{\mathit{close}(F,G)} &= \{0\};\\
q^0_{\mathit{close}(F,G)} &= 0;\\
A^I_{\mathit{close}(F,G)} &= A^O_{F\otimes G};\\
A^O_{\mathit{close}(F,G)} &= A^I_{F\otimes G};\\
A^H_{\mathit{close}(F,G)} &= \emptyset;\\
\delta_{\mathit{close}(F,G)} &= \{(0, a, 0) \mid a \in A^I_{\mathit{close}(F,G)}\}.
\end{aligned}
$$

The empty closure $\mathit{close}(F, G)$ has a single state (arbitrarily named 0), which is its initial state. It accepts all output actions of $F \otimes G$ as inputs, but does not issue any outputs. All states that are reachable in $(F \otimes G) \otimes \mathit{close}(F, G)$ are reachable solely by output and hidden actions of $F \otimes G$. Thus, if the initial state of $F \otimes G$ is compatible, then each state that is reachable in $(F \otimes G) \otimes \mathit{close}(F, G)$ must not correspond to an error state of $F \otimes G$. On the other hand, if the initial state of $F \otimes G$ is not compatible, then some error state of $F \otimes Q$ is reachable in every environment that does not constrain the outputs of $F \otimes G$, in particular, in $(F \otimes G) \otimes \mathit{close}(F, G)$. Consequently, two interface automata F and G are compatible iff for all states $(q, r) \in Q_F \times Q_G$ such that $((q, r), 0)$ is reachable in $(F \otimes G) \otimes \mathit{close}(F, G)$, we have $(q, r) \notin \mathit{error}(F, G)$.

The composition of two compatible interface automata is obtained by restricting the product of the two automata to the set of compatible states.

Definition 9. *Given two compatible interface automata F and G, the* composition $F||G$ *is the interface automaton that results from the product $F \otimes G$ by removing all transitions $(q, a, q') \in \delta_{F\otimes G}$ such that*

(1) q is a compatible state of the product $F \otimes G$;
(2) $a \in A^I_{F\otimes G}$ is an input action of the product;
(3) q' is not a compatible state of $F \otimes G$.

Example 5. In the product automaton *TryTwice* $\otimes$ *Client* from Figure 3, state 6 is an error state, and thus not compatible. However, the product *TryTwice* $\otimes$ *Client* is not closed, because its environment —the communication channel—

provides "*ack*" and "*nack*" inputs. The environment that provides input "*ack*" (or no input at all) at the product state 4 ensures that the error state 6 is not entered. Hence, the product states 0, 1, 2, 3, 4, and 5 are compatible. Since the initial state 0 of the product is compatible, the two interface automata *TryTwice* and *Client* are compatible. The result of removing from *TryTwice* $\otimes$ *Client* the input transition to the incompatible state 6 is the interface automaton *TryTwice*$||$*Client* shown in Figure 4.

Note that restricting *TryTwice* $\otimes$ *Client* to its compatible states corresponds to imposing an assumption on the environment, namely, that calls to the method "*send*" never return twice in a row the value "*nack*." Hence, when the two interface automata *TryTwice* and *Client* are composed, the assumption of *Client* that no failures occur is translated into the assumption of *TryTwice*$||$*Client* that no two consecutive transmissions fail. The illustrates how the composition of the interface automata *TryTwice* and *Client* propagates to the environment of *TryTwice*$||$*Client* the assumptions that are necessary for the correct interaction of *TryTwice* and *Client*. $\square$

The definition of composition removes only transitions, not states, from the product automaton. The removal of transitions, however, may render some states unreachable, which can then be removed also. In particular, as far as reachable states are concerned, the composition $F||G$ results from the product $F \otimes G$ by removing all incompatible states; if the result is empty, then F and G are not compatible. In general, the removal of input transitions from a product automaton may render even some compatible states unreachable. Hence, the relevant states of the composite automaton $F||G$ are those states of the product automaton $F \otimes G$ which remain reachable after all incompatible states are removed. Those states of the product automaton can be found in linear time, by forward and backward traversals of the underlying graph [5]. Thus the compatibility of two interface automata with m_1 and m_2 reachable transitions, respectively, can be checked, and their composition constructed, in time $O(m_1 \cdot m_2)$.

