Interface Automata*†

Luca de Alfaro Thomas A. Henzinger

Dept. of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1770
{dealfaro,tah}@eecs.berkeley.edu

Abstract. Conventional type systems specify interfaces in terms of values and domains. We present a light-weight formalism that captures the *temporal* aspects of software component interfaces. Specifically, we use an automata-based language to capture both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The formalism supports automatic compatibility checks between interface models, and thus constitutes a type system for component interaction. Unlike traditional uses of automata, our formalism is based on an *optimistic* approach to composition, and on an *alternating* approach to design refinement. According to the optimistic approach, two components are compatible if there is *some* environment that can make them work together. According to the alternating approach, one interface refines another if it has weaker input assumptions, and stronger output guarantees. We show that these notions have game-theoretic foundations that lead to efficient algorithms for checking compatibility and refinement.

1 Introduction

The purpose of a modeling language is to capture certain aspects of a design. For hardware, modeling languages (called hardware description languages) provide the basis for design, validation, and synthesis. For software, modeling languages such as UML [12] are becoming widely used in design and documentation, but except for very specific domains, they are either too informal or too heavy to be used effectively in validation. We present a light-weight formalism for capturing temporal aspects of software component interfaces which are beyond the reach of traditional type systems. Specifically, we use an automata-based language to capture assumptions about the order in which the methods of a component are called, and the order in which the component calls external methods. Since our language is formal, it can be used not only in design and documentation, but

*This research was supported in part by the AFOSR MURI grant F49620-00-1-0327, the DARPA MoBIES grant F33615-00-C-1703, the MARCO GSRC grant 98-DT-660, the NSF Theory grant CCR-9988172, and the NSF ITR grant CCR-0085949.

†A preliminary version of this paper will appear in the *Proceedings of the 9th Annual ACM Symposium on Foundations of Software Engineering* (FSE), 2001.

also in validation, in particular, for checking that the interfaces of two components are compatible. Since interfaces are often much simpler than the corresponding implementations, we believe that it is in this area where formal methods can be most effectively used to aid in software design.

The prevalent trend in software and system engineering is towards *component-based design*. According to this approach, new software designs are created by combining pre-existing modules with new software that provides both glue between the components, and new functionality. Indeed, component-based design, and the use of component libraries, are such standard concepts in current software engineering that new programming languages, such as Java, come already supplied with their component libraries, and the development of systems of any sophistication would be unthinkable without resorting to libraries of components. The effective reuse of components requires languages for the documentation and interface specification of components, along with methods for checking the compatibility of component interfaces in a design. The formalism of Statecharts [4], also used in UML, is widely used to document component behavior. Light-weight constraint-based languages for component specification have been presented in [5]. Automata-based specifications for component behavior have been considered in [1, 8]. Our formulation of interface compatibility is related in various respects to [13], which also considers the synthesis of component adaptors to bridge between incompatible interfaces.

We capture the temporal I/O behavior of a component by an automaton. While we are not the first to propose an automata-based formalism for modeling, our approach is, in a certain important sense, diametrically opposed to the traditional approach. The traditional approach to the modeling of a component that interacts with an environment (namely, other components) is *pessimistic*: the underlying assumption is that the environment is free to behave as it pleases, and that two components are compatible if *no* environment can lead them into an error state. Hence, the pessimistic approach considers two components compatible if they can be used together in all systems. We believe that for design purposes, where both the components and their environment are being designed, an *optimistic* view is more natural. Components are usually designed under assumptions about the environment; when they are composed, we should compose also their environment assumptions. Accordingly, two components are compatible if there is *some* environment that can make them work together, simultaneously satisfying both of their environment assumptions. Hence, under the optimistic approach two components are compatible if they can be used together in at least one design. We will demonstrate that the optimistic view, which assumes a helpful environment, leads to a simple, clean, and powerful theory of interface automata, which are typically smaller, and thus easier on the user, than the traditional pessimistic interface models, which must be prepared to cope with all environments.

Our formalism of *interface automata* is syntactically similar to the I/O automata of [9]. Interface automata interact through the synchronization of input and output actions, while the internal actions of concurrent automata are interleaved asynchronously. Input actions are used to model procedures or methods that can be called, and the receiving end of communication channels, as well as the return locations from such calls. Output actions are used to model procedure or method calls, message transmissions, the act of returning after a call or method terminates, and exceptions that arise during method execution. Components are designed under assumptions about their environment; that is, the design describes the behavior of the component only under environments that satisfy the assumptions. For example, the design of an object-oriented software component may assume that the method calls occur in a specific order, and it may behave as desired only in response to such properly ordered calls. In similar fashion, interface automata accept only some input behaviors generated by the environment, and they describe the component behavior under those inputs only. Unlike I/O automata, which at every state must be receptive towards

every possible input action (the "pessimistic," or *input-enabled*, view), for interface automata, at every state some inputs may be illegal. By not accepting certain inputs, the interface automaton expresses the assumption that the environment never generates these inputs (the "optimistic," or *environment-constraining*, view). In this way, environment assumptions can be used to encode restrictions on the order of method calls, and on the types of return values and exceptions. By capturing environment assumptions, and by freeing designers from the obligation of specifying responses to inputs that cannot occur in the intended environment, interface automata provide a concise and formal notation that parallels the natural way of evolving a component-based design.

The optimistic approach has several ramifications on the technical development of an interface modeling language. In component-based design, one wants to hand off and refine an interface, ultimately towards an implementation, independent of the design of other components. In the traditional, pessimistic view, where the interface captures only the legal component behaviors, refinement means to choose and implement a particular legal behavior. In the optimistic approach, the interface captures not only the legal component behaviors but also an assumption about the environment, in the form of permissible environment behaviors. Refinement, then, means to choose among the legal component behaviors without restricting the permissible environment behaviors. Mathematically speaking, refinement acts contravariantly on input assumptions and output guarantees: the former can be relaxed only; the latter can be restricted only. This leads to a notion of *alternating* simulation [2] as refinement.

Second, the optimistic view implies a notion of interface composition that leads to smaller compound automata than the pessimistic view. If two interface automata are composed, the result may contain *error states*, where one automaton generates an output that is an illegal input for the other automaton. While the standard pessimistic view would consider the two interfaces to be incompatible, we follow again an optimistic approach: just as each individual interface expects the environment to provide only legal inputs, so the compound interface expects the environment to steer away from all error states. The existence of a legal environment for the composition of two interfaces indicates that the interfaces are *compatible,* i.e., that there is a way to use the corresponding components together and, at the same time, ensure that the environment assumptions of both are met. The resulting *composite interface automaton* combines not only the behaviors of the two component interfaces, but also combines the environment assumptions of the components into the least restrictive composite environment assumption under which the components can work together. Algorithmically, the composite automaton is constructed by pruning from the product of the component automata all states from which the environment cannot prevent an error state from being entered in one or more steps. This algorithm solves a *game* between the product automaton (which attempts to get to an error state) and the environment (which attempts to prevent this). An interesting special case is that of *single-threaded* interface automata, used to model systems in which only one thread of execution is active at any given time. For these automata we introduce a specialized notion of composition, for which the product pruning leads to particularly big savings.

