Interfaces: a game-theoretic framework to reason 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. In previous work, we have developed several interface theories that extend type systems with the ability to reason about the dynamic behavior of software components. The works [dAH01a,CdAH$^+$02,CdAHM02] consider the temporal order in which method calls 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. 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 satisfy each other's input assumptions; (3) composition of two compatible interfaces; (4) component 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] and present techniques for checking well-formedness and compatibility, and for composing interfaces in these interface theories. Due space limitations, we do not treat the notion of interface refinement, but we refer the reader to [dAH01a] and [dA03].

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 propose a formal notion of component interfaces that provides a framework to specify and analyze component interaction. In particular, our interface theories 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 $\texttt{int} \rightarrow \texttt{real}$ expresses that a function expects integers (input assumption) and 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). Here, the input *snd?* represents the arrival of a message from a sender process, and the output *rec!*, models the delivery of a message to a receiver process. In particular, the input action *snd?* is not enabled in

[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

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.

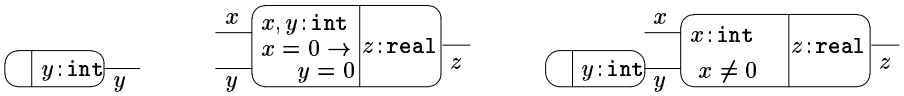
Well-formedness checking. When constructing an interface, we have to make sure that it is *well-formed*, i.e. that there exists at least one environment that satisfies its input assumptions. Otherwise, the interface is useless, since it cannot be used in any design. While rather straightforward in the 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[3]) 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 set to 0.[4] 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 $y : \texttt{int}$ and $z : \texttt{real}$, see Figure 1(c).

Summarizing, the composition of two interfaces yields a new interface for the composite system. Its input assumptions are strengthened in such a way that the component interfaces satisfy each others input assumptions and its output guarantees combine the guarantees present in the components. Compatibility and composition of interface automata will be explained later in this paper.

[3] 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.

[4] 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 $x \neq 0$.



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

Fig. 1. Migration of constraints

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 *error 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 errors. If so, then there exists an environment of the combined system which makes both components satisfy each other's input assumptions. Component composition then boils down to synthezing the most liberal input strategy in the composite system that avoid all error states. This can be done by adapting 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 an error 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 errors: Input should alway choose a value different from 0.

Since the underlying games are relatively simple, the theory for interface automata can be stated without refering to games – as we do here. Due to time progress requirements, timed interfaces induce fairly intricate games.

Related work. Various models for the analysis of components exist. However, few of them handle open systems that make input assumptions in a compositional way.

Many models are unable to express input assumptions. I/O automata [LT89,MMT91,SGSAL98], SMV [CMCHG96], and Reactive Modules [AH99] require input-enabledness, meaning that a component must always be able to accept any possible input. In this way, a component is required to work in *every* environment, ruling out the possibility to model input assumptions.

Models that can encode input assumptions, such as process algebras, often 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]. 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 the semantics of interaction [Abr96]. The reader is referred to [dA03] for a more elaborate comparison with related work.

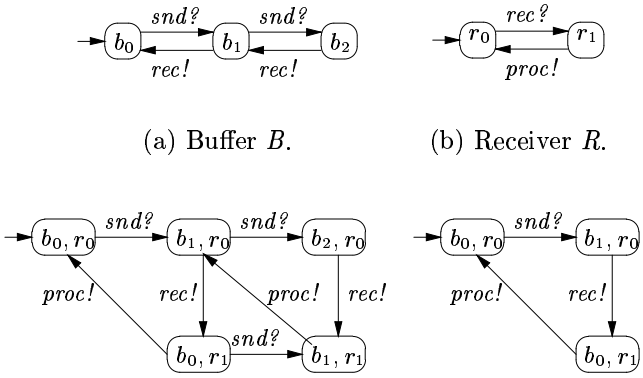
Organization of the paper. This paper treats two automaton-based formalisms for the specification and analysis of interfaces. Section 2 presents interface automata and defines well-formedness, compatibility and composition for these interfaces. In Section 3, we extend interface automata with real-time, 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. 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 2.1 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*.
- $S_P^{init} \subseteq S_P$ 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*.



(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

- $\mathcal{T}_P \subseteq S_P \times \mathcal{A}_P \times S_P$ 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[5], 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 $u = t$.

