# Composing Real-Time Concurrent Objects
## Refinement, Compatibility and Schedulability*

Mohammad Mahdi Jaghoori

LIACS, Leiden, The Netherlands
CWI, Amsterdam, The Netherlands
jaghouri@cwi.nl

**Abstract.** Concurrent objects encapsulate a processor each and communicate by asynchronous message passing; therefore, they can be composed to naturally model distributed and embedded systems. We model real-time concurrent objects using timed automata and provide each object with a context-specific scheduling policy. The envisioned usage and guaranteed deadlines of each object is specified in its behavioral interface, given also in timed automata. Furthermore, multiple objects can be composed only if they are compatible, i.e., if they respect the expected use patterns given in the behavioral interfaces of each other. In this paper, we define refinement of timed automata with inputs and outputs from a new perspective and we take account of deadlines in the refinement theory. Within this framework, we study composition and compatibility of real-time concurrent objects, and apply it in the context of compositional schedulability analysis of multiple-processor systems.

## 1 Introduction

Object oriented paradigm is a good basis for modular modeling and compositional analysis. A distributed system can be modeled as the composition of a set of concurrent objects where each concurrent object conceptually has a dedicated processor. We use timed I/O automata to model the real-time behavior of concurrent objects at an abstract level, as in our previous work [9]. Automata theory provides a rich basis for analysis; nevertheless, we need compositional techniques to overcome the complexity of large asynchronous distributed systems. A concurrent object is both the unit of concurrency and distribution; it is also a natural point for compositional analysis.

In this paper, we aim at compositional schedulability analysis of multiple-processor distributed systems specified with concurrent objects; a real-time system is schedulable if it can finish all of its tasks within their deadlines. While an object comprises a queue, a scheduling policy and several methods and is thus modeled in several automata, the abstract behavior of the object is given in one automaton, called the behavioral interface. A behavioral interface specifies at a high level and in the most general terms how an object may be used; thus it is

* This work is supported by the EU FP7-231620 project: HATS.

used as the key to compositional analysis. Each object is analyzed individually for schedulability with respect to its behavioral interface. As in modular verification [12], which is based on assume-guarantee reasoning, individually schedulable objects can be used in systems *compatible* with their behavioral interfaces. The schedulability of such systems is then guaranteed [10].

In interface-based design, refinement is usually used as the means for compositional analysis. Given a set of components $C_j$ with interfaces $I_j$, $C_j$ is considered a correct implementation if it refines $I_j$. Then ideally, when the interfaces are *compatible* their implementations should also be able to work together. To capture all incompatibilities, any behavior not allowed in the interfaces should lead to an error (e.g., [6], cf. related work); however, this is too restrictive in practice because interfaces are abstract and easily produce spurious counterexamples to compatibility. An optimistic approach (e.g., [1]) considers two interfaces compatible if there exists a common behavior that allows them to work together. This is useful if we can make sure the implementation of evey components indeed follows this common behavior. We formalized this last step in [10] by requiring the composition of the components $C =\|_j C_j$ to be a refinement of $I =\|_j I_j$.

In this paper, we give a compositional solution to checking the refinement between $C =\|_j C_j$ and $I =\|_j I_j$. The idea is that the outputs of each component $C_j$ should be expected as an input by the interface of the receiving component; this is formalized as every $C_j$ being a refinement of $I$. Traditional views on refinement do not allow this relation because $C_j$ and $I$ have incompatible sets of inputs and outputs. A contribution of this paper is generalizing refinement such that it considers the common set of actions as the observable behavior. Thus, $I$ is comparable to each $C_j$ with respect to the inputs and outputs of $C_j$.

The second contribution of the paper is adding deadlines, as parameters to actions, to the refinement theory. A deadline on an output specifies when the task is required to finish. A deadline on an input specifies the guaranteed time before which the task is finished. Usually parameters are not included in the theory of refinement; instead, they are handled by expansion, i.e., an action is expanded to several actions considering different valuations of the parameter. Deadline parameters cannot be treated by expanding. A component may require weaker deadlines than its interface on the outputs and provide stronger guarantees for the inputs. We redefine refinement giving deadlines this special treatment.

Another contribution of this paper is applying the developed refinement theory in checking compatibility of concurrent objects in a compositional way. In [10], we have defined compatibility in terms of refinement: a closed system made up of individually schedulable objects is schedulable if it is a refinement of the composition of the behavioral interfaces. With our general definition of refinement, we can apply our method in *open systems* of multiple concurrent objects, too. The behavioral interface of the composite open system is the composition of the behavioral interfaces of individual objects.

We will explain how to automate refinement checking in the tool Uppaal [13]. We show further how to check schedulability and compatibility in Uppaal.

**Related Work.** Compatibility of real-time systems in automata theory has been studied for timed interfaces [1] and timed I/O automata [6]. Alfaro et al. [1] take an optimistic approach in which two interfaces are compatible if there is a possible way for them to work properly. This leads to a simpler theory but to implement these interfaces, one needs to adhere to these possibilities to end up with a working system. David et al. [6] suggest to make specifications input-enabled by adding an Error state and directing every undesired behavior to that state. They define two specifications to be compatible if their composition does not reach the Error state. This is unfortunately too restrictive for high-level specifications; abstract behavioral interfaces easily fall into spurious incompatibilities whereas their implementations may still work together. Our approach bridges the gap between these two methods. In fact, we check whether the implementations at hand, when composed, indeed follow the behavior that makes their interfaces compatible (w.r.t. the optimistic approach of [1]).

Analyzing the composition of the concurrent objects is subject to state space explosion because of their asynchronous nature and all their queues. We proposed a testing technique for compatibility in [10]. In present paper, we will model check compatibility in a compositional way with our generalized refinement theory.

Schedulability has been studied for actor languages [15] and event driven distributed systems [8]. Unlike these works, we work with non-uniformly recurring tasks as in task automata [7] which fits better the nature of message passing in object-oriented languages. The advantage of our work over task automata is that tasks are specified and may in turn create new tasks. Furthermore, we address schedulability analysis of multiple-processor systems. Compared to [11] we deal with the problem in a compositional way.