As a consequence of the pruning, the composition of two interface automata has a nonempty set of states iff the two automata are compatible. This underlines a difference between the optimistic approach, exemplified by interface automata, and the usual pessimistic approach. In the optimistic approach, composing two components is conceptually complex (but computationally linear time), because it requires the solution of a game between the components and the environment, but checking compatibility, once the composition has been computed, is trivial. In particular, the optimistic approach obviates the need for explicitly specifying consistency properties between components. In contrast, in the usual pessimistic approach, composing components is conceptually simple, but checking for compatibility requires human supervision for providing consistency properties, and

proof that they hold in the composed system.

By enabling automatic compatibility checks between component interfaces analogous to those afforded by conventional type systems, which focus on values and domains, interface automata offer a "type system for component interaction," as propagated in [7].

2 A Preview of Interface Automata

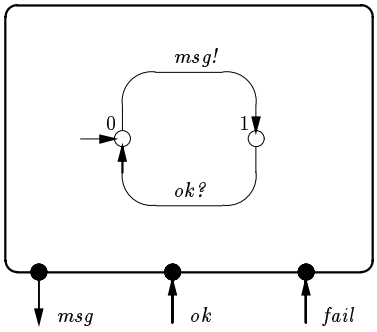
We illustrate the basic features of interface automata by applying them to the modeling of a software component that implements a message-transmission service. The component has a method *msg*, used to send messages. Whenever this method is called, the component returns either *ok* or *fail*. To perform this service, the component relies on an interface to a communication channel that provides the method *send* for sending messages. The two possible return values are *ack*, which indicates a successful transmission, and *nack*, which indicates a failure. When the method *msg* is invoked, the component tries to send the message, and resends it if the first transmission fails. If both transmissions fail, the component reports failure; otherwise, it reports success. The interface automaton *Comp* modeling this component is illustrated in Figure 1(b). The automaton *Comp* is not receptive; its illegal inputs are used to specify assumptions about the environment. For example, the input *msg* is accepted only in state 0. This represents the assumption that, once the method *msg* is called, the environment will wait for an *ok* or *fail* response before issuing another call of *msg*.

Assume that the component *Comp* is used by a user component that expects messages to be successfully sent, and makes no provisions for handling failures. The interface automaton *User* shown in Figure 1(a) models such a component: after calling the method *msg*, it accepts the return value *ok*, but does not accept the return value *fail*. The expectation that the return value is *ok* is an assumption by the component *Comp* about its environment; that is, the component *Comp* is designed to be used only with message-transmission services that cannot fail.

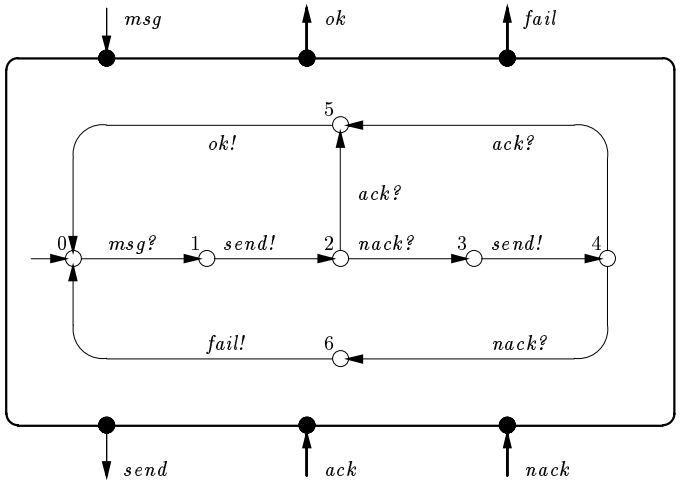
In Figure 1(c) we present the product $User \otimes Comp$ of the two automata *Comp* and *User*. Each state of the product consists of a state of *User* together with a state of *Comp*. Each step of the product is either a joint *msg* step, which represents the call of the method *msg* by *User*, or a joint *ok* step, which represents the termination of the method *msg* with return value *ok*, or a step of *Comp* calling the method *send* of the (unspecified) channel, or a step of *Comp* receiving the return value *ack* or *nack* from the channel. Consider the following sequence of events. The component *User* calls the method *msg*; then *Comp* calls twice the method *send* and receives 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 1 of *User* and state 6 of *Comp*. In state 6, the component *Comp* tries to report failure by returning *fail*, but not expecting failure, the component *User* does not accept the return value *fail* in state 1. We declare the "unexpected" state 6 of the product automaton $User \otimes Comp$ to be *illegal*, because if the environment assumption of the component *Comp* is satisfied by its environment, then this state does not occur.

There are two ways of dealing with illegal states. The standard "pessimistic" approach considers two interfaces incompatible if the product can reach illegal states. In the example, the components *User* and *Comp* would be incompatible, since state 6 is reachable in $User \otimes Comp$, and *User* and *Comp* would be declared incompatible, because *Comp* does not by itself satisfy the environment assumption of *User*, namely, that every call to *msg* returns the value *fail*.

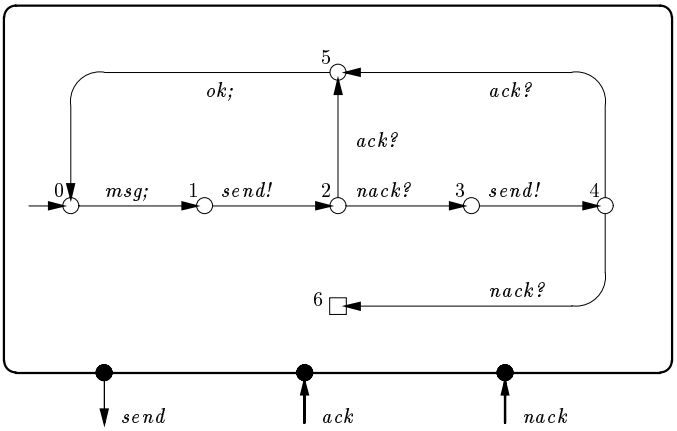
The pessimistic approach is appropriate when the product system is closed. However, the product $User \otimes Comp$ is again an open system, with an environment —the communication channel— which provides *ack* and *nack* values to $User \otimes Comp$. By declaring *User* and *Comp* to be incom-



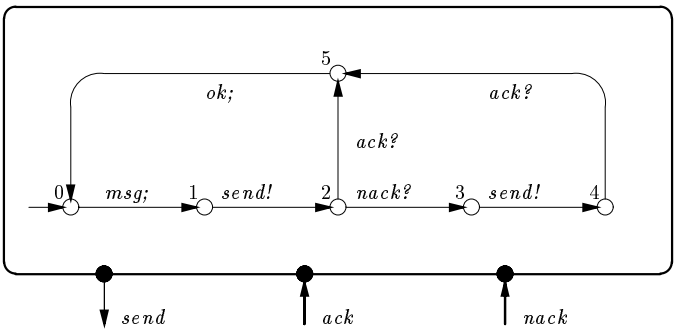
(a) Interface automaton *User*



(b) Interface automaton *Comp*



(c) $User \otimes Comp$. The illegal state of the product is depicted as a square.