The following theorem shows that interface automata support incremental design.

Theorem 3. *For all interface automata F, G, H, and I, if $F \sim G$ and $H \sim I$ and $F||G \sim H||I$, then $F \sim H$ and $G \sim I$ and $F||H \sim G||I$.*

Proof sketch. For composability, note that from the premises of the theorem it follows that A_i^H is disjoint from A_j for all $j \neq i$, that all A_i^I's are pairwise disjoint, and that all A_i^O's are pairwise disjoint. Consider the product automaton $F \otimes G \otimes H \otimes I$; the associativity of $\otimes$ implies that parentheses do not matter. Define a state (p_1, p_2, p_3, p_4) to be an error state of $F \otimes G \otimes H \otimes I$ if some pair (p_i, p_j) is an error state of the corresponding subproduct; e.g., if (p_1, p_3) is an error state of $F \otimes H$. Define a state p to be an incompatible state of $F \otimes G \otimes H \otimes I$ if some error state of $F \otimes G \otimes H \otimes I$ is autonomously reachable from p, that is, reachable via hidden and output transitions. For $\ell \geq 0$, define a state p to be a rank-ℓ incompatible state if some error state is autonomously reachable from p in at most ℓ transitions.

We show that under the premises of the theorem, the composition $F||G||H||I$ is achieved, for any insertion of parentheses, by removing the incompatible states from the product $F \otimes G \otimes H \otimes I$. The proof proceeds in two steps. First, we show that if some projection of a product state $p = (p_1, p_2, p_3, p_4)$ is an incompatible state of the corresponding subproduct (say, $F \otimes G$), then p is an incompatible state of the full product $F \otimes G \otimes H \otimes I$. Second, we show that if p is an incompatible state of $F \otimes G \otimes H \otimes I$, and some input transitions are removed by constructing the composition of any subproduct (say, $F||H$), then even in the product without the removed transitions, there remains an autonomous path from p to an error state.

(1) Consider a state p of the product $F \otimes G \otimes H \otimes I$, and a projection p' of p which is a rank-ℓ incompatible state of the corresponding subproduct. We show that p is a rank-ℓ' incompatible state of $F \otimes G \otimes H \otimes I$ for some $\ell' \leq \ell$. Consider a shortest autonomous path from p' to an error state in the subproduct. There are three cases. First, if p' is an error state of the subproduct (rank 0), then p is an error state of the full product. Second, if the first transition of the error path from p' in the subproduct corresponds to an output or hidden transition of the full product, then the rank of the successor state is $\ell - 1$ and the claim follows by induction. Third, if the first transition of the error path from p' is an output transition of the subproduct which does not have a matching input transition in the full product, then p is an error state of the full product and has rank 0.

(2) Consider an incompatible state p of the product $F \otimes G \otimes H \otimes I$. Suppose that some input transitions are removed by constructing the composition of a subproduct, and remove the corresponding transitions in the full product $F \otimes G \otimes H \otimes I$. The only kind of transition that might be removed in this way is a hidden transition (q, a, r) of the product whose projection onto the subproduct is an input transition (q', a, r'), which is matched in the full product by an output transition. Once (q, a, r) is removed, the input action a is no longer enabled at the state q', because interface automata are input-deterministic. Hence in the full product, the state q is an error state. Therefore even after the removal of the transition (q, a, r) from the product $F \otimes G \otimes H \otimes I$, there is an autonomous path from p to an error state, namely, to q. □

As a consequence of Theorem 17, we can check whether $k > 0$ interface automata $F_1, \ldots, F_k$ are compatible by computing their composition $F_1|| \cdots ||F_k$ incrementally, by adding one interface automaton at a time. The potential efficiency of the incremental product construction lies in the fact that product states can be pruned as soon as they become either incompatible, or unreachable through the pruning of incompatible states. Thus, in some cases the exponential explosion of states inherent in a product construction may be avoided.