A characteristic of our work is modularity. A behavioral interface models the most general message arrival pattern for an object. A behavioral interface can be viewed as a contract as in 'design by contract' [14] or as a most general assumption in modular model checking [12] (based on assume-guarantee reasoning); schedulability is guaranteed if the real use of the object satisfies this assumption. In the literature, a model of the environment is usually the task generation scheme in a specific situation. For example in TAXYS [4], different models of the environment can be used to check schedulability of the application in different situations. However, a behavioral interface in our analysis covers all allowable usages of the object, and is thus an over-approximation of all environments in which the object can be used. This adds to the modularity of our approach; every use of the object foreseen in the interface is verified to be schedulable.

## 2 Timed Automata

Suppose $\mathcal{B}(C)$ is the set of all clock constraints on the set of clocks $C$. A *timed automaton*, as defined by Alur and Dill [2], over actions $\Sigma$ and clocks $C$ is a tuple $\langle L, l_0, \longrightarrow, I\rangle$ representing

- a finite set of locations $L$ (including an initial location $l_0$);
- the set of edges $\longrightarrow \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$; and,

– a function $I : L \mapsto \mathcal{B}(C)$ assigning an invariant to each location.

An edge $(l, g, a, r, l')$ implies that action '$a$' may change the location $l$ to $l'$ by resetting the clocks in $r$, if the clock constraints in $g$ (as well as the invariant of $l'$) hold. When we use UPPAAL [13] for analysis, we allow defining variables of type boolean and bounded integers. Variables can appear in guards and updates.

A timed automaton is called *deterministic* if and only if for each $a \in \Sigma$, if there are two edges $(l, g, a, r, l')$ and $(l, g', a, r', l'')$ from $l$ labeled by the same action $a$ then the guards $g$ and $g'$ are disjoint (i.e., $g \wedge g'$ is unsatisfiable).

**Semantics** A timed automaton defines an infinite labeled transition system whose states are pairs $(l, u)$ where $l \in L$ and $u : C \to \mathbb{R}_+$ is a *clock assignment*. We denote by $\mathbf{0}$ the assignment mapping every clock in $C$ to 0. The initial state is $s_0 = (l_0, \mathbf{0})$. There are two types of transitions from a given state $(l, u)$:

– action transitions $(l, u) \xrightarrow{a} (l', u')$ where $a \in \Sigma$, if there exists $(l, g, a, r, l')$ such that $u$ satisfies the guard $g$, $u'$ is obtained by resetting all clocks in $r$ and leaving the others unchanged and $u'$ satisfies the invariant of $l'$;
– delay transitions $(l, u) \xrightarrow{d} (l, u')$ where $d \in \mathbb{R}_+$, if $u'$ is obtained by delaying every clock for $d$ time units and for each $0 \leq d' \leq d$, $u'$ satisfies the invariant of location $l$.

**Refinement** In general refinement is decidable for deterministic timed automata [2]. Traditionally, two timed automata are considered comparable if they have the same set of (observable) actions. We do not define observable and hidden actions explicitly for timed automata. Given two timed automata $A$ and $B$, we consider them comparable if their common set of actions $\Sigma_A \cap \Sigma_B$ is not empty. A meaningful use of this definition is when there is a meaningful relation between the sets of actions of the two automata.

**Definition 1 (TA Refinement).** *Given two timed automata $A$ and $B$, we say $A$ refines $B$ (written $A \sqsubseteq B$) iff there is a relation $R$ between their underlying transition systems such that $(s_0^A, s_0^B) \in R$, and if $(s, t) \in R$ then*

– *for $a \in \Sigma_A \cap \Sigma_B$, if $s \xrightarrow{a}_A s'$ then $t \xrightarrow{a}_B t'$ and $(s', t') \in R$;*
– *if $A$ can delay: $s \xrightarrow{d}_A s'$, then $B$ can also delay: $t \xrightarrow{d}_B t'$ and $(s', t') \in R$.*

## 3 Timed I/O Automata

In timed I/O automata, the action set $\Sigma$ is partitioned into inputs ($\Sigma^I$), outputs ($\Sigma^O$) and internal actions ($\Sigma^\tau$). This allows us to model different components of a real-time system using automata while their communication is modeled by synchronization on matching input and output actions. Internal actions are not necessarily hidden; as we will see in composition, internal actions may model internal communication which may still be observable, i.e., included in refinement checking.

We consider timed I/O automata as a superclass of timed automata such that in normal timed automata $\Sigma^I = \Sigma^O = \emptyset$. For an action $a$ we will write $a!$, $a?$ and $a$ to denote that it is treated as an output, input or internal action, respectively.

**Composition** Composition of two timed I/O automata $A$ and $B$ is written as $S = A \parallel B$. The set of locations of $S$ is the Cartesian product of those of $A$ and $B$, denoted $L(S) = L(A) \times L(B)$. The composed automata synchronize on the set of sync actions $\Sigma_\cap = (\Sigma^I_A \cap \Sigma^O_B) \cup (\Sigma^O_A \cap \Sigma^I_B)$, which are made internal in $S$:

- input actions: $\Sigma^I_S = (\Sigma^I_A \cup \Sigma^I_B) \setminus \Sigma_\cap$
- output actions: $\Sigma^O_S = (\Sigma^O_A \cup \Sigma^O_B) \setminus \Sigma_\cap$
- internal actions: $\Sigma^\tau_S = \Sigma^\tau_A \cup \Sigma^\tau_B \cup \Sigma_\cap$

The set of transitions of $S$ is computed as follows:

- $(u, v) \xrightarrow{g,a,r}_S (u', v)$ when $u \xrightarrow{g,a,r}_A u'$ and $a \notin \Sigma_\cap$
- $(u, v) \xrightarrow{g,a,r}_S (u, v')$ when $v \xrightarrow{g,a,r}_B v'$ and $a \notin \Sigma_\cap$
- $(u, v) \xrightarrow{g \wedge g', a, r \wedge r'}_S (u', v')$ when $u \xrightarrow{g,a,r}_A u'$ and $v \xrightarrow{g',a,r'}_B v'$ and $a \in \Sigma_\cap$

where by $r \wedge r'$ we mean updating both $r$ and $r'$. Semantically, $S$ can delay if both $A$ and $B$ can delay; $S$ can perform a sync action $a \in \Sigma_\cap$ if both $A$ and $B$ can perform $a$; $S$ can do any other action if either $A$ or $B$ can do that action.