(d) $User \| Comp$

Figure 1: Interface automata. We enclose each automaton in a box, whose ports correspond to the input and output actions. We append to the name of the actions the symbol "?" (resp. "!", ";") to denote that the action is an input (resp. output, internal) action. An arrow without source denotes the initial state of the automaton.

patible, the pessimistic approach forecloses on the possibility that the channel is helpful and makes the two components work together. In fact, the pessimistic approach rejects an open system if there is *some* environment under which it behaves incorrectly. In particular, the pessimistic approach fails to propagate the environment assumption of *User* to its composition with *Comp*.

In contrast, according to our optimistic approach *User* and *Comp* are compatible. In fact, we consider two components compatible if there is some environment in which they work correctly: indeed, the environment that provides input *ack* at state 4 ensures that the illegal state 6 is not entered. Such environments, that prevent illegal states from being reached, are called the *legal* environments. The states of the product $User \otimes Comp$ from which the environment can ensure that no illegal state is entered are called the *compatible states*; in the example, they are 0, 1, 2, 3, 4, and 5. The *composition* $User \| Comp$ of the two interface automata is obtained by restricting the product $User \otimes Comp$ to its compatible states, as depicted in Figure 1(d). Note how restricting $User \otimes Comp$ to its compatible states corresponds to imposing an assumption on the environment, namely, that calls to the method *msg* never return twice in a row the value *nack*. Hence, when the two automata *Comp* and *User* are composed, the assumption of *User* that no failures occur is translated into the assumption of $User \| Comp$ that no two consecutive transmissions fail. The indicates how, under the optimistic approach, the composition of the interface automata *User* and *Comp* propagates to the environment of $User \| Comp$ the assumptions that are necessary for the correct interaction of *User* and *Comp*.

This example illustrates the deep difference between the pessimistic and optimistic approaches. While the optimistic approach considers many components compatible which would be incompatible under the pessimistic approach, there are of course components that cannot work together even under the optimistic view, namely those for which no legal environment exists. The compatibility check is performed by computing compatible states, which amounts to solving a game between the product automaton (which tries to enter illegal states) and its environment (which tries to prevent this).

3 Interface Automata

Definition 3.1 *An* interface automaton $P = \langle V_P, V_P^{init}, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, \mathcal{T}_P \rangle$ *consists of the following elements:*

- V_P *is a set of* states.
- $V_P^{init} \subseteq V_P$ *is a set of* initial states. *We require that* V_P^{init} *contains at most one state. If* $V_P^{init} = \emptyset$*, then* P *is called* empty.
- $\mathcal{A}_P^I$, $\mathcal{A}_P^O$, *and* $\mathcal{A}_P^H$ *are mutually disjoint sets of* input, output, *and* internal *actions. We denote by* $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O \cup \mathcal{A}_P^H$ *the set of all* actions.
- $\mathcal{T}_P \subseteq V_P \times \mathcal{A}_P \times V_P$ *is a set of* steps. ∎

If $a \in \mathcal{A}_P^I$ (resp. $a \in \mathcal{A}_P^O$, $a \in \mathcal{A}_P^H$), then (v, a, v') is called an input (resp. output, internal) step. We denote by $\mathcal{T}_P^I$ (resp. $\mathcal{T}_P^O$, $\mathcal{T}_P^H$) the set of input (resp. output, internal) steps. The interface automaton P is *closed* if it has only internal actions, that is, $\mathcal{A}_P^I = \mathcal{A}_P^O = \emptyset$; otherwise, we say that P is *open.* An action $a \in \mathcal{A}_P$ is *enabled* at a state $v \in V_P$ if there is a step $(v, a, v') \in \mathcal{T}_P$ for some $v' \in V_P$. We indicate by $\mathcal{A}_P^I(v)$, $\mathcal{A}_P^O(v)$, $\mathcal{A}_P^H(v)$ the subsets of input, output, and internal actions that are enabled at the state v, and we let $\mathcal{A}_P(v) = \mathcal{A}_P^I(v) \cup \mathcal{A}_P^O(v) \cup \mathcal{A}_P^H(v)$. Unlike I/O automata [9], an interface automaton is not required to be input-enabled; that is, we do not require

that $\mathcal{A}_P^I(v) = \mathcal{A}_P^I$ for all states $v \in V_P$. The set $\mathcal{A}_P^I(v)$ of enabled input actions specifies which inputs are accepted at the state v; we call the inputs in $\mathcal{A}_P^I \setminus \mathcal{A}_P^I(v)$ the *illegal inputs* at v. Also note that an interface automaton is not required to be non-blocking; that is, we do not require that $\mathcal{A}_P(v) \neq \emptyset$ for all states $v \in V_P$. Blocking states can be used to model terminating processes. The *size* of an interface automaton P is defined by $|P| = |V_P| + |\mathcal{T}_P|$.

Example 3.1 *The interface automaton User of Figure 1 has two input actions, ok and fail, one output action, msg, and no internal actions. It has two states, with state 0 being initial, and two steps,* $(0, msg, 1)$ *and* $(1, ok, 0)$. ∎

Definition 3.2 *An* execution fragment *of an interface automaton P is a finite alternating sequence of states and actions $v_0, a_0, v_1, a_1, \ldots, v_n$ such that $(v_i, a_i, v_{i+1}) \in \mathcal{T}_P$ for all $0 \leq i < n$. Given two states $v, u \in V_P$, we say that u is* reachable from v *if there is an execution fragment whose first state is v, and whose last state is in u. The state u is* reachable in P *if there exists an initial state $v \in V_P^{init}$ such that u is reachable from v.* ∎

In the definition of interface automaton, it is not required that all states are reachable. However, one is generally not interested in unreachable states, and they can be removed in linear time.

3.1 Compatibility and composition

We define the composition of two interface automata only if their actions are disjoint, except that an input action of one may coincide with an output action of the other. The two automata will synchronize on such shared actions, and asynchronously interleave all other actions.

