Replace this file with `prentcsmacro.sty` for your meeting, or with `entcsmacro.sty` for your meeting. Both can be found at the ENTCS Macro Home Page.

Interfaces: A Game-Theoretic Framework for Reasoning About Component-Based Systems

Luca de Alfaro and Mariëlle Stoelinga [1,2]

Department of Computer Engineering, UC Santa Cruz, USA

Abstract

Traditional type systems specify interfaces in terms of values and domains. When we apply a function to an argument, or when we compose two functions, we have to check that their types match. Interface models extend type systems with the ability to reason about the dynamic behavior of design components. For instance, interface models are able to capture temporal-ordering constraints on component interaction, such as constraints on the ordering of method calls or protocol messages, timing constraints on a component's input and output signals, and constraints on the usage of shared resources. Like type systems, interfaces specify both the input assumptions a component makes on its environment and the output guarantees it provides.

Interfaces are based on two-player games in which the system plays against the environment. The moves of the environment (player Input) represent the inputs that the system can receive from the environment, that is, the input assumption of the system. Symmetrically, the moves of the system (player Output) represent the possible outputs that can be generated by the system. Interfaces are built around the concepts of (1) well-formedness, requiring that the input assumptions of an interface be satisfiable; (2) compatibility, asking whether two components can be used in a way that satisfies the input assumptions of both components; (3) composition of compatible interfaces; and (4) refinement, asking whether one component (being an implementation) correctly implements another one (being the specification).

This paper provides a tutorial-style introduction to interfaces and discusses the basic concepts and ideas. In particular, we elaborate on the automaton-based interfaces from [dAH01a] and the timed interfaces from [dAHS02]. Due space limitations, we do not treat the notion of interface refinement, but we refer the reader to [dAH01a] and [dA03a].

Key words: Component-based design, behavioral type system, game theory.

1 Introduction

The prevalent trend in software and system engineering is towards *component-based design:* systems are designed by combining components, some of them off-the-shelf, other application-specific. The appeal of component-based design is twofold: it helps to tame complexity through decomposition, and it facilitates reuse. Components offer the unit in which complex design problems can be decomposed, allowing the reduction of a single complex design problem into smaller design problems, more manageable in complexity, that can be solved in parallel by design teams. Components also provide a unit of design *reuse,* defining the boundaries in which functionality can be packaged, documented, and reused.

Components are designed to work as parts of larger systems: they make assumptions on their environment, and they expect that these assumptions will be met in the actual environment. For instance, a software component may require its objects to be initialized before any other methods are called. Hence, the effective reuse of software requires adequate documentation of the components' behavior and the conditions under which it can be used, along with methods for checking that components are assembled in an appropriate way.

We describe here a formal notion of component interfaces that provides a framework for the specification and analysis of component interaction. The interface models we describe are able to capture dynamic aspects of component interaction, and are in many respects similar to type systems: indeed, they could be termed a "behavioral" type system for component interaction. In previous work, we have introduced interface theories for various aspects of interaction: [dAH01a,CdAH$^+$02,CdAHM02] consider the temporal order in which method calls (or messages) occur, [dAHS02] reasons about timing constraints on a component's input and output signals, [CdAHS03] deals with constraints on the resource usage of the component and [dAH01b] presents a general theory of interfaces. This tutorial focuses on two of these models: *interface automata* [dAH01a] and *timed interfaces* [dAHS02], and presents the underlying ideas first in a simple, untimed setting, subsequently extending them to deal with real-time input and output specifications.

Interfaces support component-based design in the following ways.

Interface specification. An interface specifies how a component interacts with its environment. It describes the input assumptions the component makes on the environment and the output guarantees it provides. A simple example of an interface is a type in a programming language. The type `int` $\rightarrow$ `real` specifies that a function expects integers as input (input assumption) and

[1] This research was supported in part by the NSF CAREER award CCR-0132780, the NSF grant CCR-0234690, and the ONR grant N00014-02-1-0671.
[2] Email: {`luca,marielle`}`@soe.ucsc.edu`

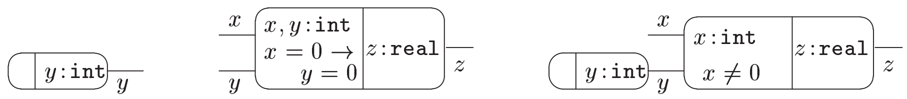
produces reals (output guarantee). A slightly more complicated type is given in Figure 1(b), where a component produces a real z and expects two integers x and y such that $y = 0$ whenever $x = 0$. These types are two examples of static interfaces, i.e. they do not change during the execution of the program. An example of a dynamic type, where the input assumptions and output guarantees can vary with the state of the system, is given in Figure 2(a). This interface automaton models the interface of a 2-place buffer, where state b_i represents the buffer containing i messages. In each state, the automaton models the inputs the component can receive (input assumption) and the outputs it can produce (output guarantee). In particular, the input action *snd?*[3] is not enabled in state b_2, modeling that the buffer cannot receive any messages when it is full. Similarly, the buffer does not produce an output *rec!* in state b_0, modeling that it does not create messages out of the blue if it is empty. Thus, the buffer requires its environment not to send a message while it is in state b_2 and guarantees that it will not produce one in state b_0.

Well-formedness checking. When constructing an interface, we have to make sure that it is *well-formed*, i.e. that there exists at least one environment that satisfies its input assumptions. Otherwise, the interface is useless, since it cannot be used in any design. While rather straightforward in the untimed case, well-formedness becomes more complicated in the timed case, where time progress requirements have to be taken into account.

Interface Composition. Due to the presence of input assumptions, we have to check for *compatibility* when we assemble a system from two (or more[4]) components. That is, when we put together two components P and R, we have to make sure that P's output guarantees imply R's input assumptions and vice versa. Since the composition of two components is generally still an open component, it depends on the environment (of the composite system) whether or not these input assumptions are met. This phenomenon is known as *migration of constraints*: constraints migrate from the components to the composite system. We are interested in the most liberal assumptions on the composite system that ensures compatibility of the components. For example, consider again the interface P_1 in Figure 1(b), producing a real z and expecting two integers x, y with $y = 0$ whenever $x = 0$. Now, we compose P_1 with component P_2 in Figure 1(a). The latter has no inputs (hence, no input assumptions) and can output any integer. To ensure that P_1's input assumption $x = 0 \implies y = 0$ is met, we require that the input x is never

[3] Here and in subsequent examples, the marks ? and ! are appended to an action to indicate whether it is input or output, respectively, but they are not part of the action name itself.

[4] Since composition is commutative and associative, multi-component composition can be obtained via binary composition by successively composing a single component with a system that was previously composed from other components.



(a) P_2. (b) P_1. (c) $P_1 \| P_2$.

Fig. 1. Migration of constraints

set to 0.[5] Since $x \neq 0$ is the weakest predicate with this property, the input assumption of the composition $P_1 \| P_2$ is exactly $x \neq 0$. Its output guarantee is y : `int` and z : `real`, see Figure 1(c).

Summarizing, the composition of two interfaces yields a new interface for the composite system. The input assumptions of the new interface guarantee that the input assumptions of the composed interfaces are met; the output guarantees of the new interface combine the output guarantees of the composed interfaces. The compatibility and composition of interface automata will be explained later in this paper.

Compatibility checking. If the input assumption of the composite system $P \| R$ is equivalent to *false*, i.e. $P \| R$ is not well-formed, then no environment can make P and R work together. In this case P and R are called *incompatible*. In other words, P and R are compatible if and only if there is at least one environment that makes P and R mutually satisfy their input assumptions.

Interfaces as games. An interface is naturally modeled as a game between the players Output and Input. Output represents the component: the moves of Output represent the possible outputs generated by the component (output guarantees). Input represents the environment: the moves of Input represent the inputs accepted from the environment (input assumptions).