Input assumptions and output guarantees are expressed via the enabling conditions in P: an input action a that is not enabled in state s, puts a requirement on the environment, asking that a is not produced while P is in s. 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.

Definition 1 A *run* of an interface P is a finite sequence $s_0, a_1, s_1, \ldots, a_n, s_n$ such that $s_{k-1} \xrightarrow{a}_P s_k$ for all $1 \leq k \leq n$. A state $s \in S_P$ is *reachable* if there exists a run from an initial state to s, i.e. a run $s_0, a_1, s_1, \ldots, a_n, s_n$ in P with $s_0 \in S_P^{init}$ and $s_n = s$.

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 *error states*, where the output assumptions of P do not imply the input assumptions of R, or vice versa. That is, 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

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

the input assumptions of $P \otimes R$ in such a way that all error states are avoided, thus ensuring that P and R mutually satisfy their input assumptions.

Composability imposes restrictions on the action sets to avoid name clashes.

Definition 2.2 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 interfaces P and R is an interface automaton $P \otimes R$ that represents the joint behavior of the components. 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. The components synchronize their shared actions in $\mathcal{A}_P \cap \mathcal{A}_R$. This means that whenever one component performs a transition involving a shared action a, the other one should do so; if it cannot, the transition is not part of the product. The components interleave asynchronously all non-shared actions: one component takes a step, while the other stays in the same state.

Definition 2.3 If P and R are composable interface automata, their *product* $P \otimes R$ is the interface automaton defined by

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

Example 2.4 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: input actions that are not enabled in a state encode the assumption that these will not be present in that state. So, if they are produced nevertheless, the assumption is violated. The states in $P \otimes R$ where this happens are called *error states* of P and R.

Definition 2.5 Given two composable interface automata P and R, the set

$Error(P,R)$ of *error states* is defined by

$$Error(P,R) \;=\; \big\{(s,t) \in S_P \times S_R \;\big|\; \exists a \in shared_{P,R}\,.\; (s \xrightarrow{a!}_P \wedge\, t \not\xrightarrow{a?}_R) \vee (t \xrightarrow{a!}_R \wedge\, s \not\xrightarrow{a?}_P)\big\}.$$

Here, $s \xrightarrow{a!}_P$ means that $a \in \mathcal{A}^O_P$ and there exists a $t \in S_P$ with $s \xrightarrow{a}_P t$. Similarly, $s \not\xrightarrow{a?}_P$ means that $a \in \mathcal{A}^I_P$, but there is no $t \in S_P$ with $s \xrightarrow{a}_P t$.

Example 2.6 The state (b_1, r_1) is an error state in the product $B \otimes R$ (Figure 2(c)), because the output step $b_1 \xrightarrow{rec!} b_0$ in B has no corresponding input step in R (i.e. $r_1 \not\xrightarrow{rec?}_R$).

To derive the input assumptions on the composite system that prevent all error states, we observe that error states propagate through the system. For example, if $B \otimes R$ is in the state (b_2, r_0), then, 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; the system decides which of its enabled output actions to take. Hence, the environment should also avoid the state (b_2, r_0), otherwise it does prevent reaching the error state. For the same reasons, the environment should avoid any state from which there is a run that leads to an error state by only following only output transitions. Such states are called *incompatible*.

Definition 2 A state s of $P \otimes R$ is called *incompatible* these exists a run $s_0 a_1 s_1 \ldots s_n$ in $P \otimes R$ such that $s = s_0$, $s_n \in Error(P,P)$ and, for $1 \leq i \leq n$, $a_i \in \mathcal{A}^O_P$. We write $Incmp(P,R)$ for the set of incompatible states and $Cmp(P,R) = S_{P\otimes R} \setminus Incmp(Error(P,P))$ for the set of incompatible ones.

Example 2.7 As we saw before, $Incmp(B,R) = \{(b_1,r_1),(b_2,r_0)\}$.

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 2.8 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 2.9 The interfaces B and R are clearly compatible, as the initial state (b_0, r_0) is not among the incompatible states.

In state (b_0, r_1) on the other hand, the environment can prevent entering error states, viz. by not providing the input *snd?*. Hence, the state (b_0, r_1) does not have to be avoided, but its outgoing *snd?* action should. This is achieved by automatically if we remove incompatible state (b_1, r_1). Thus, strengthening the input assumptions to avoid incompatible states simply proceeds by throwing away all incompatible states. That is, the composition $P \| P$ is obtained by removing the set $Incmp(P,R)$ from the product $P \otimes P$. As a consequence, all edges leading from a compatible state to an incompatible state are removed

as well. Note that all these edges are labeled by an input action. In this way, the input action becomes disabled in the source state of the edge.