Definition 3.3 *Two interface automata P and Q are* composable *if*

$$\mathcal{A}_P^H \cap \mathcal{A}_Q = \emptyset \qquad \mathcal{A}_P^I \cap \mathcal{A}_Q^I = \emptyset$$
$$\mathcal{A}_P^O \cap \mathcal{A}_Q^O = \emptyset \qquad \mathcal{A}_Q^H \cap \mathcal{A}_P = \emptyset.$$

We let $shared(P, Q) = \mathcal{A}_P \cap \mathcal{A}_Q$. ∎

Note that if two interface automata P and Q are composable, then $shared(P, Q) = (\mathcal{A}_P^I \cap \mathcal{A}_Q^O) \cup (\mathcal{A}_P^O \cap \mathcal{A}_Q^I)$. We define the composition of interface automata in stages, defining first the product automaton $P \otimes Q$. This product coincides with the composition of I/O automata [9], except that since P and Q are not necessarily input-enabled, some steps present in P or Q may not be present in the product.

Definition 3.4 *If P and Q are composable interface automata, their* product $P \otimes Q$ *is the interface automaton defined by*

$$V_{P\otimes Q} = V_P \times V_Q$$
$$V_{P\otimes Q}^{init} = V_P^{init} \times V_Q^{init}$$
$$\mathcal{A}_{P\otimes Q}^I = (\mathcal{A}_P^I \cup \mathcal{A}_Q^I) \setminus shared(P, Q)$$
$$\mathcal{A}_{P\otimes Q}^O = (\mathcal{A}_P^O \cup \mathcal{A}_Q^O) \setminus shared(P, Q)$$
$$\mathcal{A}_{P\otimes Q}^H = \mathcal{A}_P^H \cup \mathcal{A}_Q^H \cup shared(P, Q).$$

The set $\mathcal{T}_{P\otimes Q}$ of steps is defined in Figure 2. ∎

$$\begin{aligned}\mathcal{T}_{P\otimes Q} &= \{((v,u),a,(v',u)) \mid (v,a,v') \in \mathcal{T}_P \wedge a \notin \mathit{shared}(P,Q) \wedge u \in V_Q\} \\ &\cup \{((v,u),a,(v,u')) \mid (u,a,u') \in \mathcal{T}_Q \wedge a \notin \mathit{shared}(P,Q) \wedge v \in V_P\} \\ &\cup \{((v,u),a,(v',u')) \mid (v,a,v') \in \mathcal{T}_P \wedge (u,a,u') \in \mathcal{T}_Q \wedge a \in \mathit{shared}(P,Q)\}.\end{aligned}$$

Figure 2: Definition of the steps of the product of two interface automata.

$$\mathit{Illegal}(P,Q) = \left\{ (v,u) \in V_P \times V_Q \;\middle|\; \exists a \in \mathit{shared}(P,Q) \,.\, \left(\begin{array}{c} a \in \mathcal{A}_P^O(v) \wedge a \notin \mathcal{A}_Q^I(u) \\ \vee \\ a \in \mathcal{A}_Q^O(u) \wedge a \notin \mathcal{A}_P^I(v) \end{array} \right) \right\}.$$

Figure 3: Definition of the illegal states of a product automaton.

Example 3.2 *The product User $\otimes$ Comp of the interface automata User and Comp of Figure 1 is shown in Figure 1(c). We have only depicted the reachable states of the product. After each operation involving interface automata, such as product, we routinely remove from the resulting automaton all unreachable states. Note that state 6 has no outgoing edges, because fail $\in$ shared(User, Comp), but the step $(6, \mathit{fail}, 0)$ of Comp has no corresponding step in User.* ∎

Since interface automata are not necessarily input-enabled, in the product $P \otimes Q$ of two interface automata P and Q, one of the automata may produce an output action that is an input action of the other automaton, but is not accepted. The set $\mathit{Illegal}(P,Q)$ of states of $P \otimes Q$ where this happens are called the *illegal states* of the product.

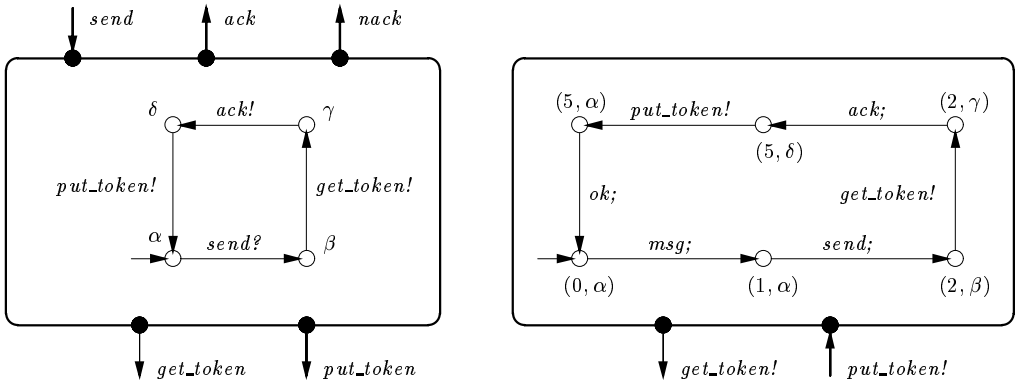
Definition 3.5 *Given two composable interface automata P and Q, the set $\mathit{Illegal}(P,Q) \subseteq V_P \times V_Q$ of* illegal states *of $P \otimes Q$ is defined in Figure 3.* ∎

Example 3.3 *Referring again to the product User $\otimes$ Comp shown in Figure 1, we have $6 \in \mathit{Illegal}(\mathit{User}, \mathit{Comp})$, because the output step $(6, \mathit{fail}, 0)$ of Comp has no corresponding input step in User.* ∎

When the product $P \otimes Q$ is closed, we say that P and Q are *compatible* if no illegal state of $P \otimes Q$ is reachable. When $P \otimes Q$ is open, however, the fact that a state in $\mathit{Illegal}(P,Q)$ is reachable does not necessarily indicate an incompatibility, because by generating appropriate inputs, the environment of $P \otimes Q$ may be able to ensure that the set $\mathit{Illegal}(P,Q)$ is not entered. Such an environment, which steers away from the illegal states, is called a *legal environment.* The existence of a legal environment indicates that there is a way to use the interfaces P and Q together without giving rise to incompatibilities. A legal environment for $R = P \otimes Q$ needs to satisfy the following side conditions.

Definition 3.6 *An* environment *for an interface automaton R is an interface automaton E such that (1) E is composable with R, (2) E is nonempty, (3) $\mathcal{A}_E^I = \mathcal{A}_R^O$, and (4) $\mathit{Illegal}(R,E) = \emptyset$.* ∎

The second condition ensures that the environment does not constrain the reachable states of R by having no initial state. The third and fourth conditions ensure that the environment does not constrain R by not accepting some of its output steps, and that the environment generates only inputs to R that can be accepted by R.