Then, an interface is *well-formed* if the Input player has a winning strategy in the game, i.e., the environment can meet all input assumptions. For timed interfaces, we need the additional well-formedness condition that a player must not achieve its goal by blocking time forever. When two interfaces are composed, the combined interface may contain *locally incompatible states.* These occur when one component interface can generate an output that violates an input assumption of the second. Two interfaces are *compatible* if there is a way for the Input player, who chooses the inputs of the composite interface, to avoid all local incompatibilities. Interface compatibility is equivalent to the existence of an environment for the combined interfaces that ensures that the input assumptions of both individual interfaces are satisfied. Component composition thus consists in synthesizing the most liberal input strategy in the composite system that avoid all locally incompatible states. This can be

[5] If $x = 0$, then P_1's assumptions may be met, in case P_2 happens to provide a non-zero integer, but is not guaranteed to be met, as P_2 can set $y = 0$. To ensure satisfaction of the input assumption *for all* behaviors of P_2, we need $x \neq 0$.

done by classical game-theoretic algorithms.

Consider the interface P_1 in Figure 1(b). The Input player chooses values for x and y and the Output player for z. The interface is clearly well-formed, because Input can choose values that meet the input assumptions. When we compose P_1 with P_2, every state with $x = 0$, $y \neq 0$ is a locally incompatible state. The Input player of the composite system (who chooses values for x) has a strategy that, irrespective of the Output strategy (in the composite system choosing values for y and z), avoids these states: namely, the strategy that chooses any integer different from 0.

Related work. Models that can encode input assumptions, such as process algebras, usually phrase the compatibility question as a *graph*, whereas we treat it as a *game* question. In a graph model, input and output play the same role and two components are considered compatible if they cannot reach a deadlock [RR01,CRR02,RR02] In our game-based approach, input and output play dual roles. Two components are compatible if there is *some* input behavior such that, for *all* output behaviors, no incompatibility arises. This notion captures the idea that an interface can be useful as long as it can be used in some design. In this respect, interfaces are close to types in programming languages, to trace theory [Dil88], and to game semantics [Abr96,Abr97,AGN97,AHM98]. The reader is referred to [dA03b] for a more elaborate comparison with related work.

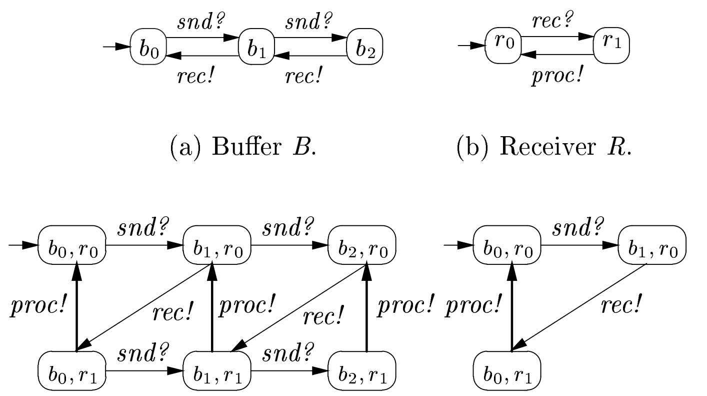
Organization of the paper. This paper treats two automaton-based formalisms for the specification and analysis of interfaces. Section 2 presents interface automata and defines well-formedness, compatibility and composition for these interfaces. In Section 3, we extend interface automata with real-time, yielding timed interface automata. Again, we explore the notions of well-formedness, compatibility and composition. In particular, we explain how timed interfaces deal with time progress conditions, which are needed to ensure that time can advance in every system behavior.

2 Interface Automata

This section presents an automaton-based interface theory that is capable of expressing assumptions and guarantees on the order in which method calls or signals to the component occur [dAH01a]. As one can see from the example in Figure 2(a), interface automata are similar to normal automata (a.k.a. labeled transition systems or state machines); it is in the notion of composition that interfaces differ from ordinary state machines.

Definition 1 An *interface automaton* $P = \langle S_P, S_P^{init}, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{T}_P \rangle$ consists of the following elements.

- S_P is a set of *states*.



(a) Buffer B. (b) Receiver R.

(c) The product $B \otimes R$. (d) The composition $B \| R$.

Fig. 2. Product and composition of interface automata

- $S_P^{init} \subseteq S_P$ is a set of *initial states.*
- $\mathcal{A}_P^I$ and $\mathcal{A}_P^O$ are disjoint sets of *input* and *output* actions. We denote by $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O$ the set of all *actions.*
- $\mathcal{T}_P \subseteq S_P \times \mathcal{A}_P \times S_P$ is a set of *transitions* or *steps.* We write $s \xrightarrow{a}_P t$ for $(s, a, t) \in \mathcal{T}_P$. If $s \xrightarrow{a}_P t$ for some $t \in S_P$, then we say that action a is *enabled* in state s.

We require that P is deterministic[6], that is, (1) S_P^{init} contains at most one state and (2) if $s \xrightarrow{a}_P t$ and $s \xrightarrow{a}_P u$ then $u = t$. ∎

For $s \in S_P$, we let

$$\Gamma_P^I(s) = \{a \in \mathcal{A}_P^I \mid \exists t \in S_P . s \xrightarrow{a}_P t\} \text{ and } \Gamma_P^O(s) = \{a \in \mathcal{A}_P^O \mid \exists t \in S_P . s \xrightarrow{a}_P t\}$$

be respectively the sets of Input and Output *moves* at s. These sets s express the input assumptions and output guarantees of P: at state s, only the actions in $\Gamma_P^I(s)$ are accepted as inputs. In particular, no input in $\mathcal{A}_P^I \setminus \Gamma_P^I(s)$ can be accepted when P is at s. Symmetrically, when P is at s, only actions actions in $\Gamma_P^O(s)$ can be produced. We say that P is *well-formed* if $S_P^{init} \neq \emptyset$. Ill-formed interfaces correspond to the input assumption *false* and are not useful: no environment can interact with such interfaces in a meaningful way.

The behavior of a player, i.e. the successive choices being made in the course of the game, is given by a strategy. A strategy is a function that, given the history of the game, i.e. a sequence of states, yields zero or more of the player's enabled moves.

Definition 2 (strategies and outcomes) An *input* (resp. *output*) strategy for P is a mapping $\pi^I : S_P^+ \mapsto 2^{\mathcal{A}_P^I}$ (resp., a mapping $\pi^O : S_P^+ \mapsto 2^{\mathcal{A}_P^O}$)

[6] This requirement is not present in [dAH01a], but simplifies the technicalities, while the main concepts are the same.

such that, for all $s \in S_P$ and all $\sigma \in S_P^*$, we have $\pi^I(\sigma\, s) \subseteq \Gamma_P^I(s)$ (resp. $\pi^O(\sigma\, s) \subseteq \Gamma_P^O(s)$). We denote by Π_P^I and Π_P^O the set of input and output strategies of P, respectively. ∎

An input and an output strategy jointly determine a *set* of *outcomes* in S_P^+: at each step, if the input strategy proposes a set $\mathcal{B}^I$ of input actions, and the output strategy proposes a set $\mathcal{B}^O$, an action from $\mathcal{B}^I \cup \mathcal{B}^O$ is selected nondeterministically. Since our definitions of compatibility and composition do not require the consideration of progress properties, we define the outcomes of strategies in terms of finite traces.