Definition 3 Consider 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 2.10 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

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. This means that there should be at least one environment that avoids entering any error state. Let a legal environment of P and R be an interface automaton E such that (1) E is well-formed, (2) E is composable with $P \otimes R$, (3) E synchronizes on every output action of $P \otimes R$, i.e. $\mathcal{A}^I_E = \mathcal{A}^O_{P\otimes R}$, (4) if $(s,t) \in Error(P,R)$, then (u,s,t) is unreachable in $E \otimes P \otimes R$, for every state $u \in S_E$.

Proposition 2.11 *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 legal 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 2.12 *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*. 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 both models differ, because timed automata are interpreted as labeled transition systems and timed interfaces as 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[6] b_0, the buffer is empty and can receive a message. If so, it moves to location b_1, while resetting the clock x. The *output invariant* $x \leq 4$ in location b_1 specifies that an output action (i.c. *snd!*) has to be taken before $x > 4$, thus forcing a delivery within 4 time units. The *transition guard* $1 \leq x$ specifies that the *snd!*-action can be taken if $1 \leq x$, thus enabling a delivery after 1 time unit.

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 with 7 time units after the previous receipt. The transition guard $2 \leq y$ says that this action can be taken after 2 time units. Thus, invariants express when actions must be taken and guards express when they can be taken.

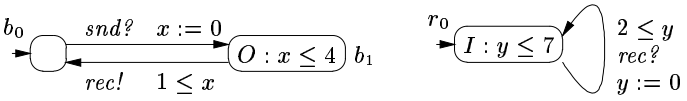
Guards and invariants are specified by clock conditions, being any boolean combination of formulas of the form $x \prec c$ or $x - y \prec c$. Here, c is an integer, x, y are clocks in a given set $\mathcal{X}$, and $\prec$ is either of $<$ or $\leq$. We denote the set of all clock conditions over $\mathcal{X}$ by $K[\mathcal{X}]$.

Note that all time is spent in locations; transitions are instantaneous, i.e. take no time.

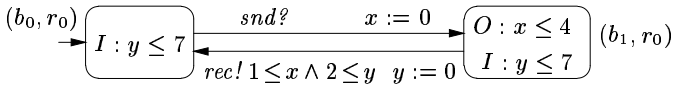
Definition 4 A *timed interface automaton* (or TIA) is a tuple $P = (Q_P, q_P^{init}, \mathcal{X}_P, \mathcal{A}_P^I, \mathcal{A}_P^O, Inv_P^I, Inv_P^O, \mathcal{T}_P)$ consisting of the following components.

- 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*.

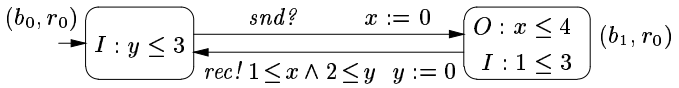
[6] The nodes of timed interface automata are called locations, because the word 'state' already refers to a location together with a clock valuation, see below.



(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

- $\mathcal{A}_P^I$ 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*, 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 $(q, g, a, r, q') \in \mathcal{T}_P$.

Timed versus untimed interfaces. As said before, interfaces are interpreted as games between the players Output and Input, representing the system and the environment, respectively.

Due to the presence of time, timed interfaces and their underlying games have to deal with time divergence. For an interface to be physically meaningful, time should advance in every behavior of the system. Therefore, we call a timed interface *well-formed* if it has an initial state and time can progress in every reachable state. The latter means that both players have a strategy that ensures time progress.

The error states of a timed interface are as in the untimed case, but incompatible states now incorporate time progress requirements. This is because after removing the set of incompatible states from the product, time should

advance in any of the remaining states. Hence, we do not only consider a state incompatible if it can reach an error state by following only output transitions, but also if the other incompatible states can only be avoided by violating the time progress conditions. In this way, incompatibilities propagate through the system: if, by following output transitions only, a state s can reach another state that is incompatible because it has to violate time progress conditions to avoid other incompatible states, then state s is incompatible itself.