(a) *Channel* (b) $User \otimes Comp \otimes Channel$

Figure 4: A legal environment *Channel* for (*User*, *Comp*), and the product $User \otimes Comp \otimes Channel$.

Definition 3.7 *Given two composable interface automata P and Q, a* legal environment *for the pair (P, Q) is an environment for $P \otimes Q$ such that no state in $Illegal(P, Q) \times V_E$ is reachable in $(P \otimes Q) \otimes E$.* ∎

Example 3.4 *The interface automaton Channel shown in Figure 4(a) is a legal environment for (User, Comp), because in the product (User ⊗ Comp) ⊗ Channel (see Figure 4(b)), the state $(6, u)$ is not reachable for any $u \in \{\alpha, \beta, \gamma, \delta\}$.* ∎

We define compatibility as the existence of a legal environment.

Definition 3.8 *Two interface automata P and Q are* compatible *if they are nonempty, composable, and there exists a legal environment for (P, Q).* ∎

Example 3.5 *The two interface automata User (Figure 1(a)) and Comp (Figure 1(b)) are compatible, because the automaton Channel (Figure 4(a)) is a legal environment for $(User, E)$.* ∎

The composition of two interface automata is obtained by restricting the product of the two automata to the set of *compatible states,* which are the states from which the environment can prevent entering illegal states.

Definition 3.9 *Consider two composable interface automata P and Q. A pair $(v, u) \in V_P \times V_Q$ of states is* compatible *if there is an environment E for $P \otimes Q$ such that no state in $Illegal(P, Q) \times V_E$ is reachable in $(P \otimes Q) \otimes E$ from the state $\{(v, u)\} \times V_E^{init}$. We write $Cmp(P, Q)$ for the set of compatible states of $P \otimes Q$.* ∎

Hence, we can rephrase the definition of compatibility for interface automata as follows: two nonempty, composable interface automata P and Q are compatible iff their initial states are compatible, i.e., if $V_P^{init} \times V_Q^{init} \subseteq Cmp(P, Q)$.

Definition 3.10 *Consider two composable interface automata P and Q. The* composition $P \| Q$ *is an interface automaton with the same action sets as $P \otimes Q$. The states are $V_{P\|Q} = Cmp(P, Q)$, the initial states are $V_{P\|Q}^{init} = V_{P\otimes Q}^{init} \cap Cmp(P, Q)$, and the steps are $\mathcal{T}_{P\|Q} = \mathcal{T}_{P\otimes Q} \cap (Cmp(P, Q) \times \mathcal{A}_{P\|Q} \times Cmp(P, Q))$.* ∎

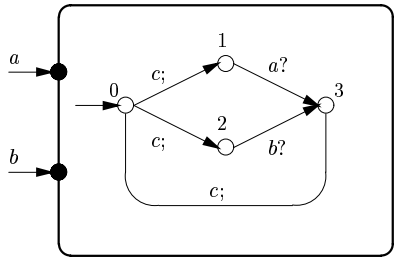


Figure 5: Interface automaton R. State 3 is not reachable after composition with any environment for R.

Example 3.6 *In the automaton User⊗Comp (Figure 1(c)), the states 0, 1, 2, 3, 4, and 5 are compatible. The result of restricting User⊗Comp to its compatible states is the automaton User‖Comp, depicted in Figure 1(d). In general, the restriction to compatible states may render some states unreachable. They can then be removed from the composite automaton.* ■

Recalling that two interface automata are compatible if their initial states are compatible, the definition of composition yields the following alternative characterization of compatibility:

> *Two interface automata P and Q are* compatible *iff (a) they are composable and (b) their composition is nonempty.*

This criterion will be used in our algorithmic check of compatibility, based on the computation of automaton composition. The composition of interface automata is associative.

Theorem 3.1 *For all interface automata P, Q, and R, either both $(P\|Q)\|R$ and $P\|(R\|Q)$ are undefined, because some of the automata are not composable, or $(P\|Q)\|R = P\|(R\|Q)$.*

3.2 Discussion

An interface automaton represents both assumptions about the environment, and guarantees about the specified component. The environment assumptions are twofold: each output step incorporates the assumption that the corresponding action is accepted by the environment as input; and 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 sequences and choices of input, output, and internal 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.

One interesting note about interface automata is that while some states may be reachable in an interface automaton R, they cannot be reached in any composition of R with an environment E. This may happen, because the environment cannot observe the state of R, only its input and output actions. Hence, in order to satisfy condition (4) of the definition of environment, namely, that $R \otimes E$ contains no illegal states, the environment must be conservative and provide an input to R only if R accepts that input in all states in which it could possibly be. Because of this, there may be reachable states v of R such that for *all* environments E of R, no state of the form $(v, \cdot)$ is reachable in the product $R \otimes E$. These states can, of course, be removed from R in order to make R smaller. However, the best known algorithm for finding these states requires exponential time (and polynomial space), and relies on an adaptation of the subset construction of [11].

Example 3.7 *Consider the interface automaton R of Figure 5. State 3 of the automaton is not reachable after composition with any environment for R. An environment for R can provide neither input a, nor input b, to R. The reason is that, because of the internal steps from state 0 to states 1 and 2, the environment can never be sure that R is ready to accept these inputs.* ∎

Whenever two interface automata P and Q are compatible, there is a particularly simple legal environment for (P, Q), namely, the one that accepts all outputs of $P \otimes Q$, and that generates no inputs for $P \otimes Q$: clearly, this environment avoids entering $\mathit{Illegal}(P, Q)$ whenever possible. 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 automaton that produces no inputs is the best for showing compatibility. There are several ways of enabling interface automata to specify inputs that must occur: among them, synchronicity, adding fairness, or adding real-time constraints. Such extensions are beyond the scope of this work.

3.3 Computing the composition

Given two interface automata P and Q, a pair $(v, u) \in V_P \times V_Q$ is compatible if there is some environment under which $\mathit{Illegal}(P, Q)$ is not reachable in $P \otimes Q$ from (v, u). As remarked above, the best environment corresponds to accepting all outputs of $P \otimes Q$, and generating no inputs for $P \otimes Q$. On the basis of this observation, the set $\mathit{Cmp}(P, Q)$ can be computed by performing a backward reachability analysis from $\mathit{Illegal}(P, Q)$ which traverses only internal and output steps, and removes all states thus reachable. To present the algorithm, we introduce the operator OHpre. Intuitively, for a set U of states of an interface automaton R, the set $\mathit{OHpre}_R(U)$ contains the states of R that can enter a state in U by taking an internal or output step. Formally, the operator $\mathit{OHpre}_R : 2^{V_R} \mapsto 2^{V_R}$ is defined for all sets $U \subseteq V_R$ by

$$\mathit{OHpre}_R(U) = \{v \in V_R \mid \exists (v, a, u) \in \mathcal{T}_R^O \cup \mathcal{T}_R^H \,.\, u \in U\}.$$

The set $\mathit{Cmp}(P, Q)$ can be computed by iterating the operator $\mathit{OHpre}_{P \otimes Q}$, starting from $\mathit{Illegal}(P, Q)$ until no new states are found.