Definition 3 (strategy outcomes) Given a state $s \in S_P$, an input strategy $\pi^I \in \Pi_P^I$ and an output strategy $\pi^O \in \Pi_P^O$, the set $Outc_P(s, \pi^I, \pi^O) \subseteq S_P^+$ is the smallest set satisfying the following clauses:

- $s \in Outc_P(s, \pi^I, \pi^O)$;
- if $\sigma\, t \in Outc_P(s, \pi^I, \pi^O)$ for $\sigma \in S_P^+$ and $t \in S_P$, then for all $a \in \pi^I(\sigma\, t) \cup \pi^O(\sigma\, t)$ and for all $t' \in S_P$ such that $t \xrightarrow{a}_P t'$, we have $\sigma\, t\, t' \in Outc_P(s, \pi^I, \pi^O)$. ∎

We say that a state s *occurs* in an outcome $s_0 \ldots s_n$ (written by slight abuse of notation $s \in s_0 \ldots s_n$) if there is $k \in [0..n]$ such that $s = s_k$. A state $s \in S_P$ is *reachable* in P if there are strategies $\pi^I \in \Pi_P^I$ and $\pi^O \in \Pi_P^O$ and a state $s_0 \in S_P^{init}$ such that s appears in some outcome in $Outc(s_0, \pi^I, \pi^O)$; otherwise s is *unreachable*.

2.1 Compatibility and Composition

We define the composition of two interface P and R in four steps. First, we require that P and R are *composable*, i.e. that their action signatures match. If so, we define the *product* $P \otimes R$ as the classical automaton-theoretic product, where P and R synchronize on shared actions and evolve independently on others. Within this product, we identify a set of *locally incompatible states*, where P can produce an output that is not accepted by R, or vice versa. Finally, we obtain the composition $P \| R$ from $P \otimes R$ by strengthening the input assumptions of $P \otimes R$ in such a way that all locally incompatible states are avoided, thus ensuring that P and R mutually satisfy their input assumptions.

Definition 4 (composability) Two interface automata P and R are *composable* if $\mathcal{A}_P^O \cap \mathcal{A}_R^O = \emptyset$. We let $shared_{P,R} = \mathcal{A}_P \cap \mathcal{A}_R$ to be the set of shared actions of P and R. ∎

The product of two composable interface automata P and R is an interface automaton $P \otimes R$ that represents the joint behavior of P and R. The state space of $P \otimes R$ consists of pairs (s, t), reflecting that P is in state s and R is in state t. In the product, the shared actions of P and R are synchronized: whenever an automaton performs a transition involving a shared action, the other automaton should also do so; if it cannot, the transition is not part of the product. The automata interleave asynchronously all non-shared actions: one automaton takes a step, while the other stays in the same state.

Definition 5 (interface product) If P and R are composable interface automata, their *product* $P \otimes R$ is the interface automaton defined by

$$\begin{aligned}
S_{P\otimes R} =& S_P \times S_R \\
S^{init}_{P\otimes R} =& S^{init}_P \times S^{init}_R \\
\mathcal{A}^O_{P\otimes R} =& \mathcal{A}^O_P \cup \mathcal{A}^O_R \\
\mathcal{A}^I_{P\otimes R} =& (\mathcal{A}^I_P \cup \mathcal{A}^I_R) \setminus \mathcal{A}^O_{P\otimes R} \\
\mathcal{T}_{P\otimes R} =& \{(s,t) \xrightarrow{a} (s',t) \mid s \xrightarrow{a}_P s' \wedge a \in \mathcal{A}_P \setminus \mathcal{A}_R\} \cup \\
& \{(s,t) \xrightarrow{a} (s,t') \mid t \xrightarrow{a}_R t' \wedge a \in \mathcal{A}_R \setminus \mathcal{A}_P\} \cup \\
& \{(s,t) \xrightarrow{a} (s',t') \mid s \xrightarrow{a}_P s' \wedge t \xrightarrow{a}_R t' \wedge a \in \mathcal{A}_P \cap \mathcal{A}_R\}. \blacksquare
\end{aligned}$$

Example 1 The automaton R in Figure 2(b) represents the interface of a receiver component. In state r_0, R can receive a message, in which case it moves to the state r_1. In r_1, it processes the message and moves back to r_0. Since R cannot receive a message in state r_1, it can hold only one message at the time. The product $B \otimes R$ is displayed in Figure 2(c). Note that B and R synchronize on *rec!* and evolve independently on *proc!* and *snd?*.

The product $P \otimes R$ may contain states in which one of the components (say P) can produce an output action that is an input action of the other automaton (R), but is not accepted. This constitutes a violation of the input assumptions of P, and such states are said to be *locally incompatible.*

Definition 6 (locally incompatible states) Given two composable interface automata P and R, the set $Error(P,R)$ of *locally incompatible states* is defined by

$$\begin{aligned}
Error(P,R) \;=\; \Big\{(s,t) \in S_P \times S_R \,\Big|\, & \exists a \in shared_{P,R} \,.\, \big(a \in \Gamma^O_P(s) \setminus \Gamma^I_R(t) \vee \\
& a \in \Gamma^O_R(t) \setminus \Gamma^I_P(s)\big)\Big\}. \blacksquare
\end{aligned}$$

Example 2 The state (b_1, r_1) is an error state in the product $B \otimes R$ (Figure 2(c)), because there is $b_1 \xrightarrow{rec!} b_0$ in B but $rec? \notin \Gamma^I_R(r_1)$. Similarly, the state (b_2, r_1) is an error state.

After forming the product $P \otimes R$ of P and R, we must strengthen the input assumptions of $P \otimes R$ to ensure that no local incompatibility is reached. This corresponds to synthesizing the weakest input assumption that ensures that both the original input assumptions of P and R are respected. This is an example of *assumption propagation:* the original assumptions of P and R propagate and combine into a new, and possibly stronger, assumption for their composition $P \| R$. To this end, we say that a state of $P \otimes R$ is *incompatible* if a locally incompatible state can be reached regardless of how we constrain the environment. That is, s is incompatible if there is no input strategy that from s avoids all locally incompatible states. For example, if $B \otimes R$ is in the state (b_2, r_0), no matter how we constrain the environment, the system cannot be prevented from taking the *rec!*-transition, leading to the error state (b_1, r_1). This is because the environment can only influence the system through its input actions, and *rec!* is an output action.

Definition 7 (compatible states) A state s of $P \otimes R$ is called *compatible* with respect to $Error(P, R)$ if there is $\pi^I \in \Pi^I_{P\otimes R}$ such that, for all $\pi^O \in \Pi^O_{P\otimes R}$, all $\sigma \in Outc_{P\otimes R}(s, \pi^I, \pi^O)$, and all $w \in Error(P, R)$, we have $w \notin \sigma$. We write $Cmp(P, R)$ for the set of compatible states and $Incmp(P, R) = S_{P\otimes R} \setminus Cmp(P, R)$ for the set of incompatible ones. ■

Example 3 With reference to Figure 2(c), $Incmp(B, R) = \{(b_1, r_1), (b_2, r_0), (b_2, r_1)\}$.

Since input strategies can only prevent input actions from occurring, but cannot restrict output actions, we have that a state is incompatible iff it can reach a local incompatibility by following output actions only. This observation provides an efficient criterion for checking the compatibility of states.

Lemma 1 *A state $s \in S_{P\otimes R}$ is compatible in $P \otimes R$ with respect to $Error(P, R)$ iff there is no sequence $s_0 s_1 \dots s_n \in S^*_{P\otimes R}$ with $s_0 = s$, $s_n \in Error(P, R)$, and such that for all $0 \leq k < n$, there is $a_k \in \Gamma^O_{P\otimes R}(s_k)$ with $u_k \xrightarrow{a_k}_{P\otimes R} u_{k+1}$.*

If the initial state of $P \otimes R$ is incompatible, then no environment of $P \otimes R$ can avoid entering the error state. Therefore such interfaces P and R are incompatible.