This notion of incompatibility is naturally expressed in terms of games. To derive the composition from the product, we are looking for the weakest new input assumptions ensuring that all error states are avoided and time can always progress. This means that, no matter how the system behaves, no state in $\mathit{Error}(P,R)$ is entered and time can always advance. In terms of games: we are looking for the most general input strategy, such that for every output strategy, $\mathit{Error}(P,R)$ is avoided and time progress is ensured. Such a strategy is called a *winning input* strategy for the goal "avoid $\mathit{Error}(P,R)$ and ensure time progress" denoted by $\Box\neg\mathit{Error}(P,R) \cap t\_div$. Winning strategies for such goals can be computed by adapting classical game-theoretic algorithms played on the region graph. States from which the Input player does not have a winning strategy are *incompatible*. Hence, they are removed when computing the composition from the interface.

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 $\mathit{Val}(\mathcal{X})$ 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 $v \models y - x \leq 0$. For a valuation $v \in \mathit{Val}(\mathcal{X})$, 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 $(Q, q^{init}, \mathcal{X}, \mathcal{A}^I, \mathcal{A}^O, \mathit{Inv}^I, \mathit{Inv}^O, \mathcal{T})$. We obtain $[\![P]\!]$ from P as follows. The states (q, v) of $[\![P]\!]$ consist of a location q in P and a clock valuation $v \in \mathit{Val}(\mathcal{X})$. Hence, a state records both the location the interface is in and the values of all its clocks. Initially, all clocks are 0 and the two invariants have to be met. This means that $[\![P]\!]$ has an initial state $(q^{init}, \mathbf{0})$ if $\mathbf{0}$ meets the invariants $\mathit{Inv}^I(q^{init})$ and $\mathit{Inv}^O(q^{init})$ of the initial location q^{init}, (i.e. $\mathbf{0} \models \mathit{Inv}^I(q^{init}) \wedge \mathit{Inv}^O(q^{init})$). If the invariants are not met, then $[\![P]\!]$ 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 $a \in \mathcal{A}$. Here, immediate input transitions $\xrightarrow{a}^I_{[\![P]\!]}$ are labeled by an input action $a \in \mathcal{A}^I_P$, while immediate output transitions $\xrightarrow{a}^O_{[\![P]\!]}$ are labeled by an output action $a \in \mathcal{A}^O_P$. More precisely, let γ is be one of the players I or O. A timed transition $s \xrightarrow{d}^\gamma s'$ represents the passage of d time units. It increases all clocks with d time units and leaves the location unchanged. Hence, the destination s' arises from s by increasing all clocks by d: writing $s = (q, v)$, we have $s' = (q, v + d)$. The transition $s \xrightarrow{d}^\gamma s'$ is enabled if increasing the clocks with d time units is allowed by γ's location invariant $Inv^\gamma(q)$. This means that this invariant has to continuously hold if we let d time units pass: $\forall d' \leq d \,.\, v + d' \models Inv^\gamma(q)$. 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 are set according to the transition involved, that is, v' is obtained from v by setting all clocks in r to 0, i.e. $v' = v[r := 0]$. Summarizing, we have the following.

Definition 5 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 $[\![P]\!]$ are defined by $\langle q, v\rangle \xrightarrow{\alpha}^\gamma_{[\![P]\!]} \langle q', v'\rangle$ if either (1) $\alpha \in \mathbb{R}^{\geq 0}$, $q = q'$, $v' = v + \alpha$, and for all $0 \leq d' \leq \alpha$, we have $v + d' \models Inv^\gamma_P(q)$; or (2) $\alpha \in \mathcal{A}^\gamma_P$, and there is a tuple $(q, g, \alpha, r, q') \in \mathcal{T}_P$ with $v \models Inv^\gamma_P(q) \wedge g$, $v' = v[r := 0]$, and $v' \models Inv^\gamma_P(q')$.

Example 3.1 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 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 deter-

mine 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, an outcome of two moves is a triple (a, γ, s'), where a is the action being taken, γ is the player who took it and s' the destination state.

Definition 6 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 $\delta(s, a)$ for s'. Furthermore, let $\Gamma^{\gamma}_{[\![P]\!]}(s)$ be 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 two 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

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

The behavior of a player, i.e. the successive choices being made in the course of the game, is given by a strategy. This is a function that assigns to every state [7] one of the player's enabled moves. The outcomes that can occur when player I plays according to strategy π^I and O according to π^O, form a set $Outc_{[\![P]\!]}(s, \pi^I, \pi^O)$. Strategies are partial functions, rather than total ones, because the sets of moves available to the players at a state may be empty. As a result, a game outcome may be finite, and if so, it ends in a state where one of the players has no moves.