For a finite set of timed I/O automata $A_i$ $(1 \leq i \leq n)$ to be *composable*, they should have disjoint observable actions: $\forall_{1 \leq i,j \leq n}\ \Sigma^I_{A_i} \cap \Sigma^I_{A_j} = \Sigma^O_{A_i} \cap \Sigma^O_{A_j} = \emptyset$. In this case, composition is associative, i.e., $(A \parallel B) \parallel C = A \parallel (B \parallel C)$. Thus, one could simply write $S = A_1 \parallel \cdots \parallel A_n$ to describe the composition of several composable timed I/O automata communicating with each other. The composition is called a *closed* system when $\Sigma^I_S = \Sigma^O_S = \emptyset$.

### 3.1 Refinement for Timed I/O Automata

A recent work by David et al. [6] gives a game-theoretic solution for checking refinement of timed I/O automata, but they assume input-enabled specifications. Our definition of refinement for timed automata and timed I/O automata does not assume input-enabledness; this leads to a more precise notion of compatibility (cf. [1, 6]). This is more practical and will be used in Section 6 for schedulability analysis.

Two timed I/O automata $A$ and $B$ are traditionally (e.g., in [6]) considered comparable if they have the same sets of input and output actions, i.e., $\Sigma^I_A = \Sigma^I_B$ and $\Sigma^O_A = \Sigma^O_B$. Since we want to consider refinement in the context of composition where inputs and outputs may need to be compared to internal actions (which are in turn the result of synchronization), we need to be more liberal with the relation of the action sets. We say the timed I/O automata $A$ and $B$ are comparable for the relation $A \sqsubseteq B$ if:

$$\Sigma^I_A \subseteq \Sigma^I_B \cup \Sigma^\tau_B \quad \wedge \quad \Sigma^O_A \subseteq \Sigma^O_B \cup \Sigma^\tau_B \quad \wedge \quad \Sigma^\tau_A \cap (\Sigma^I_B \cup \Sigma^O_B) = \emptyset$$

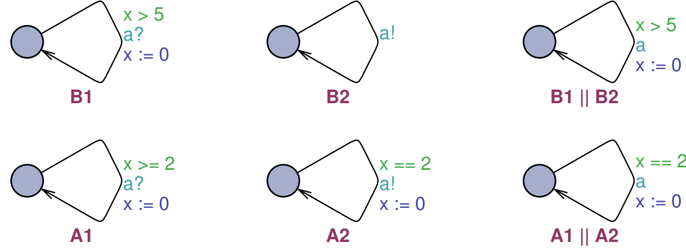



**Fig. 1.** Composition and refinement: $A_1 \sqsubseteq B_1$ and $A_2 \sqsubseteq B_2$ but $A_1 \parallel A_2 \not\sqsubseteq B_1 \parallel B_2$. This result is expected because $A_1 \sqsubseteq B_1 \parallel B_2$ but $A_2 \not\sqsubseteq B_1 \parallel B_2$.

In refinement between timed I/O automata, inputs and outputs are treated differently, as in alternating refinement [3]. Intuitively, when $A$ refines $B$, the refined model $A$ must accept any input that is acceptable in $B$; and, $A$ may produce an output only if it is allowed at the abstract level $B$ (e.g., see Fig. 1).

As timed IO automata are used in component-based design, we would like to be able to use them in compositional analysis, too. Given $A_1 \sqsubseteq B_1$ and $A_2 \sqsubseteq B_2$, $B_1$ and $B_2$ could be abstract interfaces for components $A_1$ and $A_2$, and one might expect that $A_1 \parallel A_2 \sqsubseteq B_1 \parallel B_2$. Such a compositional reasoning does not hold for timed I/O automata; this is illustrated in the counterexample in Fig. 1. In this example, $A_1$ admits the input $a?$ in a bigger time interval than $B_1$; it becomes an internal action after synchronization with $a!$ in $A_2$ (cf. composition in Section 3), this action would exist in $A_1 \parallel A_2$ but not necessarily in $B_1 \parallel B_2$. To compositionally infer $A_1 \parallel A_2 \sqsubseteq B_1 \parallel B_2$, we suggest an extra sufficient step $A_1 \sqsubseteq B_1 \parallel B_2$ and $A_2 \sqsubseteq B_1 \parallel B_2$. Here, we give a definition for refinement of timed I/O automata that supports such compositional analysis.

**Definition 2 (TIOA Refinement).** *Given two comparable timed I/O automata $A$ and $B$, we say $A$ refines $B$ iff there is a relation $R$ between their underlying transition systems such that $(s_0^A, s_0^B) \in R$, and if $(s,t) \in R$*

- *for $a \in \Sigma_A^I$, if $t \xrightarrow{a}_B t'$, then $s \xrightarrow{a}_A s'$ and $(s', t') \in R$;*
- *for $a \in \Sigma_A^O \cup (\Sigma_A^\tau \cap \Sigma_B)$, if $s \xrightarrow{a}_A s'$, then $t \xrightarrow{a}_B t'$ and $(s', t') \in R$;*
- *if $A$ can delay: $s \xrightarrow{d}_A s'$ then $B$ can also delay: $t \xrightarrow{d}_B t'$ and $(s', t') \in R$.*

It is easy to see that when $A$ and $B$ are normal timed automata, i.e., the input and output action sets are empty, Def. 2 simplifies to Def. 1. Furthermore, we do not require any direct relation between the inputs (resp. outputs) of $A$ and $B$; we may compare inputs or outputs of $A$ with internal actions of $B$. Thus we can compare arbitrary automata which helps us check refinement in a compositional way, described below.

**Theorem 1.** *Given the timed I/O automata $A_1$, $A_2$ and $B$, we have:*

$$A_1 \sqsubseteq B \;\wedge\; A_2 \sqsubseteq B \implies A_1 \parallel A_2 \sqsubseteq B$$

The essence of the proof is to consider synchronization of an output in $A_1$ with an input in $A_2$. To show that the resulting transition (with an internal action) exists in $B$, we take the corresponding output in $A_1$. In other cases, the refinement is in fact straightforward because an action in $A$ is due to the same action in $A_1$ or $A_2$.

In the example in Figure 1, we can make $A_2$ to be a refinement of $B_1 \parallel B_2$ by changing the guard on $a!$, for example, to $x == 6$. It is easy to see that in this case $A_1 \parallel A_2$ is also a refinement of $B_1 \parallel B_2$.