Definition 8 (compatibility) Two composable interface automata P and R are *incompatible* if $S^{init}_{P\otimes R} \cap Cmp(P, R) = \emptyset$. They are *compatible* if $S^{init}_{P\otimes R} \cap Cmp(P, R) \neq \emptyset$. ■

Example 4 The interfaces B and R are clearly compatible, as the initial state (b_0, r_0) is so with respect to $Error(B, R)$. Indeed, in state (b_0, r_1) the environment can prevent entering error states by not providing the input *snd?*. Hence, while the state (b_0, r_1) itself does not have to be avoided, its outgoing *snd?* action should be avoided. This is achieved automatically by removing the incompatible state (b_1, r_1), along with the transitions leading to it.

This example illustrates how strengthening the input assumptions to avoid locally incompatible states can be performed by simply pruning all incompatible states, along with the transitions leading to them.

Definition 9 (interface composition) For two composable interface automata P and R, the *composition* $P \| R$ is an interface automaton with the same action sets as $P \otimes R$. The states are $S_{P\|R} = Cmp(P, R)$, $S^{init}_{P\|R} = S^{init}_{P\otimes R} \cap Cmp(P, R)$, and the steps are $\mathcal{T}_{P\|R} = \mathcal{T}_{P\otimes R} \cap (Cmp(P, R) \times \mathcal{A}_{P\|R} \times Cmp(P, R))$. ■

Example 5 The composition $B \| R$, displayed in Figure 2(d), is obtained by removing the incompatible states (b_2, r_0) and (b_1, r_1) from $B \otimes R$. Notice how the constraint that R can only hold one message migrates from R to the composition $B \| R$. The input assumptions of $B \| R$ require that no message must arrive before the previous one has been processed. In state (b_0, r_1), where the receiver already holds a message, this is achieved by disabling the *snd?* action. To prevent entering (b_2, r_0) (and hence (b_1, r_1)), the action *snd?* also has to be disabled in state (b_1, r_0). Again, the sender should not provide a new message until R has processed the old one, but now the old message is still in the buffer.

2.2 Properties of Interface Automata

Let a proper environment for P and R be an interface automaton E such that:

- E is well-formed;
- E is composable with $P \otimes R$;
- E synchronizes on every output action of $P \otimes R$, i.e. $\mathcal{A}^I_E = \mathcal{A}^O_{P \otimes R}$;
- if $(s, t) \in \mathit{Error}(P, R)$, then (u, s, t) is unreachable in $E \otimes P \otimes R$, for every state $u \in S_E$.

The result below shows that two automata are compatible if there exists at least one environment that makes the automata satisfy each other's input assumptions.

Theorem 1 *Let P and R be composable interfaces. The following statements are equivalent.*

(i) *P and R are compatible,*

(ii) *$P \| R$ is well-formed,*

(iii) *there exists a proper environment for P and R.*

The following result states that composition is transitive, i.e. that the order in which we compose multiple components is irrelevant, if we restrict our attention to the reachable states. We write $P \equiv R$ if P and R are identical once we remove all unreachable states.

Theorem 2 *Let P, R and M be pairwise composable interface automata. Then $(P \| R) \| M \equiv P \| (M \| R)$.*

3 Timed Interface Automata

This section extends the interface automaton model with timing constraints, yielding *timed interface automata* [dAHS02]. A timed interface automaton augments an interface automaton with a set of real-valued clocks. Clocks occur in location invariants and transition guards, respectively specifying deadlines and enabling conditions on the actions of the interface. Timed interface automata are syntactically similar to timed automata [AD94], except that they have two kinds of invariants, one for input and one for output actions. Semantically, however, the two models differ: timed automata are interpreted as labeled transition systems, while timed interfaces are interpreted as timed games.

3.1 The timed interface model

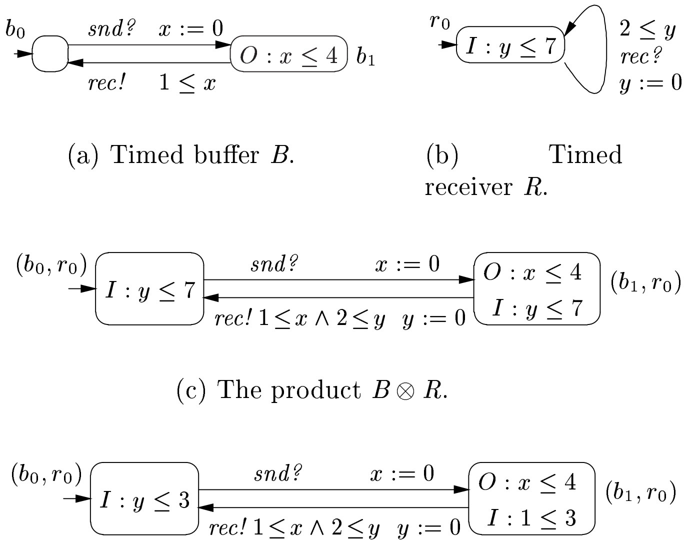
The timed interface automaton B in Figure 3(a) represents a 1-place buffer, which delivers messages within 1 to 4 time units. The clock x measures the time since the last arrival of a message. In location [7] b_0, the buffer is empty and can receive a message. Upon receiving a message, it moves to location b_1, and it resets the clock x. The *output invariant* $x \leq 4$ in location b_1 specifies that the location must be left before $x > 4$, thus forcing a delivery (action *snd!*) within 4 time units. The *transition guard* $1 \leq x$ specifies that the action *snd!* can be taken if $1 \leq x$, thus enabling a delivery after 1 time unit. Note that all time is spent in locations; transitions are instantaneous, i.e. take no time.

The timed interface in Figure 3(b) represents a component that must receive a message every 2 to 7 time units: the clock y measures the time between two consecutive message receipts. The *input invariant* $y \leq 7$ forces the input action *rec?* to be taken within 7 time units of the previous receipt. The transition guard $2 \leq y$ says that this action can be taken after a minimum delay of 2 time units. Thus, invariants express when actions must be taken and guards express when they can be taken. Guards and invariants are specified by clock conditions, that are any boolean combination of formulas of the form $x \prec c$ or $x - y \prec c$, where c is an integer, x, y are clocks in a given set $\mathcal{X}$, and $\prec$ is either of $<$ or $\leq$. We denote the set of all clock conditions over $\mathcal{X}$ by $K[\mathcal{X}]$.

Definition 10 (timed interface automaton) A *timed interface automaton* (or TIA) is a tuple $P = (Q_P, q_P^{init}, \mathcal{X}_P, \mathcal{A}_P^I, \mathcal{A}_P^O, Inv_P^I, Inv_P^O, \mathcal{T}_P)$ consisting of the following components.

- Q_P is a finite set of *locations.*
- $q_P^{init} \in Q_P$ is the *initial location.*
- $\mathcal{X}_P$ is a finite set of *clocks.*
- $\mathcal{A}_P^I$ and $\mathcal{A}_P^O$ are finite and disjoint sets of immediate *input* and *output actions*, respectively. Let $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O$ denote the set of all immediate actions of P.
- Inv_P^I: $Q_P \to K[\mathcal{X}_P]$ maps each location of P to its *input invariant.*
- Inv_P^O: $Q_P \to K[\mathcal{X}_P]$ maps each location of P to its *output invariant.*
- $\mathcal{T}_P \subseteq Q_P \times K[\mathcal{X}_P] \times \mathcal{A}_P \times 2^{\mathcal{X}_P} \times Q_P$ is the *transition relation.* For $(q, g, a, r, q') \in \mathcal{T}_P$, $q \in Q_P$ is the *source* of transition, $q' \in Q_P$ is the *destination*, $g \in K[\mathcal{X}_P]$ is the *transition guard*, $a \in \mathcal{A}_P$ is an *immediate action*,

[7] The nodes of timed interface automata are called locations, because the word 'state' already refers to a location together with a clock valuation, see below.