4.2 Refinement

In the stateful input-enabled setting, refinement is usually defined as trace containment or simulation; this ensures that all output behaviors of the implementation are allowed by the specification. However, such definitions are not

appropriate in a non-input-enabled setting, such as interface automata: if one were to require that the set of accepted *inputs* of the implementation is a subset of the inputs allowed by the specification, then the implementation would make stronger assumptions about the environment, and could not be used in all contexts in which the specification is used.

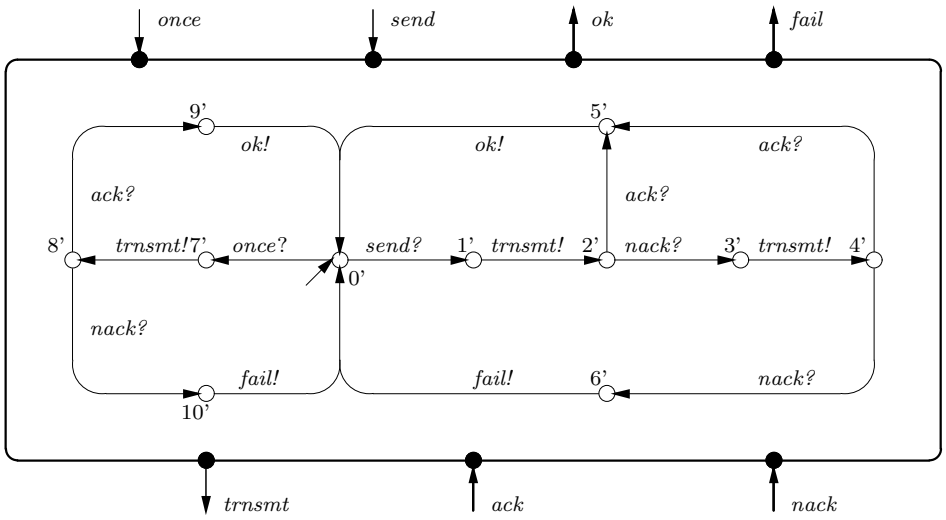

Fig. 5. The interface automaton *OnceOrTwice*.

Example 6. Consider the interface automaton *OnceOrTwice* of Figure 5. This automaton represents a component that provides two services: the first is the try-twice service "*send*" provided also by the automaton *TryTwice* of Figure 1; the second is a try-once-only service "*once*" designed for messages that are useless when stale. Clearly, we would like to define refinement so that *OnceOrTwice* is a refinement of *TryTwice*, because the component *OnceOrTwice* implements all services provided by the component *TryTwice*, and it is consistent with *TryTwice* in their implementation. Hence, in all contexts in which *TryTwice* is used, *OnceOrTwice* can be used instead. The language of *OnceOrTwice*, however, is not contained in the language of *TryTwice*; indeed, "*once*" is not even an action of *TryTwice*. □

Therefore, for interface automata we define refinement in a contravariant fashion: the implementation must accept more inputs, and provide fewer outputs, than the specification. For efficient checkability of refinement, we choose a contravariant refinement relation in the spirit of simulation, rather than in the spirit of language containment. This leads to the definition of refinement as *alternating* simulation [1]. Roughly, an interface automaton F' refines an interface

automaton F if each input transition of F can be simulated by F', and each output transition of F' can be simulated by F. The precise definition must take into account the hidden transitions of F and F'.