**Corollary 1.** *Given a finite set of timed I/O automata $A_i$ $(1 \leq i \leq n)$ and $B$, we have:*

$$A_1 \sqsubseteq B \ \wedge \ \ldots \ \wedge \ A_n \sqsubseteq B \implies A_1 \parallel \cdots \parallel A_n \sqsubseteq B$$

In a component-based design where different components $A_i$ implement the behavioral interfaces $B_i$, this corollary helps us check the refinement relation $A \sqsubseteq B$ in a compositional way, where $A = A_1 \parallel \cdots \parallel A_n$ and $B = B_1 \parallel \cdots \parallel B_n$. Having checked this refinement, one could prove safety properties at the abstract level for $B$, which then carries over to the refined and more complex system $A$. In Section 6 we use this approach for compositional schedulability analysis of a multiple processor system modeled in concurrent objects.

## 4 Timed I/O Automata with Deadlines

A deadline specifies the time before which a task must be done. A common property to check for real-time systems is schedulability, i.e., whether all tasks finish within their deadlines. We associate a *relative* deadline $d \in \mathbb{N}$ to input and output actions, i.e., the deadline is $d$ time units after the action is taken. The interpretation of a deadline depends on the action type:

- An automaton with an input action $a(d)?$ guarantees the deadline $d$; therefore, it naturally also guarantees $d+1$.
- An output action $a(d)!$ requires a deadline $d$; naturally, a deadline $d$ is a stronger requirement than $d+1$.

At the lowest level of abstraction, the tasks are implemented and one needs to check whether they indeed meet their deadlines, as explained in Section 5.

*Composition* In presence of deadlines, we restrict composition of timed I/O automata by allowing only compatible actions to synchronize; two actions $a(d)?$ and $a(d')!$ are compatible if $d \leq d'$, i.e., the required deadline is not stronger than the guaranteed one. As a result of this synchronization, the composed automaton will have an internal action $a(d, d')$. A deadline interval $[d..d']$ associated to an internal action $a$ is stronger than $[\delta..\delta']$ if the interval $[d..d']$ is included in $[\delta..\delta']$, i.e., $\delta \leq d$ and $d' \leq \delta'$. When two transitions have a sync action as input and output with incompatible deadlines, they do not synchronize and they do not appear in the composed automaton.

*Refinement* When considering deadlines in refinement, the refined model must provide the same (or stronger) deadline guarantees on its inputs compared to the abstract model; the refined model may not require stronger deadlines on its outputs than the abstract model. A common internal action cannot have a stronger deadline interval than the abstract one. Below, Def. 2 is extended to include deadlines with the abovementioned considerations. Taking deadline intervals for internal actions makes this definition of refinement transitive, i.e., given $A \sqsubseteq B$ and $B \sqsubseteq C$ we have $A \sqsubseteq C$.

**Definition 3 (Refinement with Deadlines).** *Given two comparable timed I/O automata, we say A refines B iff there is a relation R between their underlying transition systems such that $(s_0^A, s_0^B) \in R$, and if $(s,t) \in R$*

- *for $a \in \Sigma_A^I$, if $t \xrightarrow{a(d)?}_B t'$ then $s \xrightarrow{a(\delta)?}_A s'$ with $d \geq \delta$ and $(s',t') \in R$;*
- *for $a \in \Sigma_A^I$, if $t \xrightarrow{a(d,d')}_B t'$ then $s \xrightarrow{a(\delta)?}_A s'$ with $d \geq \delta$ and $(s',t') \in R$;*
- *for $a \in \Sigma_A^O$, if $s \xrightarrow{a(\delta)!}_A s'$ then*
  - *$t \xrightarrow{a(d)!}_B t'$ with $d \leq \delta$ and $(s',t') \in R$; or,*
  - *$t \xrightarrow{a(d,d')}_B t'$ with $d' \leq \delta$ and $(s',t') \in R$;*
- *for $a \in \Sigma_A^\tau \cap \Sigma_B$, if $s \xrightarrow{a(\delta,\delta')}_A s'$ then $t \xrightarrow{a(d,d')}_B t'$ and $\delta \leq d$ and $d' \leq \delta'$ and $(s',t') \in R$;*
- *if A can delay: $s \xrightarrow{d}_A s'$ then B can also delay: $t \xrightarrow{d}_B t'$ and $(s',t') \in R$.*

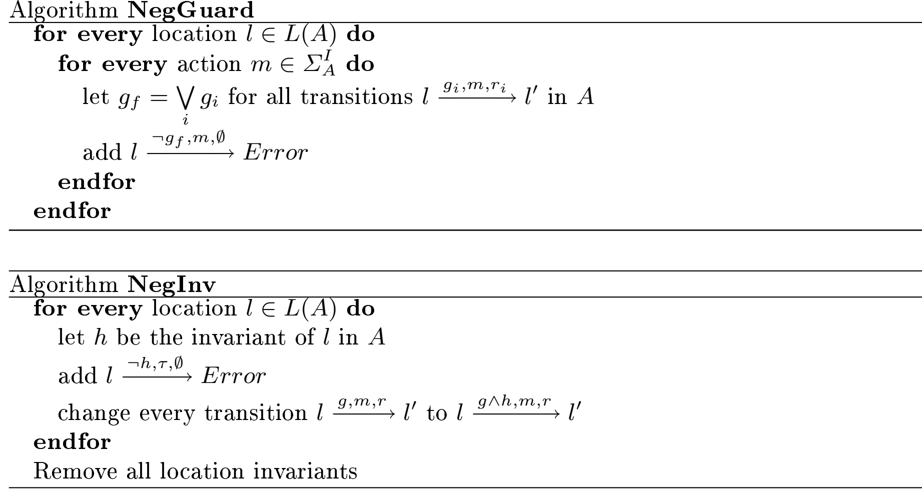
One can easily extend the proof of Theorem 1 to include deadlines. To do so, consider again the case when an input with deadline $d$ synchronizes with an output with deadline $d'$. The generated internal action has the deadline interval $(d, d')$. Considering the definition of refinement, we can easily show that the corresponding interval in $B$ is stronger than $(d, d')$.