(a) Timed buffer B.

(b) Timed receiver R.

(c) The product $B \otimes R$.

(d) The composition $B||R$.

Fig. 3. Product and composition of TIAs

and $r \subseteq \mathcal{X}_P$ is a *reset set*, containing the clocks that are reset. We require the transition relation to be deterministic: for all $q \in Q_P$ and $a \in \mathcal{A}_P$, there is at most one tuple of the form (q, g, a, r, q') with $(q, g, a, r, q') \in \mathcal{T}_P$. We sometimes write $q \xrightarrow{g\ a\ r}_P q'$ for $(q, g, a, r, q') \in \mathcal{T}_P$. ∎

3.2 The Game underlying an Interface

We unfold a TIA P into a game structure $[\![P]\!]$ by explicitly recording the clock values in P and by separating the transition relation $\rightarrow_P$ into an input transition relation $\rightarrow^I_{[\![P]\!]}$ and an output transition relation $\rightarrow^O_{[\![P]\!]}$.

A *valuation* over a set $\mathcal{X}$ of clock variables is a function v: $\mathcal{X} \mapsto \mathbb{R}^{\geq 0}$ that assigns a clock value to every clock in $\mathcal{X}$. We write $\mathbf{0}_\mathcal{X}$ (or just $\mathbf{0}$ if $\mathcal{X}$ is clear from the context) for the valuation that assigns 0 to all clocks in $\mathcal{X}$. Other clock valuations are often listed as a set of pairs, as in $\{x = 1, y = 3\}$. The set of all clock valuations is denoted by $Val(\mathcal{X})$ and for clock valuation v and a clock expression g, we can determine whether g holds for this valuation. If so, we write $v \models g$. For example, if $v(x) = 1$ and $v(y) = 3$, then $v \models x - y \leq 0$. For a valuation $v \in Val(\mathcal{X})$, we write $v + d$ for the valuation defined by $(v + d)(x) = v(x) + d$ for all $x \in \mathcal{X}$. Given a set $r \subseteq \mathcal{X}$ of clocks, we write $v[r := 0]$ for the valuation that maps x to 0 if $x \in r$, and otherwise to $v(x)$.

Let P be a TIA with components $(Q, q^{init}, \mathcal{X}, \mathcal{A}^I, \mathcal{A}^O, Inv^I, Inv^O, \mathcal{T})$. We obtain $[\![P]\!]$ from P as follows. The states (q, v) of $[\![P]\!]$ consist of a location q in P and a clock valuation $v \in Val(\mathcal{X})$. Thus, a state records the location of the interface and the values of all its clocks. Initially, all clocks are 0 and the two invariants have to be met. That is, $[\![P]\!]$ has an initial state $(q^{init}, \mathbf{0})$ if $\mathbf{0}$ meets the invariants $Inv^I(q^{init})$ and $Inv^O(q^{init})$ of the initial location q^{init}, (i.e. $\mathbf{0} \models Inv^I(q^{init}) \wedge Inv^O(q^{init})$). Otherwise, $[\![P]\!]$ has no initial state.

The input and output transition relations $\rightarrow^I_{[\![P]\!]}$ and $\rightarrow^O_{[\![P]\!]}$ update the location and clock values. We distinguish between *timed* (or *delay*) transitions, which are labeled by delay actions $d \in \mathbb{R}^{\geq 0}$, and *immediate* transitions, labeled by immediate actions $a \in \mathcal{A}$. Let γ be one of the players I or O. A timed transition $s \xrightarrow{d}{}^{\gamma} s'$ represents the passage of d time units: writing $s = (q, v)$, we have $s' = (q, v + d)$. The transition $s \xrightarrow{d}{}^{\gamma} s'$ is enabled if the location invariant $Inv^\gamma(q)$ of player γ holds at all times between 0 and d. Writing $s = \langle q, v\rangle$ and $s' = \langle q', v'\rangle$, the *immediate transition* $s \xrightarrow{a}{}^{\gamma} s'$ changes the state as specified by a transition $q \xrightarrow{g\ a\ r} q'$ in P. This transition is enabled if (1) the guard of the transition involved is met in s, i.e. $v \models g$, (2) the player γ's invariant is met both in the source s and in the destination s' i.e. $v \models Inv^\gamma(q)$ and $v' \models Inv^\gamma(q')$, and (3) the clock variables in the set r are reset to 0. The precise definition is as follows.

Definition 11 (game structure of timed interface automaton) A TIA P induces a game structure $[\![P]\!]$, which is a tuple $\langle S_{[\![P]\!]}, S^{init}_{[\![P]\!]}, \mathcal{A}^I_{[\![P]\!]}, \mathcal{A}^O_{[\![P]\!]}, \rightarrow^I_{[\![P]\!]}, \rightarrow^O_{[\![P]\!]}\rangle$ consisting of the following components.

- The state set $S_{[\![P]\!]} = \{\langle q, v\rangle \mid q \in Q_P,\ v \in Val(\mathcal{X}_P)\}$.
- The initial states $S^{init}_{[\![P]\!]} = \{\langle q^{init}, \mathbf{0}_{\mathcal{X}_P}\rangle \mid \mathbf{0}_{\mathcal{X}_P} \models Inv^I_P(q^{init}_P) \wedge Inv^O_P(q^{init}_P)\}$.
- The actions are $\mathcal{A}^I_{[\![P]\!]} = \mathcal{A}^I_P \cup \mathbb{R}^{\geq 0}$ and $\mathcal{A}^O_{[\![P]\!]} = \mathcal{A}^O_P \cup \mathbb{R}^{\geq 0}$.
- For $\gamma \in \{I, O\}$ the transition relations of $[\![P]\!]$ contains a transition $\langle q, v\rangle \xrightarrow{\alpha}{}^{\gamma}_{[\![P]\!]} \langle q', v'\rangle$ if one of the following two conditions holds:
  - *Time step:* $\alpha \in \mathbb{R}^{\geq 0}$, $q = q'$, $v' = v + \alpha$, and for all $0 \leq d' \leq \alpha$, we have $v + d' \models Inv^\gamma_P(q)$;
  - *Discrete step:* $\alpha \in \mathcal{A}^\gamma_P$, and there is a transition $q \xrightarrow{g\ \alpha\ r}_P q'$ with $v \models Inv^\gamma_P(q) \wedge g$, $v' = v[r := 0]$, and $v' \models Inv^\gamma_P(q')$. ■

Example 6 The states of $[\![B]\!]$ are $\langle b_0, \{x = d\}\rangle$ and $\langle b_1, \{x = d\}\rangle$, for every clock value d in $\mathbb{R}^{\geq 0}$. The transitions of $[\![B]\!]$ are

$$\begin{array}{ll}
\langle b_0, \{x = d\}\rangle \xrightarrow{d'}{}^{I} \langle b_0, \{x = d + d'\}\rangle & \\
\langle b_0, \{x = d\}\rangle \xrightarrow{d'}{}^{O} \langle b_0, \{x = d + d'\}\rangle & \\
\langle b_0, \{x = d\}\rangle \xrightarrow{snd?}{}^{I} \langle b_1, \{x = 0\}\rangle & \\
\langle b_1, \{x = d\}\rangle \xrightarrow{d'}{}^{O} \langle b_1, \{x = d + d'\rangle, & \text{for } d + d' \leq 4, \\
\langle b_1, \{x = d\}\rangle \xrightarrow{rec!}{}^{O} \langle b_0, \{x = d\}\rangle, & \text{for } 1 \leq d.
\end{array}$$

In each state s of the game $[\![P]\!]$, both players propose one of their available moves. That is, each player γ proposes an immediate or timed move α such that $s \xrightarrow{\alpha}{}^{\gamma} s'$, for some s'. The moves proposed by both players together determine a successor state. If both players choose timed moves d and $d' \in \mathbb{R}^{\geq 0}$, then global time will advance by $\min\{d, d'\}$; if one player chooses an immediate move a, while the other chooses a timed move d, the immediate move

a will be carried out; if both players choose immediate moves, one of them occurs nondeterministically. Formally, the outcome of two moves is a triple (α, γ, s'), where α is the action being taken, γ is the player who took it and s' the destination state.