The environment of an interface automaton F cannot see the hidden transitions of F. Consequently, if F is at a state q, and state r is invisibly reachable from q (by hidden actions only), then the environment cannot distinguish between q and r. Given a state $q \in Q$, let $\varepsilon\text{-}closure(q)$ be the set of states that are invisibly reachable from q. The environment must be able to accept all output actions in the set

$$obsA^O(q) = \{a \in A^O \mid (\exists r \in \varepsilon\text{-}closure(q))(a \in A^O(r))\}$$

of outputs that may follow after some sequence of hidden transitions from q. Conversely, the environment can safely issue all input actions in the set

$$obsA^I(q) = \{a \in A^I \mid (\forall r \in \varepsilon\text{-}closure(q))(a \in A^I(r))\}$$

of inputs that are accepted after all sequences of hidden transitions from q. For an implementation state q' to refine a specification state q we need to require that $obsA^I(q) \subseteq obsA^I(q')$ and $obsA^O(q) \supseteq obsA^O(q')$. Alternating simulation propagates this requirement from q and q' to their successor states.

To define alternating simulation formally, we use the following notation. Given a state $q \in Q$ and an action $a \in A$ of an interface automaton, let $post(q, a) = \{r \in Q \mid (q, a, r) \in \delta\}$ be the set of a-successors of q.

Definition 10. *Given two interface automata F and F', a binary relation $\succeq \subseteq Q_F \times Q_{F'}$ is an* alternating simulation by F of F' *if $q \succeq q'$ implies*

(1) for all input actions $a \in A^I(q)$ and states $r \in post(q, a)$, there is a state $r' \in post(q', a)$ such that $r \succeq r'$;
(2) for all output actions $a \in A^O(q')$ and states $r' \in post(q', a)$, there is a state $p \in \varepsilon\text{-}closure(q)$ and a state $r \in post(p, a)$ such that $r \succeq r'$;
(3) for all hidden actions $a \in A^H(q')$ and states $r' \in post(q', a)$, there is a state $r \in \varepsilon\text{-}closure(q)$ such that $r \succeq r'$.

Conditions (1) and (2) express the input/output duality between states $q \succeq q'$ in the alternating simulation relation: every input transition from q must be matched by an input transition from q', and every output transition from q' must be matched by a sequence of zero or more hidden transitions from q followed by an output transition. Condition (3) stipulates that every hidden transition from q' can be matched by a sequence of zero or more hidden transitions from q. In all three cases, matching requires that the alternating-simulation relation is propagated co-inductively. Since interface automata are input-deterministic, condition (1) can be rewritten as (1a) $A^I(q) \subseteq A^I(q')$ and (1b) for all input transitions $(q, a, r), (q', a, r') \in \delta^I$, we have $r \succeq r'$. It can be checked that if $q \succeq q'$ for some alternating simulation $\succeq$, then $obsA^I(q) \subseteq obsA^I(q')$ and $obsA^O(q) \supseteq obsA^O(q')$.

Definition 11. *An interface automaton F'* refines *an interface automaton F, written $F \succeq F'$, if*

(1) $A^I_F \subseteq A^I_{F'}$ and $A^O_F \supseteq A^O_{F'}$;
(2) there is an alternating simulation $\succeq$ by F of F' such that $q^0_F \succeq q^0_{F'}$.

Note that unlike in standard simulation, the "typing" condition (1) is contravariant on the input and output action sets. This captures a simple kind of subclassing: if $F \succeq F'$, then the implementation F' is able to provide more services than the specification F, but it must be consistent with F on the shared services. Condition (2) relates the initial states of the two automata.

Example 7. In the example of Figures 1 and 5, there is an alternating simulation that relates q with q' for all $q \in \{0, 1, 2, 3, 4, 5, 6\}$. Hence *OnceOrTwice* refines *TryTwice*. □

It can be shown that refinement relation between interface automata is a preorder. Refinement can be checked in polynomial time. More precisely, if F has n_1 reachable states and m_1 reachable transitions, and F' has n_2 reachable states and m_2 reachable transitions, then it can be checked in time $O((m_1 + m_2) \cdot (n_1 + n_2))$ whether $F \succeq F'$ [1].