### 4.1 Checking Refinement in Uppaal

It has been shown for timed automata that checking refinement $A \sqsubseteq B$ is decidable when $B$ is deterministic [2]. For input-enabled timed I/O automata, David et al. [6] use a game-theoretic approach. We gave in [10] a simple algorithm to test refinement of timed automata, in the flavor of Def. 1, using reachability analysis in Uppaal. Below, we show how to check refinement for timed I/O automata with deadlines (cf. Def. 2) again using reachability analysis in Uppaal.

To check the refinement relation $A \sqsubseteq B$, first, we assume no deadlines in checking refinement (cf. Def. 2). We start from $A^*$ and $B^*$ being copies of $A$ and $B$, respectively, and continue as below:

- First, we repartition the action sets: $\Sigma_{A^*}^I = \Sigma_{B^*}^O = \Sigma_A^I$ and $\Sigma_{A^*}^O = \Sigma_{B^*}^I = \Sigma_A^O \cup (\Sigma_A^\tau \cap \Sigma_B)$; other actions are treated as internal. Considering the requirements that make $A$ and $B$ comparable for the relation $A \sqsubseteq B$ (cf. Section 3.1), it is easy to see that the assignments above do not change the action set of $B^*$, i.e., $\Sigma_B = \Sigma_{B^*}$.



Algorithm **NegGuard**

**for every** location $l \in L(A)$ **do**
  **for every** action $m \in \Sigma_A^I$ **do**
    let $g_f = \bigvee_i g_i$ for all transitions $l \xrightarrow{g_i, m, r_i} l'$ in $A$
    add $l \xrightarrow{\neg g_f, m, \emptyset} Error$
  **endfor**
**endfor**

Algorithm **NegInv**

**for every** location $l \in L(A)$ **do**
  let $h$ be the invariant of $l$ in $A$
  add $l \xrightarrow{\neg h, \tau, \emptyset} Error$
  change every transition $l \xrightarrow{g,m,r} l'$ to $l \xrightarrow{g \wedge h, m, r} l'$
**endfor**
Remove all location invariants

**Fig. 2.** Adding transitions to the Error Location to find incompatibilities.

– We add an Error location to each of $A^*$ and $B^*$.

Next, with the algorithms in Fig. 2, we produce $A^*_E = \text{NegGuard}(A^*)$ and $B^*_E = \text{NegGuard}(\text{NegInv}(B^*))$. $A^*_E$ and $B^*_E$ basically have the same behavior as $A^*$ and $B^*$ but they are made input-enabled such that every incompatible action in $A$ or $B$ leads to a designated Error location, i.e., intuitively, unexpected inputs in $A$ and unexpected output or delay in $B$. Finally, $A^*_E$ and $B^*_E$ have matching inputs and outputs and thus their composition can be analyzed in a tool like Uppaal. The Error locations of $A^*_E$ and $B^*_E$ is not reachable iff $A \sqsubseteq B$. To sketch the proof of this, NegInv and NegGuard help detect the possible incompatibilities between delay and action transitions, respectively, with respect to Def. 2.

In order to consider deadlines in Uppaal, we add an extra step in computing $A^*$ and $B^*$:

– For every $a \in \Sigma_A^I$, we change $a(d)$ to $a(d, 0)$ in both $A^*$ and $B^*$.
– For every $a \in \Sigma_A^O$, we change $a(d)$ to $a(0, d)$ in both $A^*$ and $B^*$.

Obviously, the above rules leave the internal actions of $B$ (which already have the form $a(d, d')$) unchanged. Thus, every input and output action of $A^*$ and $B^*$ has two deadline values. Next, we add two fresh global variables $\delta$ and $\delta'$; in Uppaal, global variables are used to pass parameter values. We transform input and output actions to Uppaal format as follows:

– We change every output transition $s \xrightarrow{g, a(d,d')!, r}_{A^*} s'$ to $s \xrightarrow{g, a!, r \wedge \delta := d \wedge \delta' := d'}_{A^*} s'$.
– We change every output transition $s \xrightarrow{g, a(d,d')!, r}_{B^*} s'$ to $s \xrightarrow{g, a!, r \wedge \delta := d}_{B^*} s'$.
– We change every input transition $s \xrightarrow{g, a(d,d')?, r}_{A^*} s'$ to $s \xrightarrow{g \wedge d \leq \delta, a?, r}_{A^*} s'$.
– We change every input transition $s \xrightarrow{g, a(d,d')?, r}_{B^*} s'$ to $s \xrightarrow{g \wedge d \leq \delta \wedge \delta' \leq d', a?, r}_{B^*} s'$.

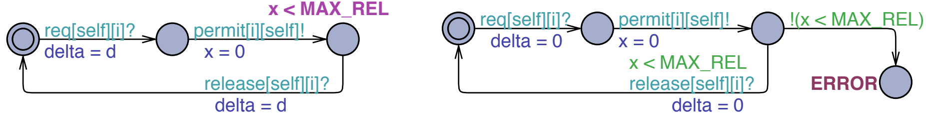



**Fig. 3.** The abstract behavioral interface of a resource and applying NegInv on it.

The outputs set the values of $\delta$ and $\delta'$ to their deadline values which are checked against the input deadline guarantees in the input actions. This way the deadline values and the corresponding checks are integrated into the guards and updates of the automata. Then we can continue in the same way as explained above without deadlines, i.e., compute $A_E^*$ and $B_E^*$ and check for the reachability of the Error locations. An example of computing $B_E^*$ is given in the next section.

## 5 Real-Time Concurrent Objects

Concurrent objects encapsulate a processor each and communicate by asynchronous message passing; therefore, they can be composed to naturally model distributed and embedded systems. An object is an instance of a class with a context-specific scheduler; a class implements a behavioral interface. In this section, we use timed I/O automata with deadlines to model behavioral interfaces, classes and schedulers.

The observable actions of concurrent objects are the messages they communicate. For their automata models to be composable, they should have disjoint sets of inputs (resp. outputs). To achieve this, we consider an action to be a triple $(m, r, s)$ where $m$ is the message name, $r$ is the receiver object identity, and $s$ is the sender object identity. The keyword *self* refers to the identity of the owner object itself.