Definition 12 (moves and move outcomes) For $\gamma \in \{I, O\}$, a player-γ *move* in a state s of $[\![P]\!]$ is an action $\alpha \in \mathcal{A}_P \cup \mathbb{R}^{\geq 0}$ such that $s \xrightarrow{\alpha}{}^{\gamma}_{[\![P]\!]} s'$ for some s'. This state s' is unique and we write $\delta(s, \alpha)$ for s'. We indicate with $\Gamma^{\gamma}_{[\![P]\!]}(s)$ the set of all player-γ moves in state s and $\Gamma^{\gamma}_{[\![P]\!]} = \mathcal{A}^{\gamma}_{[\![P]\!]} \cup \mathbb{R}^{\geq 0}$ the set of all γ-moves.

For all states $s \in S_{[\![P]\!]}$ and all moves $\alpha_I \in \Gamma^{I}_{[\![P]\!]}(s)$ and $\alpha_O \in \Gamma^{O}_{[\![P]\!]}(s)$, the *outcome* $outc_{[\![P]\!]}(s, \alpha_I, \alpha_O)$ is given by

$$
outc_{[\![P]\!]}(s, \alpha_I, \alpha_O) = \begin{cases} \{(\alpha_I, I, \delta_{[\![P]\!]}(s, \alpha_I))\} & \text{if } \alpha_I \in \mathcal{A}_P, \alpha_O \in \mathbb{R}^{\geq 0}, \\ & \text{or } \alpha_I, \alpha_O \in \mathbb{R}^{\geq 0},\ \alpha_I < \alpha_O. \\ \{(\alpha_O, O, \delta_{[\![P]\!]}(s, \alpha_O))\} & \text{if } \alpha_I \in \mathbb{R}^{\geq 0}, \alpha_O \in \mathcal{A}_P, \\ & \text{or } \alpha_I, \alpha_O \in \mathbb{R}^{\geq 0},\ \alpha_I > \alpha_O. \\ \{(\alpha_I, I, \delta_{[\![P]\!]}(s, \alpha_I)), (\alpha_O, O, \delta_{[\![P]\!]}(s, \alpha_O))\} & \text{otherwise.} \end{cases}
$$

∎

As in the untimed case, the successive choices being made by a player in the course of the game, are given by a strategy and the an input and an output strategy jointly determine a set of outcomes in S^{+}_{P}. However, unlike in the untimed case, the time progress conditions require us to consider both finite and infinite outcomes here.

Definition 13 (strategies and strategy outcomes) A *strategy* for player $\gamma \in \{I, O\}$ is a function $\pi^{\gamma}\colon S^{+}_{[\![P]\!]} \mapsto 2^{\Gamma^{\gamma}_{[\![P]\!]}}$ that associates with every sequence of states $s_0 s_1 \ldots s_n \in S^{+}_{[\![P]\!]}$ a subset of moves $\pi^{\gamma}(s_0 s_1 \ldots s_n) \subseteq \Gamma^{\gamma}_{[\![P]\!]}(s_n)$. We require that $\pi^{\gamma}(s_0 s_1 \ldots s_n) = \emptyset$ only when $\Gamma^{\gamma}_{[\![P]\!]}(s_n) = \emptyset$, thus forcing a strategy to choose at least one move, if any moves are available. For $\gamma \in \{I, O\}$, we denote by $\Pi^{\gamma}_{[\![P]\!]}$ the set of all strategies for player γ.

Given a state $s \in S_{[\![P]\!]}$, an input strategy $\pi^I \in \Pi^{I}_{[\![P]\!]}$, and an output strategy $\pi^O \in \Pi^{O}_{[\![P]\!]}$, the set of *outcomes* $Outc_{[\![P]\!]}(s, \pi^I, \pi^O)$ of π^I and π^O from s consists of all finite and infinite sequences $\sigma = s_0 \alpha_1 pl_1 s_1 pl_1 \alpha_2 pl_2 s_2 \ldots$ such that

- $s_0 = s$;
- for all $n < length(\sigma)$, there are $\beta^I \in \pi^I(s_0 s_1 \ldots s_n)$ and $\beta^O \in \pi^O(s_0 s_1 \ldots s_n)$ such that $(\alpha_{n+1}, s_{n+1}, pl_{n+1}) \in outc_{[\![P]\!]}(s_n, \beta^I, \beta^O)$.
- if $length(\sigma) < \infty$, then σ ends in a pair (s_k, pl_k) such that either $\Gamma^{I}_{[\![P]\!]}(s_k) = \emptyset$ or $\Gamma^{O}_{[\![P]\!]}(s_k) = \emptyset$. ∎

A state is reachable if there is a combination of strategies for both players that lead to it.

Definition 14 (reachable states) A state $s \in S_{[\![P]\!]}$ is *reachable* in $[\![P]\!]$ if there are strategies $\pi^I \in \Pi^I_{[\![P]\!]}$ and $\pi^O \in \Pi^O_{[\![P]\!]}$, a state $q_0 \in S^{init}_{[\![P]\!]}$, and an outcome $\sigma = s_0, a_1, pl_1, s_1, \ldots$ in $Outc_{[\![P]\!]}(q_0, \pi^I, \pi^O)$ such that $s = s_k$ for some $k \geq 0$. ∎

The objective for a player is to play a strategy that ensures that all game outcomes belong to a set of desirable outcomes, called the *goal* for that player. We are particularly interested in three kind of goals: the set $\Box U$ containing all outcomes that stay within a set of good states U; the set t_div of outcomes along which time progresses; and the set $blame^\gamma$, where Player γ is blamed for monopolizing the game, i.e. for always playing first from a certain point on.

Definition 15 (interface product) For a TIA P, we define

$$\begin{aligned}
\Box U &= \{\sigma = s_0, a_1, pl_1, s_1, \ldots \in Outc_{[\![P]\!]} \mid \forall k \leq length(\sigma) \,.\, s_k \in U\} \\
blame^\gamma &= \{\sigma = s_0, a_1, pl_1, s_1, \ldots \in Outc_{[\![P]\!]} \mid \\
&\qquad\qquad length(\sigma) = \infty \wedge \exists n \,.\, \forall k \geq n \,.\, pl_k = \gamma\} \\
t_div &= \{\sigma = s_0, a_1, pl_1, s_1, \ldots \in Outc_{[\![P]\!]} \mid \sum_{k=0}^{length(\sigma)} delay(a_k) = \infty\}
\end{aligned}$$

Here, for a move α, $delay(\alpha) = \alpha$ if $\alpha \in \mathbb{R}^{\geq 0}$, and $delay(\alpha) = 0$ otherwise. ∎

3.3 Well-formedness

Only game outcomes along which time diverges have a physical meaning. Behaviors such as $s0Is0Os0Is0Os\ldots$ and $s\frac{1}{2}Is\frac{1}{4}Os\frac{1}{8}Is\ldots$ in which total amount of time $\sum_{i=1}^{\infty} delay(a_i)$ is finite (where a_i is the i^{th} action in the sequence) do not correspond to behaviors of physical systems. Thus, we want to ensure that a player never becomes "painted into a corner", unable to let time diverge regardless of how she plays. Of course, a player can never ensure time advances, since the other player could be blocking the advancement of time (for instance, by always playing a time-step of length 0). Hence, we call *well-formed* the states from which both player can ensure time advances, unless prevented from doing so by the other player. Precisely, we say that a state is *well-formed* if both players γ can win with respect to the goal $t\_div \cup blame^{1-\gamma}$. A timed interface is well-formed if all reachable states are well-formed. We refer the reader to [dAHS02] for an algorithm that decides whether a TIA is well-formed.