The following theorem shows that interface automata support independent implementability: we can always replace an interface automaton F with a more refined version F' such that $F \succeq F'$, provided that F and F' are connected to the environment by the same inputs. The side condition is due to the fact that if the environment were to provide inputs for F' that are not provided for F, then it would be possible that new incompatibilities arise in the processing of these inputs. For software components, independent implementability is a statement of subclass polymorphism: we can always substitute a subclass for a superclass, provided no new methods of the subclass are used.

Theorem 4. *Consider three interface automata F, G, and F' such that F and G are composable and $\mathit{shared}(F', G) \subseteq \mathit{shared}(F, G)$. If $F \sim G$ and $F \succeq F'$, then $F' \sim G$ and $F||G \succeq F'||G$.*

Proof sketch. The typing conditions are straight-forward to check. Note in particular that $\mathit{shared}(F', G) \subseteq \mathit{shared}(F, G)$ implies both $A^H_{F'} \cap A_G = \emptyset$ and $(A^I_{F'} \backslash A^I_F) \cap A^O_G = \emptyset$.

To prove that $F' \sim G$ under the premises of the theorem, we show that every autonomous path leading from the initial state to an error state of $F' \otimes G$ can be matched, transition by transition, by an autonomous path leading from the initial state to an error state of $F \otimes G$. The interesting case is that of an input transition of F' in the product $F' \otimes G$, say on action a. Since the path is autonomous, the input action a of F' must be an output action of G, and because $\mathit{shared}(F', G) \subseteq \mathit{shared}(F, G)$, the action a must also be an input action of F. If a is not enabled in F, then we have already hit an error state of $F \otimes G$; otherwise, there are unique a-successors in both F and F', and the path matching can continue.

Finally, to prove that $F||G \preceq F'||G$ under the premises of the theorem, consider an alternating simulation $\preceq$ by F of F' such that $q_F^0 \preceq q_{F'}^0$. Then an alternating simulation $\preceq'$ by $F||G$ of $F'||G$ can be defined as follows: let $(p, r) \preceq' (p', r')$ iff (1) $p \preceq p'$, (2) $r = r'$, and (3) (p, r) is not an error state of $F \otimes G$. □

The property of independent implementability implies that refinement is compositional: in order to check if $F||G \preceq F'||G'$, it suffices to check both $F \preceq F'$ and $G \preceq G'$. This observation allows the decomposition of refinement proofs. Decomposition is particularly important in the case of interface automata, where the efficiency of refinement checking depends on the number of states.

5 Discussion

An interface automaton represents both assumptions about the environment, and guarantees about the specified component. The environment assumptions are twofold: (1) each output transition incorporates the assumption that the corresponding action is accepted by the environment as input; and (2) each input action that is not accepted at a state encodes the assumption that the environment does not provide that input. The component guarantees correspond to possible sequences and choices of input, output, and hidden actions, as usual. When two interface automata are composed, the composition operator || combines not only the component guarantees, as is the case in other component models, but also the environment assumptions.

Whenever two interface automata F and G are compatible, there is a particularly simple legal environment, namely, the empty closure $close(F, G)$. This points to a limitation of interface automata: while the environment assumption of an automaton can express which inputs may occur, it cannot express which inputs *must* occur. Thus, the environment that provides no inputs is always the best environment for showing compatibility. There are several ways of enriching interface automata to specify inputs that must occur, among them, synchronicity [3, 6], adding fairness [4], or adding real-time constraints [7]. In these cases, no generic best environment exists, and a legal environment must be derived as a winning strategy in a two-player game. Recall that two interfaces F and G are compatible iff the environment has a strategy to avoid incompatible states of the product $F \otimes G$. In this game, player-1 is the environment, which provides inputs to the product $F \otimes G$, and player-2 is the "team" $\{F, G\}$ of interfaces, which choose internal transitions and outputs of $F \otimes G$. The game aspect of compatibility checking is illustrated by the following example of a stateful, synchronous extension of assume/guarantee interfaces [3].