**Behavioral interfaces** A behavioral interface provides an abstract overview of the object behavior in a single automaton in terms of the messages it may receive and send. We assume a finite global set $\mathcal{M}$ for method names; sending and receiving messages are written as $m!$ and $m?$, respectively. We use natural values $d \in \mathbb{N}$ to represent deadlines. A behavioral interface $B$ providing a set of method names $M_B \subseteq \mathcal{M}$ is formally defined as a deterministic timed I/O automaton over alphabet $\Sigma^B$ which is partitioned into two sets of actions:

- object outputs received by the environment: $\Sigma_O^B = \{m! | m \in \mathcal{M} \wedge m \notin M_B\}$
- object inputs sent by the environment: $\Sigma_I^B = \{m(d)? | m \in M_B \wedge d \in \mathbb{N}\}$

We allow underspecified actions where no deadline is given, e.g., for output actions above. An underspecified deadline is potentially stronger than any specified deadline value $d \in \mathbb{N}$; therefore, to be able to reuse the definition of refinement, we assume that underspecified actions have a deadline zero.

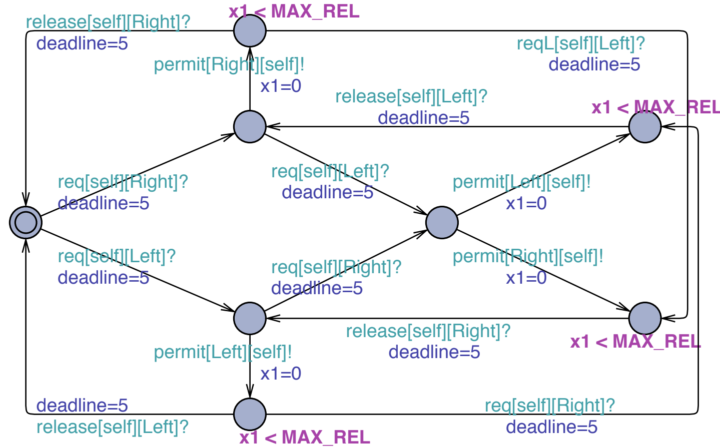



**Fig. 4.** A mutually exclusive resource.

A behavioral interface abstracts from specific method implementations, the queue in the object and the scheduling strategy. It can also be seen as the highest level of abstraction (i.e., an over-approximation) of the environments that can communicate with the object.

Fig. 3 (left) gives the behavioral interface of a resource object which guarantees the deadline $d$ on its inputs *req* and *release*. Furthermore, when a requester is permitted to take the resource, it has to release it before MAX_REL time units. This automaton is parameterized in $i$ which must be instantiated with the identity of the requester object when the requester and the resource objects are composed. If there are two requesters, the behavioral interface of the resource can be obtained by composing two instances of this automaton with different values for $i$.

Fig. 4 gives the behavioral interface of a mutually exclusive resource shared by two objects Right and Left. When there are two requests at a time, only one of them is granted in this model. This model is a refinement of the unrestricted model in Fig. 3 if $5 <= d$. To check refinement, we must apply the algorithm NegGuard to Fig. 4 and both algorithms NegGuard and NegInv to Fig. 3 (the result of the latter is shown in Fig. 3 right).

**Classes** One can define a class $R$ as a set of methods implementing a specific behavioral interface $B$, which must include at least the methods $M_B$. For an input action $m(d)!$ in the behavioral interface, a correct implementation should be able to finish method $m$ before $d$ time units. A class $R$ implementing the behavioral interface $B$ is a set $\{(m_1, A_1), \ldots, (m_n, A_n)\}$ of methods, where

- $M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of method names such that $M_B \subseteq M_R$;
- for all $i$, $1 \leq i \leq n$, $A_i$ is a timed I/O automaton representing method $m_i$ with the output alphabet $\Sigma_i = \{m! | m \in M_R\} \cup \{m(d)! \mid m \in \mathcal{M} \land d \in \mathbb{N}\}$ and no explicit inputs.

Classes have an *initial* method which is implicitly called upon initialization and is used for the system startup. Method automata only send messages while computations are abstracted into time delays. Sending a message $m \in M_R$ is called a self call. Self calls may or may not be assigned an explicit deadline. The self calls with no explicit deadline are called *delegation*. Delegation implies that the internal task (triggered by the self call) is in fact the continuation of the parent task; therefore, the delegated task inherits the (remaining) deadline of the task that triggers it.

**Schedulers** Receiving and buffering messages and executing the corresponding methods is handled by the scheduler automata. A *scheduler automaton* implements a queue for storing messages and their deadlines. The scheduler of a concurrent object is input enabled, i.e., it can receive any message in $M_R$ at any time; this is to model the asynchronous nature of communication between the objects. Whenever a method is finished, the scheduler selects another message from the queue (based on its scheduling strategy) and starts the corresponding method (called context-switch).

Since the object is strongly input-enabled, i.e., it may accept any input at any time, it is not per se a refinement of the behavioral interface; because the object may wait (i.e., have a delay transition) for an input while it is not allowed (i.e., expected) in the behavioral interface. Next, we describe how we may restrict the object behavior so that it is schedulable; in this case, it is a correct refinement of its behavioral interface.

**Schedulable Objects** An object is an instance of a class together with a scheduler automaton. An object is called *schedulable* if it can finish all of its tasks within their deadlines. An unrestricted object is trivially non-schedulable, because it may accept too many inputs in a short time. To restrict the possible ways in which the methods of an object could be called, we consider only the incoming messages specified in its behavioral interface. To check an object for schedulability (e.g., in UPPAAL), the inputs of $B$ are changed to outputs $m!$ so that they match the inputs in the scheduler written as $m?$ and the outputs of $B$ are changed to inputs written as $m?$ so that they match outputs of method automata written as $m!$.

The scheduler automaton moves to an Error location with no outgoing transitions when a task in the queue misses its deadline. Furthermore, as shown in [9], a schedulable object never puts more than $\lceil d_{max}/b_{min} \rceil$ messages in the queue, where $d_{max}$ is the longest deadline for any method called on any transition of the automata (method automata or the input actions of the behavioral interface) and $b_{min}$ is the shortest termination time of any of the method automata. Thus we can put a finite bound on the queue length such that queue overflow implies non-schedulability. We can calculate the best case runtime for timed automata as shown in [5].