Definition 16 (well-formedness) A state $s \in S_{[\![P]\!]}$ is *well-formed* if both of the following conditions hold:

(i) Input can win the game with goal $t\_div \cup blame^O$; that is, if for all strategies $\pi^O \in \Pi^O_{[\![P]\!]}$ there is a strategy $\pi^I \in \Pi^I_{[\![P]\!]}$ such that $\sigma \models t\_div \cup blame^O$ for all outcomes $\sigma \in Outc_{[\![P]\!]}(s, \pi^I, \pi^O)$ and

(ii) Output can win the game with goal $t\_div \cup blame^I$; that is, if there is a

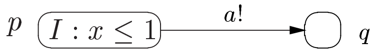


Fig. 4. An ill-formed TIA.

strategy $\pi^O \in \Pi^O_{[\![P]\!]}$ such that for all strategies $\pi^I \in \Pi^I_{[\![P]\!]}$ and outcomes $\sigma \in Outc_{[\![P]\!]}(s, \pi^I, \pi^O)$, we have $\sigma \models t\_div \cup blame^I$.

The interface P is well-formed if $[\![P]\!]$ has an initial state, and every reachable state in $[\![P]\!]$ is well-formed. ∎

The order of the quantification (first over output strategies, then over input strategies) makes the game turn-based. i.e. the Output player chooses its move first and Input can use this move to determine its own move. This is due to the asymmetrical causality relation between inputs and outputs in composition, as explained in [dAHS02].

Example 7 The timed interface in Figure 4 specifies that in location p an input should come before the deadline $x = 1$, whereas there is no input action to help the automaton out of p. Note that no environment satisfies the input assumptions of this automaton. Bound by the deadline $x \leq 1$, the Input player does not have a strategy to let time pass, when the Output player plays moves with a duration 2. Hence, this interface is not well-formed. The receiver R (Figure 3(b)) is well-formed because (1) the Input player can let time pass: it can play the *rec?* action if $y = 7$ and timed move with duration 1 otherwise. (2) the Output player can let time pass: since there are no output actions, Output can for instance always play moves with a duration 1.

3.4 Product and Composition

As in the untimed case, the composition of timed interfaces is defined via the notions of composability, product, error states, and uncontrollable states.

Definition 17 Two TIAs P and R are *composable* if $\mathcal{A}^O_P \cap \mathcal{A}^O_R = \emptyset$ and $\mathcal{X}_P \cap \mathcal{X}_R = \emptyset$. We denote by $shared_{P,R} = \mathcal{A}_P \cap \mathcal{A}_R$ their *shared actions*. ∎

As before, the product of two timed interfaces represents the joint behavior of the components, which synchronize on shared actions and interleave asynchronously on non-shared actions. The input invariant in location (s, t) is the conjunction the input invariants in s and t, requiring that the product automaton should satisfy the deadlines expressed by both automata being composed. The output invariants in s and t are conjoined as well. For a shared action a, the two transitions $s \xrightarrow{g\ a\ r} t$ and $s' \xrightarrow{g'\ a\ r'} t'$ yields the transition $(s, s') \xrightarrow{g \wedge g'\ a\ r \cup r'} (t, t')$ obtained by conjoining the invariants g and g' and taking the union of $r \cup r'$ of the reset sets.

Definition 18 (interface product) Given two composable TIAs P_1 and P_2, the *product* $P_1 \otimes P_2$ is the TIA with

- $Q_{P_1 \otimes P_2} = Q_{P_1} \times Q_{P_2}$, and $q^{init}_{P_1 \otimes P_2} = (q^{init}_{P_1}, q^{init}_{P_2})$.

- $\mathcal{X}_{P_1 \otimes P_2} = \mathcal{X}_{P_1} \cup \mathcal{X}_{P_2}$.
- $\mathcal{A}^I_{P_1 \otimes P_2} = (\mathcal{A}^I_{P_1} \cup \mathcal{A}^I_{P_2}) \setminus \mathit{shared}_{P_1,P_2}$, and $\mathcal{A}^O_{P_1 \otimes P_2} = \mathcal{A}^O_{P_1} \cup \mathcal{A}^O_{P_2}$.
- $\mathit{Inv}^I_{P_1 \otimes P_2}(p, q) = \mathit{Inv}^I_{P_1}(p) \wedge \mathit{Inv}^I_{P_2}(q)$ and $\mathit{Inv}^O_{P_1 \otimes P_2}(p, q) = \mathit{Inv}^O_{P_1}(p) \wedge \mathit{Inv}^O_{P_2}(q)$.
- $(q_1, q_2) \xrightarrow{g_1 \wedge g_2 \ a \ r_1 \cup r_2} (q'_1, q'_2)$ is a transition of $P_1 \otimes P_2$ iff, for $i = 1, 2$, if $a \in \mathcal{A}_{P_i}$, then there is a transition $q_i \xrightarrow{g_i \ a \ r_i}_{P_i} q'_i)$ in $\mathcal{T}_{P_i}$; otherwise $q_i = q'_i$, $g_i = \mathit{true}$, and $r_i = \emptyset$. ∎

Example 8 The product $B \otimes R$ is of B and R is displayed in Figure 3(c).

The composition of two TIAs is again obtained from their product by strengthening the input assumptions to avoid all error states. In TIAs, input strengthening means strengthening the input invariants.[8] A product of two TIAs may contain two kind of locally incompatible states: I/O-incompatible states and timing-incompatible states. A state is *I/O-incompatible* when one component can preform an output action that is not accepted by the other component, as in the untimed case.

Definition 19 (I/O-incompatible states) Given two composable interface automata P and R, the set $\mathit{Error}(P, R)$ of *I/O-incompatible states* is defined by

$$\begin{aligned}\mathit{Error}(P, R) \ = \ & \big\{(v, u) \in S_{[\![P]\!]} \times S_{[\![R]\!]} \;\big|\; \exists a \in \mathit{shared}_{P,R} \,. \\ & \big(a \in \Gamma^O_R(u) \setminus \Gamma^I_P(v) \vee a \in \Gamma^O_P(v) \setminus \Gamma^I_R(u)\big)\big\}.\end{aligned}$$

We write $\mathit{Good}(P, R)$ for the set of states in $P \otimes R$ that are not in $\mathit{Error}(P, R)$. ∎

Example 9 The state $(b_1, r_0, \{x = 1, y = 1\})$ is an I/O-incompatible state in $B \otimes R$, because $\mathit{rec!} \in \Gamma^O_{[\![B]\!]}(b_1, \{x = 1\})$, but $\mathit{rec?} \notin \Gamma^I_{[\![R]\!]}(r_0, \{y = 1\})$. The state $(b_1, r_0, \{x = 3, y = 3\})$ is I/O-compatible, because $\mathit{rec!} \in \Gamma^O_{[\![B]\!]}(b_1, \{x = 3\})$ and $\mathit{rec?} \in \Gamma^I_{[\![R]\!]}(r_0, \{y = 3\})$.