Algorithm 1
Input: *Interface automata P and Q.*
Output: $\mathit{Cmp}(P, Q)$.

> **Initialization:** *Let* $U_0 = \mathit{Illegal}(P, Q)$.
> **Repeat:** *For* $k \geq 0$, *let* $U_{k+1} = U_k \cup \mathit{OHpre}_{P \otimes Q}(U_k)$.
> **Until:** $U_{k+1} = U_k$.
> **Return:** $V_{P \otimes Q} \setminus U_k$. ∎

Using this algorithm, the composition of two interface automata can be computed by first pruning the incompatible states from the product, and then, for optimization, removing any unreachable states. The following theorem summarizes the complexity of deciding compatibility, and computing the composition, of interface automata.

Theorem 3.2 *Given two interface automata P and Q, we can decide whether they are compatible, and we can compute $P \| Q$ in time linear in $|P|$ and $|Q|$.*

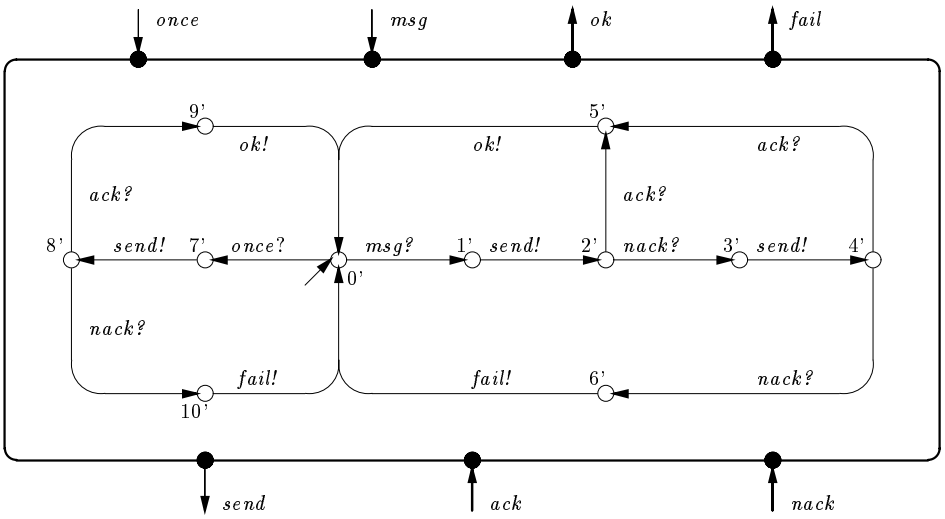


Figure 6: Interface automaton *QuickComp*, offering retry-once and retry-twice message transmission.

Since the composition of interface automata is associative, we can check whether $n > 0$ interface automata $P_1, \ldots, P_n$ are compatible by computing their composition $P_1 \| \cdots \| P_n$ in a gradual fashion, constructing $(P_1 \| \cdots \| P_{i-1}) \| P_i$ for $i = 1, 2, \ldots, n$, and checking for each i that the resulting composition is nonempty. The efficiency of this check lies in the fact that, in the computation of $(P_1 \| \cdots \| P_{i-1}) \| P_i$, the incompatible states are pruned as early as possible. This idea is closely related to the use of games for the early detection of errors in verification [3]. We can further improve the algorithm by composing the automata $P_1, \ldots, P_n$ in a tree-like fashion, rather than in a linear order.

4 Refinement

The *refinement relation* aims at formalizing the relation between abstract and concrete versions of the same component, for example, between an interface specification and its implementation. In the input-enabled setting, refinement is usually defined as trace containment or simulation [10]; this ensures that the output behaviors of the implementation are behaviors that are allowed by the specification. However, such definitions are not appropriate in a non-input-enabled setting, such as interface automata: if one requires that also the set of legal inputs of the implementation is a subset of the inputs allowed by the specification, then the implementation could be used in fewer environments than the interface specification. To illustrate the shortcomings of the standard definition, consider the interface automaton *QuickComp* of Figure 6. This automaton represents a component that provides two services: the first is the try-twice service *msg* provided also by the automaton *Comp* of Figure 1(b); 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 *QuickComp* is a refinement of *Comp*, because *QuickComp* implements all services provided by *Comp*, and is consistent with *Comp* in their implementation. The language of *QuickComp*, however, is not contained in the language of *Comp*; indeed, *once* is not even an action of *Comp*. Instead, we must define refinement in a contravariant fashion: the implementation must allow more legal inputs, and fewer outputs, than the specification.

We choose a contravariant refinement relation in the spirit of simulation, rather than language

containment. This leads to the definition of refinement as *alternating simulation* [2]. Roughly, an interface automaton P refines an interface automaton Q if all input steps of Q can be simulated by P, and all the output steps of P can be simulated by Q. The precise definition must take into account the fact that the internal steps of P and Q are independent. For this, we need some preliminary notions. The ε-closure of a state v consists of the set of states that can be reached from v by taking only internal steps.

Definition 4.1 *Given an interface automaton P and a state $v \in V_P$, the set ε-closure$_P(v)$ is the smallest set $U \subseteq V_P$ such that (1) $v \in U$ and (2) if $u \in U$ and $(u, a, u') \in \mathcal{T}_P^H$, then $u' \in U$.* ∎

The environment of an interface automaton P cannot see the internal steps of P. Consequently, if P is at a state v, then the environment cannot distinguish between v and any state in ε-closure$_P(v)$. In particular, the environment must be able to accept all output actions in $ExtEn_P^O(v)$, because P can issue these outputs without any forewarning to the environment. Conversely, the environment can safely issue an action a as input to P only if a is accepted at all states in ε-closure$_P(v)$, because P could have transitioned to any of these states, unbeknownst to the environment. This motivates the following definition.

Definition 4.2 *Consider an interface automaton P, and a state $v \in V_P$. We let*

$$\begin{aligned} ExtEn_P^O(v) &= \{a \mid \exists u \in \varepsilon\text{-closure}_P(v).\ a \in \mathcal{A}_P^O(u)\} \\ ExtEn_P^I(v) &= \{a \mid \forall u \in \varepsilon\text{-closure}_P(v).\ a \in \mathcal{A}_P^I(u)\} \end{aligned}$$

be the sets of externally enabled *output and input actions, respectively, at v.* ∎

The following definition introduces an abbreviation for the set of states that are reachable by taking externally enabled actions.