Example 8. Suppose that F and G are two generalized A/G interfaces, which receive inputs and issue outputs in a sequence of rounds and may change, in each round, their input assumptions and output guarantees. The interface F has no inputs and the single output variable x; the interface G has the two input

variables x and y, and no outputs. In the first round, the interface F either goes to state q_0 and outputs $x = 0$, or it goes to state q_1 and outputs $x \neq 0$. Also in the first round, on input $y = 0$ the interface G goes to state r_0, and on input $y \neq 0$ it goes to state r_1. In the second round, in state q_0 the interface F outputs $x = 0$, and in state q_1 it outputs $x \neq 0$, after which it goes back to the initial state. Also the second round, in state r_0 the interface G has the input assumption $x = 0$, and in state r_1 it has the input assumption $x \neq 0$. After the second round, also G returns to its initial state and the process repeats ad infinitum.

Note that the state q_0 of interface F is compatible with the state r_0 of interface G, and q_1 is compatible with r_1, but q_0 is not compatible with r_1, and q_1 is not compatible with r_0. The environment provides the input y to the interface G in every round. The environment can avoid incompatibilities by copying, in each (odd) round, the value of x into y. In this way the environment can ensure that F and G are always in compatible states. Hence the two interfaces F and G are compatible. The helpful strategy of the environment can be synthesized as a winning strategy of the two-player game "environment" versus "interfaces." In this simple example, it is a game with complete information, because at all times the environment, by observing the output x of F, can deduce the internal state of F. □

In the presence of hidden transitions, interface languages must be designed carefully. This is because if the state of an interface is not visible to the environment, then a legal environment corresponds to a winning strategy in a game with *partial* information. The derivation of such strategies requires, in general, exponential time, by involving a subset construction that considers all sets of possible interface states [9]. Any model with an exponential cost for binary composition, however, is unlikely to be practical. This is why we have focused, in this article, on the asynchronous case with hidden transitions, and elsewhere [3, 6, 7], on more general, synchronous and real-time interfaces but without hidden transitions. Another interesting direction is to investigate stronger but more efficient compatibility checks, which consider only restricted sets of strategies for the environment. Such checks would be conservative (i.e., sufficient but not necessary) yet might still achieve the desired properties of incremental design and independent implementability.

Rich, stateful interfaces as games have been developed further in [2, 4]. In the former article, multiple instances of a component, such as a recursive software module, may be active simultaneously. Compatibility checking for the corresponding interface language is based on solving push-down games. In the latter article, the notion of error state is generalized to handle resource constraints of a system: an error occurs if two or more components simultaneously access or overuse a constrained resource. Critical resources may include power, buffer capacity, or cost. While the basic set-up of the game "environment" versus "interfaces" remains the same, the objective function of the game changes and may include quantitative aspects, such as minimizing resource use.

Acknowledgment. We thank Mariëlle Stoelinga for pointing out errors in a previous version of this article.

References

[1] R. Alur, T.A. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *Proc. Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 163–178. Springer-Verlag, 1998.

[2] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules. In *Proc. Computer-Aided Verification*, Lecture Notes in Computer Science 2404, pages 428–441. Springer-Verlag, 2002.

[3] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *Proc. Computer-Aided Verification*, Lecture Notes in Computer Science 2404, pages 414–427. Springer-Verlag, 2002.

[4] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. Embedded Software*, Lecture Notes in Computer Science 2855, pages 117–133. Springer-Verlag, 2003.

[5] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proc. Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[6] L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *Proc. Embedded Software*, Lecture Notes in Computer Science 2211, pages 148–165. Springer-Verlag, 2001.

[7] L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Timed interfaces. In *Proc. Embedded Software*, Lecture Notes in Computer Science 2491, pages 108–122. Springer-Verlag, 2002.

[8] N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[9] J. Reif. The complexity of two-player games of incomplete information. *J. Computer and System Sciences*, 29:274–301, 1984.