We explained in [9] how the restricted behavioral model of an object can be constructed as one automaton. The actions of this automaton are the same as its

behavioral interface. We have also shown in [9] how to model an object in UPPAAL. To capture possible design errors, one can start with checking for deadlock in UPPAAL. A deadlock may be caused by a mismatching invariant and guards in a method implementation, or if the Error location in the scheduler is reached. To ensure schedulability at the same time, one should add a check for queue overflow. This can be written in UPPAAL as "A$\Box$ not deadlock and tail $\leq$MAX". Furthermore, one may check other properties on the restricted object behavior. It is easy to see that when the restricted object model is schedulable, it is also a true refinement of the behavioral interface.

## 6 Real-Time Distributed Systems

Once an object is checked for schedulability with respect to its behavioral interface, it can be used as an off-the-shelf component to compose distributed systems. If the assumptions in the behavioral interface of the object are satisfied, the correct behavior of the object is guaranteed (with respect to the properties already checked for the object, e.g., its schedulability). Checking this is usually referred to as *compatibility* check.

In an optimistic approach [1] two interfaces are considered compatible when there is a way that they can work together. In this case, there exists at least one implementation of those interfaces that are compatible, too. What actually needs to be done next is to check whether the implementations at hand indeed follow the traces that make their interfaces work together.

For concurrent objects, the composition of their behavioral interfaces shows the acceptable sequences of messages that may be communicated between the objects. As compatibility is defined in [10], the system implementation at hand must be a refinement of the composition of the behavioral interfaces. It is shown in [10] that, assuming individually schedulable concurrent objects, their composition is schedulable if they are compatible.

Since our definition of refinement in this paper is not restricted to closed systems, we can generalize compatibility to any open or closed system. When compatible concurrent objects form an open component, the composition of their behavioral interfaces serves as the behavioral interface of their composition. Below, we write $A : B$ to denote an object $A$ with its input behavior restricted to a behavioral interface $B$ (as explained in the previous section).

**Definition 4 (Compatibility).** *We define the concurrent objects $A_i : B_i$ ($1 \leq i \leq n$) to be compatible iff $A_1 \parallel \cdots \parallel A_n \sqsubseteq B_1 \parallel \cdots \parallel B_n$.*

Since the compsition of concurrent objects is usually too big (due to their asynchrony and message queues), model checking compatibility is subject to state-space explosion; therefore, a testing method has been proposed in [10]. Here, we propose to use the compositional refinement check to verify compatibility in this sense.

Given $A_i : B_i$ ($1 \leq i \leq n$) when $A_i : B_i \sqsubseteq B_1 \parallel \cdots \parallel B_n$ for all $1 \leq i \leq n$, it follows from Theorem 1 that the composition of the restricted objects $A' =$

$A_1 : B_1 \parallel \cdots \parallel A_n : B_n$ is a refinement of $B = B_1 \parallel \cdots \parallel B_n$. We still need to show that the composition of the unrestricted objects $A = A_1 \parallel \cdots \parallel A_n$ is also a refinement of $B$; in fact, in this setting the behavior of $A$ and $A'$ is the same.

**Theorem 2.** *The closed system $A_1 \parallel \cdots \parallel A_n$ is trace equivalent to the restricted system $A_1 : B_1 \parallel \cdots \parallel A_n : B_n$ if $\forall_{1 \leq i \leq n} A_i : B_i \sqsubseteq B_1 \parallel \cdots \parallel B_n$.*

Theorems 1 and 2 result in the following corollary:

**Corollary 2.** *The concurrent objects $A_i : B_i$ $(1 \leq i \leq n)$ are compatible iff $A_i : B_i \sqsubseteq B_1 \parallel \cdots \parallel B_n$ for all $1 \leq i \leq n$.*

This implies that given individually schedulable objects, their composition is also schedulable if we can show that each object is a refinement of the composition of the behavioral interfaces of all objects.

## 7 Conclusions and Future Work

We bridge the gap between automata theory and object orientation. In previous work, we developed schedulability analysis techniques for concurrent objects modeled in timed I/O automata. In this work, we further developed the related automata theory such that we can check compatibility in a compositional way.

To be able to argue about schedulability, we extended timed I/O automata with deadlines. Furthermore, we extended the definition of composition and refinement to include deadlines. On the other hand, our definition of refinement is not restricted to automata with the same sets of inputs and outputs; this allows us to compare a component, modeled as an automaton, with a composition of components for refinement.

We applied the refinement theory for timed I/O automata with deadlines to compositional schedulability analysis of systems modeled with concurrent objects. Each concurrent object is model checked to be schedulable when its input behavior is restricted as specified in its behavioral interface; a system is schedulable when all objects receive inputs as they expect according to their behavioral interface. This *compatibility* can be ensured by checking whether the system is a refinement of the composition of the behavioral interfaces. We showed in this paper how to model check this in a compositional way.

A possible line of future research is considering network delays between concurrent objects when composed. Network delays both affect the deadlines of messages and the input assumptions of the object receiving that message. Moreover, complex network structures can also be added to coordinate distributed schedulable services; for example, to balance the load of a fast client between multiple slow servers.

## References

1. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: Proc. Embedded Software (EMSOFT). LNCS, vol. 2491, pp. 108–122 (2002)

2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: CONCUR '98. LNCS, vol. 1466, pp. 163–178 (1998)
4. Closse, E., Poize, M., Pulou, J., Sifakis, J., Venter, P., Weil, D., Yovine, S.: TAXYS: A tool for the development and verification of real-time embedded systems. In: Proc. CAV'01. LNCS, vol. 2102, pp. 391–395. Springer (2001)
5. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. Formal Methods in System Design 1(4), 385–415 (1992)
6. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: Proc. Hybrid Systems: Computation and Control (HSCC'10). pp. 91–100. ACM (2010)
7. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)
8. Garcia, J.J.G., Gutierrez, J.C.P., Harbour, M.G.: Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In: Proc. 12th Euromicro Conference on Real-Time Systems. pp. 15–24. IEEE (2000)
9. Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. J. Logic and Alg. Prog. 78(5), 402 – 416 (2009)
10. Jaghoori, M.M., Longuet, D., de Boer, F.S., Chothia, T.: Schedulability and compatibility of real time asynchronous objects. In: Proc. RTSS'08. pp. 70–79. IEEE CS (2008)
11. Krcal, P., Stigge, M., Yi, W.: Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. In: Proc. Formal Modeling and Analysis of Timed Systems. LNCS, vol. 4763, pp. 274–289 (2007)
12. Kupferman, O., Vardi, M.Y., Wolper, P.: Module checking. Information and Computation 164(2), 322–344 (2001)
13. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)
14. Meyer, B.: Eiffel: The language. Prentice-Hall (1992)
15. Nigro, L., Pupo, F.: Schedulability analysis of real time actor systems using coloured petri nets. In: Proc. Concurrent Object-Oriented Programming and Petri Nets. LNCS, vol. 2001, pp. 493–513. Springer (2001)