Definition 4.3 *Consider an interface automaton P and a state $v \in V_P$. For all externally enabled input and output actions $a \in ExtEn_P^I(v) \cup ExtEn_P^O(v)$, we let*

$$ExtDest_P(v, a) = \{u' \mid \exists (u, a, u') \in \mathcal{T}_P.\ u \in \varepsilon\text{-closure}_P(v)\}.$$ ∎

Using this notation, we are ready to define alternating simulation on interface automata.

Definition 4.4 *Consider two interface automata P and Q. A binary relation $\succeq \subseteq V_P \times V_Q$ is an* alternating simulation from Q to P *if for all states $v \in V_P$ and $u \in V_Q$ such that $v \succeq u$, the following conditions hold:*

1. *$ExtEn_P^I(v) \subseteq ExtEn_Q^I(u)$ and $ExtEn_Q^O(u) \subseteq ExtEn_P^O(v)$.*
2. *For all actions $a \in ExtEn_P^I(v) \cup ExtEn_Q^O(u)$ and all states $u' \in ExtDest_Q(u, a)$, there is a state $v' \in ExtDest_P(v, a)$ such that $v' \succeq u'$.* ∎

Condition 1 expresses the input-output duality between states $v \succeq u$ in the alternating simulation relation: all externally enabled inputs of v are also externally enabled in u, and conversely, all externally enabled outputs of u are externally enabled in v. Condition 2 recursively propagates the simulation relation: all steps from u that correspond to externally enabled actions can be matched by steps from v. This definition of alternating simulation is more involved than the one of [2] because it deals with internal steps. In the example of Figures 1(b) and 6, there is an alternating simulation that relates i with i', for $i \in \{0, 1, 2, 3, 4, 5, 6\}$.

Definition 4.5 *The interface automaton Q* refines *the interface automaton P, written $P \succeq Q$, if (1) $\mathcal{A}_P^I \subseteq \mathcal{A}_Q^I$ and $\mathcal{A}_P^O \supseteq \mathcal{A}_Q^O$, and (2) there is an alternating simulation $\succeq$ from Q to P, a state $v \in V_P^{init}$, and a state $u \in V_Q^{init}$ such that $v \succeq u$.* ∎

Note that unlike in standard simulation, the "typing" condition (1) is contravariant on the action sets. The definition of refinement captures a simple kind of sub-classing: if $P \succeq Q$, then the implementation Q is able to provide more services than the specification P, but it must be consistent with P on the shared services. Refinement between interface automata is a preorder (i.e., reflexive and transitive).

Theorem 4.1 *For all interface automata P, Q, and R, we have $P \succeq P$, and if $P \succeq Q$ and $Q \succeq R$, then $P \succeq R$.*

The following theorem states two important properties of refinement between interface automata. First, refinement and compatibility are related as follows: we can always replace a component P with a more refined version Q such that $P \succeq Q$, provided that Q and P 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 Q that are not provided for P, then it would be possible that new incompatibilities arise in the processing of these inputs. For software components, this property is a statement of sub-class polymorphism: we can always substitute a sub-class for a super-class, provided no new methods of the sub-class are used. In general, this property captures the essence of *component-based design*: the designer of the environment (i.e., the other system components) needs to ensure only compatibility with the component specification P, and in this way guarantees compatibility with the component implementation Q. Second, refinement is *compositional*: in order to check if $P \| P' \succeq Q \| Q'$ it suffices to check both $P \succeq Q$ and $P' \succeq Q'$. Since the latter checks involve smaller automata, they are more efficient.

Theorem 4.2 *Consider three interface automata P, Q, and R such that Q and R are composable, and $\mathcal{A}_Q^I \cap \mathcal{A}_R^O \subseteq \mathcal{A}_P^I \cap \mathcal{A}_R^O$. If P and R are compatible and $P \succeq Q$, then Q and R are compatible and $P \| R \succeq Q \| R$.*

The set of alternating simulations between two interface automata is closed under union. Hence, there is a unique alternating simulation from Q to P that is maximal in the partial order induced by set inclusion. To check whether $P \succeq Q$, it thus suffices to compute this maximal alternating simulation, and check if it relates the initial states of P and Q. The maximal alternating simulation can be computed by starting from the relation $V_P \times V_Q$ and iteratively removing pairs, until either the computed relation satisfies the conditions of an alternating simulation, or no pairs are left, in which case there is no alternating simulation. This procedure is analogous to the one presented in [2] for alternating simulation relations between game structures.

Algorithm 2
Input: *Interface automata P and Q.*
Output: *The unique maximal alternating simulation from Q to P.*

> **Initialization:** *Let $\succeq_0 = V_P \times V_Q$.*
> **Repeat:** *For $k \geq 0$, define $\succeq_{k+1} \subseteq \succeq_k$ by $v \succeq_{k+1} u$ if $v \succeq_k u$ and conditions 1 and 2 of Definition 4.4 are satisfied by v and u, with $\succeq$ replaced by $\succeq_k$.*
> **Until:** $\succeq_{k+1} = \succeq_k$.
> **Return:** $\succeq_k$. ∎

From the hardness results on ordinary simulation [6] we have the following theorem, analogous to a result of [2].

Theorem 4.3 *Checking refinement between interface automata is PTIME-complete.*

5 Single-Threaded Interface Automata

When there is only one active thread of control in a system, we can take advantage of this fact by providing specialized definitions of interface automata and composition. These *single-threaded* versions of interface automata give rise to smaller automata for composite systems.

Definition 5.1 *A* single-threaded interface automaton *(STIA) P is an interface automaton that satisfies the following conditions:*

1. *The set V_P of states is partitioned into two disjoint sets $V_P = V_P^O \cup V_P^I$. The states in V_P^O are called* running, *because only internal and output actions are enabled: for all $v \in V_P^O$, we have $\mathcal{A}_P^I(v) = \emptyset$. The states in V_P^I are called* waiting, *because only input actions are enabled: for all $v \in V_P^I$, we have $\mathcal{A}_P^O(v) = \mathcal{A}_P^H(v) = \emptyset$.*

2. *All output steps must lead to waiting states: for all $(u, a, v) \in \mathcal{T}_P^O$, we have $v \in V_P^I$. Conversely, only output steps can lead to waiting states: for all $v \in V_P^I$ and all $(u, a, v) \in \mathcal{T}_P$, we have $a \in \mathcal{A}_P^O$.* ∎

Condition 1 rules out states where there is a choice between output/internal actions (which are caused by the automaton advancing a thread of control) and input actions (which are caused by some other automaton advancing a thread of control). The running states indicate ownership of the single thread of control; the waiting states indicate non-ownership. Condition 2 ensures that an STIA waits for an input precisely after issuing an output action; this is because if there is a single thread of control, then each output step relinquishes that thread.