It is an important property that the set of I/O-incompatible states for a certain location is expressible using clock conditions. The (reachable) I/O-incompatible states in location (b_1, r_0) of $B \otimes R$ are given by the clock condition $1 \leq x \leq 4 \wedge y < 2$. ∎

Timing-incompatible states are the states where at least one of the players cannot let time progress. They typically arise when an input deadline is not met. The state $\langle(b_1, r_0), \{x = 0, y = 3.4\}\rangle$ in $B \otimes R$ is a timing-incompatible state: due to the invariants, we cannot stay in (b_1, r_0) forever. However, we can only leave the state by the *rec!*-transition, which is enabled if $x \geq 1$. This means that we have to remain in (b_1, r_0) for at most one time unit, but if we do so, the input invariant $x \leq 4$ is violated. Hence, $\langle(b_1, r_0), \{x = 0, y = 3.4\}\rangle$ is

[8] It would also make sense to strengthen the guards on input transitions, but we do not need this.

a timing-incompatible state. A timing-incompatible state is simply a state of the product that is not well-formed: we do not need any additional definition of timing incompatibility.

When avoiding I/O-incompatible states, the input player has to let time diverge: it should not avoid those states by blocking time. Thus, we say that a state s in the product is compatible if the input player has a strategy that, at the same time, avoids the I/O-incompatible states and lets time progress (or blames the Output player). Precisely, s is compatible if Input has a strategy that wins with respect to the goal $\mathit{Good}(P,R) \cap (t\_div \cup \mathit{blame}^O)$.

Definition 20 (compatible states) A state s of $[\![P \otimes R]\!]$ is *compatible* if for all strategies $\pi^O \in \Pi^O_{[\![P]\!]}$ there is a strategy $\pi^I \in \Pi^I_{[\![P]\!]}$ such that all outcomes $\sigma \in \mathit{Outc}_{[\![P]\!]}(s, \pi^I, \pi^O)$ satisfy $\sigma \models \Box \mathit{Good}(P,R) \cap (t\_div \cup \mathit{blame}^O)$. The states in $S_{P \otimes R}$ that are not compatible are called *incompatible*. ∎

Example 10 Note that the states in which one of the invariants is violated are always incompatible. By reasoning as in Example 9, one can show that every state $\langle (b_0, r_0), v\rangle$ or $\langle (b_1, r_0), v\rangle$ in $B \otimes R$ with $v(x) > 3$ is incompatible.

A crucial result is that, given a location q, the set of states $\langle q, x\rangle$ from which the Input player can win with respect to the goal $\Box \mathit{Good}(P,R) \cap (t\_div \cup \mathit{blame}^O)$ is expressible as a clock condition, which we denote by $\mathit{Compat}_{P\otimes R}(q)$. The clock conditions $\mathit{Compat}_{P\otimes R}(q)$ can be computed with the game-theoretical algorithms discussed in [dAHS02].

Example 11 Example 10 shows that $\mathit{Compat}_{B\otimes R}(b_0, r_0) = \mathit{Compat}_{B\otimes R}(b_1, r_0) = y \leq 3$.

The composition $P \| R$ is obtained by restricting the product $P \otimes R$ to the states from which player Input can avoid all incompatible states, that is, by strengthening the input invariants $\mathit{Inv}^I_{P\otimes R}(q)$ to $\mathit{Compat}_{P\otimes R}(q)$.

Definition 21 (interface composition) The *composition* $P \| R$ of two timed interface automata P and R is obtained from the product $P \otimes R$ by replacing for each location $q \in Q_{P\otimes R}$ the input invariant $\mathit{Inv}^I_{P\otimes R}(q)$ with $\mathit{Compat}_{P\otimes R}(q)$. The two timed interface automata P and R are *compatible* if their initial state satisfies the new input assumptions, or $\mathbf{0} \models \mathit{Compat}_{P\otimes R}(q^{\mathit{init}}_{P\otimes R})$. ∎

Example 12 From the product $B \otimes R$, we obtain the composition $B \| R$ shown in Figure 3(d).

3.5 Properties of Timed Interface Automata

The result below states that if we compose two well-formed and compatible interfaces, we get a well-formed interface. As well-formedness corresponds to the interface being useful in some environment, we see that composing two useful interfaces that can be used together, yields another useful interface. Note that this result trivially holds in the untimed case.

Theorem 3 *Let P and R be composable TIAs. If P and R compatible and well-formed, then $P\|R$ is well-formed as well.*

Corollary 1 *Let P and R be composable TIAs. Then P and R be compatible if and only if $[\![P\|R]\!]$ has an initial state.*

As before, we write $P \equiv R$ if P and R are isomorphic once we remove all unreachable states. Then interface composition is commutative and associative upto $\equiv$.

Theorem 4 *Let P, R, and M be pairwise composable TIAs. Then, $P\|R \equiv R\|P$, and $(P\|R)\|M \equiv P\|(R\|M)$.*

We remark that, while associativity of composition usually follows immediately from the definition of composition in the standard non-game setting, in our game-theoretic setting it is instead a non-trivial result. In fact, Theorem 4 states that the input assumptions of a system formed by components P, R, M can be computed compositionally in two equivalent ways: either by first computing the requirements that P and R impose on their environment, and then combining these requirements with those of M, or by first computing the requirements that R and M impose on their environment, and then combining these requirements with those of P. In other words, the order in which we migrate the input requirements from the individual components to a composite design does not matter.

References

[Abr96] S. Abramsky. Semantics of interaction. In H. Kirchner, editor, *Trees in Algebra and Programming – CAAP'96, Proc. 21st Int. Coll., Linköping*, volume 1059 of *Lect. Notes in Comp. Sci.*, page 1. Springer-Verlag, 1996.

[Abr97] S. Abramsky. Games in the semantics of programming languages. In *Proc. of the 11th Amsterdam Colloquium*, pages 1–6. ILLC, Dept. of Phylosophy, University of Amsterdam, 1997.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theor. Comp. Sci.*, 126:183–235, 1994.

[AGN97] S. Abramsky, S. Gay, and R. Nagarajan. A type-theoretic approach to deadlock-freedom of asynchronous systems. In *TACS'97: Theoretical Aspects of Computer Software. Third International Symposium*, 1997.

[AHM98] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proc. 13th IEEE Symp. Logic in Comp. Sci.*, pages 334–344. IEEE Computer Society Press, 1998.

[CdAH+02] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, Marcin Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules.

In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, Lect. Notes in Comp. Sci. Springer-Verlag, 2002.

[CdAHM02] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, Lect. Notes in Comp. Sci. Springer-Verlag, 2002.

[CdAHS03] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and M.I.A. Stoelinga. Resource interfaces. In *Proceedings of the Third International Workshop on Embedded Software (EMSOFT 2003)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2003.

[CRR02] S. Chaki, S.K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Proc. 29th ACM Symp. Princ. of Prog. Lang.*, 2002.

[dA03a] L. de Alfaro. Game models for open systems. In *Int. Symposium on Verification celebrating Zohar Manna's 64th Birthday*, volume 2772 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2003.

[dA03b] L. de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verification (Theory in Practice)*, volume 2772 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2003.

[dAH01a] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM Press, 2001.

[dAH01b] L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *EMSOFT 01: 1st Intl. Workshop on Embedded Software*, volume 2211 of *Lect. Notes in Comp. Sci.*, pages 148–165. Springer-Verlag, 2001.

[dAHS02] L. de Alfaro, T.A. Henzinger, and M.I.A. Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT 2002)*, volume 2491 of *Lect. Notes in Comp. Sci.*, pages 108–122. Springer-Verlag, 2002.

[Dil88] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1988.

[RR01] S.K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *Proc. SAS 01, Static Analysis Symposium*, volume 2126 of *Lect. Notes in Comp. Sci.*, pages 375–394. Springer-Verlag, 2001.

[RR02] S.K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, volume 2404 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002.