Definition 7 A *strategy* for player $\gamma \in \{I, O\}$ is a partial function π^γ: $S_{[\![P]\!]} \rightharpoonup \Gamma^{\gamma}_{[\![P]\!]}$ that associates, with every state $s \in S_{[\![P]\!]}$ a move $\pi^\gamma(s) \in \Gamma^{\gamma}_{[\![P]\!]}(s)$ provided that $\Gamma^{\gamma}_{[\![P]\!]}(s) \neq \emptyset$; otherwise $\pi^\gamma(s)$ is undefined. Let $\Pi^{I}_{[\![P]\!]}$ be the set of all strategies for player I, and let $\Pi^{O}_{[\![P]\!]}$ be the set of all strategies for player O. 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 (1) $s_0 = s$; (2) for all $n < length(\sigma)$, we have $(\alpha_{n+1}, s_{n+1}, pl_{n+1}) \in outc_{[\![P]\!]}(s_n, \pi^I(s_n), \pi^O(s_n))$, where such that either $\Gamma^{I}_{[\![P]\!]}(s_k) = \emptyset$ or $\Gamma^{O}_{[\![P]\!]}(s_k) = \emptyset$; and (3) if $length(\sigma) < \infty$, then σ ends in a pair (s_k, pl_k).

Informally, a state is reachable if it can be reached from the initial state

[7] In general, a strategy may depend on the history of the game, that is, on the entire run leading to the state. In our case history-free strategies suffice.

via a pair of strategies.

Definition 8 A state s 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 which 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, viz 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 playing alone from a certain point on.

Definition 9 For a TIA P, we define

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

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 make physically sense. Behaviors such as $s, 0, I, s, 0, O, s, 0, I, s, 0, O, s \ldots$ and $s, \frac{1}{2}, I, s, \frac{1}{4}, O, s, \frac{1}{8}, I, s, \ldots$ 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 exist in reality, because an infinite number of moves is made in a finite amount of time. We rule out such behaviors by only considering well-formed states, in which both players have the possibility to let time diverge. We want to say that a state s is called *well-formed* if from s both players have a strategy to win the goal t_div. However, the situation is slightly more complex because one of the players may "monopolize" the game, by playing alone from a certain point on. In such a situation, the opponent is not responsible if time stops. Therefore, we consider all behaviors in which this happens to be winning for the opponent, irrespective of time divergence. Thus, a state is *well-formed* if both players γ can win the goal $t\_div \cup blame^{1-\gamma}$. A timed automaton 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 10 A state $s \in S_{[\![P]\!]}$ is *well-formed* if (1) 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 (2) Output can win the game with goal $t\_div \cup blame^I$; that is, if there is a strategy $\pi^O \in \Pi^O_{[\![P]\!]}$ such that for all strategies $\pi^I \in \Pi^I_{[\![P]\!]}$

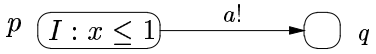


Fig. 4. An ill-formed TIA

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, i.e. $S^{init}_{[\![P]\!]} \neq \emptyset$, 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 use it to determine its own move. The motivation for this is discussed in [dAHS02].

Example 3.2 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. Such an interface does not make sense, because no environment meets 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 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 is defined via the notions of composability, product, error states, uncontrollable states.

Definition 11 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 others. 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 invariants. The output invariants in s and t are conjoined as well. Two synchronizing 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 12 For 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 $q^{init}_{P_1 \otimes P_2} = (q^{init}_{P_1}, q^{init}_{P_2})$.
- $\mathcal{X}_{P_1 \otimes P_2} = \mathcal{X}_{P_1} \cup \mathcal{X}_{P_2}$.
- $\mathcal{A}^I_{P_1 \otimes P_2} = \mathcal{A}^I_{P_1} \cup \mathcal{A}^I_{P_2} \setminus shared_{P_1,P_2}$, and $\mathcal{A}^O_{P_1 \otimes P_2} = \mathcal{A}^O_{P_1} \cup \mathcal{A}^O_{P_2}$.