Definition 5.2 *Two STIAs P and Q are* composable *if (1) they are composable when considered as interface automata, and (2) at most one of the initial states is running; that is, $V_P^{init} \subseteq V_P^I$ or $V_Q^{init} \subseteq V_Q^I$. Two STIAs are* compatible *if they are composable as STIAs, and compatible when considered as interface automata.* ∎

In the definition of single-threaded compatibility, we do not need a restriction to single-threaded environments, because the "optimal" environment, which issues no outputs, has only waiting states, and is therefore single-threaded. Together, these definitions ensure that in every reachable state of a composition of STIAs, at most one of the components is in a running state. The composition $P \| Q$ of two compatible STIAs P and Q is not necessarily an STIA. This is because $P \| Q$ may contain reachable states v where both input and output actions are enabled. However, these input actions are never taken if $P \| Q$ is composed only with other STIAs: since one of P or Q is running at v, no other STIA can be running and cause the input actions to be taken. The theorem below makes this statement precise.

Theorem 5.1 *Consider two compatible STIAs P and Q, a state $v \in V_{P\|Q}$, and two actions $a, b \in \mathcal{A}_{P\|Q}(v)$ with $a \in \mathcal{A}_{P\|Q}^I$ and $b \in \mathcal{A}_{P\|Q}^H \cup \mathcal{A}_{P\|Q}^O$. If R is an STIA compatible with $P \| Q$ such that $(P \| Q) \| R$ is closed, then for all states $u \in V_R$ we have $a \notin \mathcal{A}_{(P\|Q)\|R}(v, u)$.*

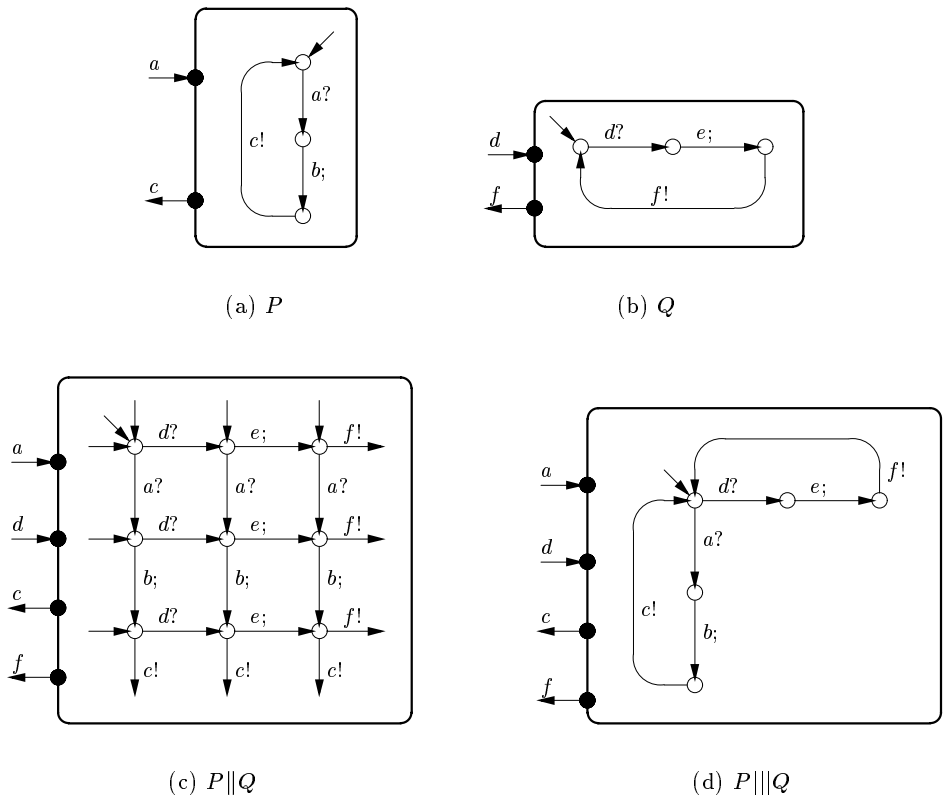


Figure 7: Single-threaded interface automata P and Q, their composition $P\|Q$, and their single-threaded composition $P|||Q$. The steps of the composition $P\|Q$ are connected on a toroidal topology.

Hence we introduce a special version of composition for STIAs, called *single-threaded composition,* which prunes the input actions that occur at states where internal or output actions are also enabled.

Definition 5.3 *Consider two composable STIAs P and Q. The* single-threaded composition $P|||Q$ *is obtained from $P\|Q$ by first removing all steps $(v, a, u) \in \mathcal{T}^I_{P\|Q}$ for which $\mathcal{A}^O_{P\|Q}(v) \cup \mathcal{A}^H_{P\|Q}(v) \neq \emptyset$, and then removing all states that become unreachable from $V^{init}_{P\|Q}$.* ∎

Single-threaded composition is again associative. In Figure 7 we illustrate that single-threaded composition may lead to a substantial reduction in the size of the state space. By performing the single-theaded pruning on-the-fly during the product construction, it is possible to construct $P|||Q$ without constructing the entire reachable state space of $P\|Q$. The following theorem indicates that the single-threaded composition of STIAs yields STIAs, and that ordinary and single-threaded composition give rise to the same notion of compatibility.

Theorem 5.2 *For composable STIAs P and Q, the single-threaded composition $P|||Q$ is an STIA. Furthermore, if $P\|Q$ is nonempty, then $P|||Q$ is nonempty.*

Acknowledgments. We thank Edward A. Lee, Xiaojun Liu, Freddy Mang, and Yuhong Xiong for fruitful discussions.

References

[1] R. Allen and D. Garland. Formalizing architectural connection. In *Proc. 16th IEEE Conf. Software Engineering*, pages 71–80, 1994.

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

[3] L. de Alfaro, T. Henzinger, and F. Mang. Detecting errors before reaching them. In *Computer-Aided Verification*, Lecture Notes in Computer Science 1855, pages 186–201. Springer-Verlag, 2000.

[4] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[5] D. Jackson. Enforcing design constraints with object logic. In *Static Analysis Sumposium*, Lecture Notes in Computer Science 1824, pages 1–21. Springer-Verlag, 2000.

[6] O. Kupferman and M. Vardi. Verification of fair transition systems. *Chicago J. Theoretical Computer Science*, 2, 1998.

[7] E. Lee and Y. Xiong. *System-level Types for Component-based Design.* Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley, 2000.

[8] N. Leveson. *System Safety and Computers.* Addison-Wesley, 1995.

[9] N. Lynch and M. Tuttle. Hierarcical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. Principles of Distributed Computing*, pages 137–151, 1987.

[10] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.

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

[12] J. Rumbaugh, G. Booch, and I. Jacobson. *The UML Reference Guide.* Addison-Wesley, 1999.

[13] D. Yellin and R. Strom. Protocol specifications and component adapters. *ACM Trans. Programming Languages and Systems*, 19:292–333, 1997.