## Proofs Omitted From Text

**Theorem 1.** Given the timed I/O automata $A_1$, $A_2$ and $B$, we have:

$$A_1 \sqsubseteq B \ \wedge \ A_2 \sqsubseteq B \implies A_1 \parallel A_2 \sqsubseteq B$$

*Proof.* For simplicity, we give the proof without considering deadlines. Deadlines can be added to the proof in a straightforward way.

We write the states of (the underlying transition system of) $A = A_1 \parallel A_2$ as $(s_1, s_2)$ where $s_i$ is a state in (the underlying transition system of) $A_i$. We write $(s_1, s_2)R(t)$ to relate a state $(s_1, s_2)$ in $A$ to $t$ in $B$ using a relation $R$. We assume $A_1 \sqsubseteq B$ and $A_2 \sqsubseteq B$ with the refinement relations $R_1$ and $R_2$, respectively, as defined in Def. 2. We define $R$ such that $(s_1, s_2)R(t)$ if and only if $(s_1, t) \in R_1$ or

$(s_2, t) \in R_2$. We show below that the relation $R$ satisfies the requirements put forward in Def. 2 and therefore $A \sqsubseteq B$.

Obviously $R$ relates the initial states of $A$ and $B$. Let's assume that $(s_1, s_2)R(t)$. The set of sync actions of $A_1$ and $A_2$ are $\Sigma_\cap = (\Sigma^I_{A_1} \cap \Sigma^O_{A_2}) \cup (\Sigma^O_{A_1} \cap \Sigma^I_{A_2})$. By definition of composition, we have $\Sigma_\cap \subseteq \Sigma^\tau_A$.

- For $a \in \Sigma^I_A$, we know that $a \in \Sigma^I_{A_1}$ or $a \in \Sigma^I_{A_2}$ and $a$ is not a sync action. Without loss of generality, we take $a \in \Sigma^I_{A_1}$. Since $A_1 \sqsubseteq B$, by Def. 2, we know that if $t \xrightarrow{a} t'$ in $B$ there is a transition $s_1 \xrightarrow{a} s'_1$ in $A_1$ and $(s'_1, t') \in R_1$. Since $a$ is not a sync action, there is a transition $(s_1, s_2) \xrightarrow{a} (s'_1, s_2)$ in $A$, too. Since $(s'_1, t') \in R_1$, we have $(s'_1, s_2)R(t')$.
- For $a \in \Sigma^O_A \cup ((\Sigma^\tau_A \cap \Sigma_B) \setminus \Sigma_\cap)$, i.e., excluding sync actions, we assume, without loss of generality, that $a \in \Sigma^O_{A_1} \cup (\Sigma^\tau_{A_1} \cap \Sigma_B)$. In this case, $A$ may have a transition $(s_1, s_2) \xrightarrow{a} (s'_1, s_2)$ only if there is $s_1 \xrightarrow{a} s'_1$ in $A_1$. From $A_1 \sqsubseteq B$, we can say that there is also a transition $t \xrightarrow{a} t'$ in $B$ and $(s'_1, t') \in R_1$. Since $(s'_1, t') \in R_1$, we have $(s'_1, s_2)R(t')$.
- For $a \in \Sigma^\tau_A \cap \Sigma_B \cap \Sigma_\cap$, $A$ may have a transition $(s_1, s_2) \xrightarrow{a} (s'_1, s'_2)$ only if there are $s_1 \xrightarrow{a} s'_1$ in $A_1$ and $s_2 \xrightarrow{a} s'_2$ in $A_2$. Without loss of generality, we assume $a \in \Sigma^O_{A_1} \cap \Sigma^I_{A_2}$. By considering $A_1 \sqsubseteq B$ and $s_1 \xrightarrow{a} s'_1$, we can conclude that there is a transition $t \xrightarrow{a} t'$ in $B$ and $(s'_1, t') \in R_1$. Since $(s'_1, t') \in R_1$, we have $(s'_1, s'_2)R(t')$.
- Finally, if $A$ can delay for $d$ time units, both $A_1$ and $A_2$ can delay and therefore $B$ can delay for $d$ time units. It is easy to see that the target states are related by $R$.

□

**Theorem 2.** The closed system $A_1 \parallel \cdots \parallel A_n$ is trace equivalent to the restricted system $A_1 : B_1 \parallel \cdots \parallel A_n : B_n$ if $\forall_{1 \leq i \leq n} A_i : B_i \sqsubseteq B_1 \parallel \cdots \parallel B_n$.

*Proof (idea).* It is easy to see that every trace in $A'$ also exists in $A$, because every $A_i : B_i$ is a in fact restriction of $A_i$.

To show the other direction, take a trace $\sigma = (t_1, a_1) \dots (t_n, a_n)$ from $A$. We use induction to show that $\sigma$ is also a trace in $A'$. As the base case, since $A$ and $A'$ start in the same initial states, they can generate the same initial outputs. Therefore, $A'$ can output $a_1$ at time $t_1$. Assume that for $j < n$, $\sigma_j = (t_1, a_1) \dots (t_{j-1}, a_{j-1})$ exists in $A'$ and furthermore $A'$ can output $a_j$ at time $t_j$. We must show that $a_j$ is also an acceptable input at time $t_j$.

Suppose $a_j$ is an output action of $A_{j1}$ and an input action of $A_{j2}$. Since $A_{j1} : B_{j1}$ is a refinement of $B$, the action $a_j$ exists in $B$; and since $A_{j2} : B_{j2}$ is also a refinement of $B$, the action $a_j$ is acceptable in $A_{j2} : B_{j2}$ at time $t_j$. Next, $A'$ can produce the output action $a_{j+1}$ at time $t_{j+1}$ because it has the same methods as $A$. □