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

Example 3.3 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 error states: immediate error states and timed error states. *Immediate error* states are as in the untimed case, where one component can preform an output action that is not accepted by the other component.

Definition 3.4 Given two composable interface automata P and R, the set $Error(P,R)$ of *immediate error states* is defined by

$$Error(P,R) = \big\{(v,u) \in S_{[\![P]\!]} \times S_{[\![R]\!]} \;\big|\; \exists a \in shared_{P,R} \,. \; (v \xrightarrow{a!}_{[\![P]\!]} \wedge\, u \not\xrightarrow{a?}_{[\![R]\!]}) \vee (u \xrightarrow{a!}_{[\![R]\!]} \wedge\, v \not\xrightarrow{a?}_{[\![P]\!]})\big\}.$$

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

Example 3.5 For example, the state $(b_1, r_0, \{x = 1, y = 1\})$ is an immediate error state in $B \otimes R$, because $(b_1, \{x = 1\}) \xrightarrow{rec!}_{[\![B]\!]}$, but $(r_0, \{y = 1\}) \not\xrightarrow{rec?}_{[\![R]\!]}$. The state $(b_1, r_0, \{x = 3, y = 3\})$ is a good state, because $(b_1, \{x = 3\}) \xrightarrow{rec!}_{[\![B]\!]}$ and $(r_0, \{y = 3\}) \xrightarrow{rec?}_{[\![R]\!]}$.

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

Timed error states are 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 timed error 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, $\langle (b_1, r_0), \{x = 0, y = 3.4\}\rangle$ a timed error state. Timed error states are simply the ill-formed states in the product.

However, also when avoiding immediate error states, the input player has to let time diverge: it should not avoid those states by blocking time. Thus, a state s in the product is compatible if the input player has a strategy that,

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

at the same time, avoids the immediate error states and lets time pass (or blames the Output player). That is, an Input strategy that wins the goal $Good(P,R) \cap (t\_div \cup blame^O)$.

Definition 13 A state s in $[\![P \otimes R]\!]$ *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 $\sigma \in Outc_{[\![P]\!]}(s, \pi^I, \pi^O)$ satisfy $\sigma \models \Box Good(P,R) \cap (t\_div \cup blame^O)$. States in $S_{P\otimes R}$ that are not compatible are called *incompatible*.

Example 3.6 Note that states in which one of the invariants is violated are always error states. By reasoning as before, 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 the goal $\Box Good(P,R) \cap (t\_div \cup blame^O)$ is expressible as a clock condition, which we denote by $Compat_{P\otimes R}(q)$. The clock conditions $Compat_{P\otimes R}(q)$ can be computed with the game theoretical algorithms discussed in [dAHS02].

Example 3.7 Example 3.6 shows $Compat_{B\otimes R}(b_0, r_0) = Compat_{B\otimes R}(b_1, r_0) = y \leq 3$.

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

Definition 14 The *composition* $P\|R$ is obtained from the product $P \otimes R$ by replacing for each location $q \in Q_{P\otimes R}$ the input invariant $Inv^I_{P\otimes R}(q)$ with $Compat_{P\otimes R}(q)$.

Example 3.8 Substituting the input invariants in $B \otimes R$ yields 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 1 *Let P and R be composible TIAs. If P and R compatible and well-formed, then $P\|R$ is well-formed as well.*

Corollary 3.9 *Let P and R be composible 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 identical once we remove all unreachable states. Then interface composition is associative upto $\equiv$.

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

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.

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

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, pages 7–48, 1999.

[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. Stoelinga. Resource interfaces. In *Proceedings of the Third International Workshop on Embedded Software (EMSOFT 2003)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2003.

[CMCHG96] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *CAV 96: Proc. of 8th Conf. on Computer Aided Verification*, Lect. Notes in Comp. Sci., pages 419–422. Springer-Verlag, 1996.

[dA03] 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.

[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. Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT 2002)*, Lect. Notes in Comp. Sci. Springer-Verlag, 2002.

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

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[MMT91] M. Merritt, F. Modugno, and M. Tuttle. Time constrained automata. In *CONCUR'91: Concurrency Theory. 2nd Int. Conf.*, volume 527 of *Lect. Notes in Comp. Sci.*, pages 408–423. Springer-Verlag, 1991.

[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.

[SGSAL98] R. Segala, G. Gawlick, J. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, 1998.