Decomposing Verification of Timed I/O Automata

Dilsun Kırlı Kaynar and Nancy Lynch

MIT Computer Science and Artificial Intelligence Laboratory
{dilsun,lynch}@csail.mit.edu

Abstract. This paper presents assume-guarantee style substitutivity results for the recently published timed I/O automaton modeling framework. These results are useful for decomposing verification of systems where the implementation and the specification are represented as timed I/O automata. We first present a theorem that is applicable in verification tasks in which system specifications express safety properties. This theorem has an interesting corollary that involves the use of auxiliary automata in simplifying the proof obligations. We then derive a new result that shows how the same technique can be applied to the case where system specifications express liveness properties.

1 Introduction

The timed I/O automata (TIOA) modeling framework [KLSV03a,KLSV03b] provides a *composition operation*, by which TIOAs modeling individual timed system components can be combined to produce a model for a larger timed system. The model for the composed system can describe interactions among the components, which involves joint participation in discrete transitions. Composition requires certain "compatibility" conditions, namely, that each output action be controlled by at most one automaton, and that internal actions of one automaton cannot be shared by any other automaton.

The composition operation for TIOAs satisfies *projection* and *pasting* results, which are fundamental for compositional design and verification of systems: a trace of a composition of TIOAs "projects" to give traces of the individual TIOAs, and traces of components are "pastable" to give behaviors of the composition. This allows one to derive conclusions about the behavior of a large system by combining the results obtained from the analysis of each individual component.

The composition operation for TIOAs also satisfies a basic substitutivity result that states that the composition operation respects the implementation relation for TIOAs. An automaton $\mathcal{A}_1$ is said to *implement* an automaton $\mathcal{A}_2$ if the set of traces of $\mathcal{A}_1$ is included in the the set of traces of $\mathcal{A}_2$. The implementation relation is a congruence with respect to parallel composition. That is, given an automaton $\mathcal{B}$, if $\mathcal{A}_1$ implements $\mathcal{A}_2$ then the composition $\mathcal{A}_1 \| \mathcal{B}$ implements the composition $\mathcal{A}_2 \| \mathcal{B}$. A corollary of this basic substitutivity result is that, if $\mathcal{A}_1$ implements a specification $\mathcal{A}_2$ and $\mathcal{B}_1$ implements a specification $\mathcal{B}_2$ then $\mathcal{A}_1 \| \mathcal{B}_1$ implements $\mathcal{A}_2 \| \mathcal{B}_2$.

The basic substitutivity property described above is desirable for any formalism for interacting processes. For design purposes, it enables one to refine individual components without violating the correctness of the system as a whole. For verification purposes, it enables one to prove that a composite system satisfies its specification by proving that each component satisfies its specification, thereby breaking down the verification task into more manageable pieces. However, it might not always be possible or easy to show that each component $\mathcal{A}_1$ (resp. $\mathcal{B}_1$) satisfies its specification

$\mathcal{A}_2$ (resp. $\mathcal{B}_2$) without using any assumptions about the environment of the component. *Assume-guarantee* style results such as those presented in [Jon83,Pnu84,Sta85,AL93,AL95,HQR00,TAKB96] are special kinds of substitutivity results that state what *guarantees* are expected from each component in an environment constrained by certain *assumptions*. Since the environment of each component consists of the other components in the system, assume-guarantee style results need to break the circular dependencies between the assumptions and guarantees for components.

This paper presents assume-guarantee style theorems for use in verification and analysis of timed systems within the TIOA framework. The first theorem allows one to conclude that $\mathcal{A}_1 \| \mathcal{B}_1$ implements $\mathcal{A}_2 \| \mathcal{B}_2$ provided that $\mathcal{A}_1$ implements $\mathcal{A}_2$ in the context of $\mathcal{B}_2$ and $\mathcal{B}_1$ implements $\mathcal{B}_2$ in the context of $\mathcal{A}_2$, where $\mathcal{A}_2$ and $\mathcal{B}_2$ express safety constraints and admit arbitrary time-passage. This theorem has an interesting corollary that involves the use of auxiliary automata $\mathcal{A}_3$ and $\mathcal{B}_3$ in decomposing the proof that $\mathcal{A}_1 \| \mathcal{B}_1$ implements $\mathcal{A}_2 \| \mathcal{B}_2$. The main idea behind this corollary is to capture, by means of $\mathcal{A}_3$ and $\mathcal{B}_3$, what is essential about the behavior of the contexts $\mathcal{A}_2$ and $\mathcal{B}_2$ in proving the implementation relationship. The second theorem extends this corollary to the case where liveness conditions are added to automaton specifications. This theorem requires one to find the appropriate auxiliary liveness properties for $\mathcal{A}_3$ and $\mathcal{B}_3$, in addition to what is already needed for proving the safety part of the specification. The liveness properties devised for $\mathcal{A}_3$ and $\mathcal{B}_3$ are supposed to capture what liveness guarantees of the contexts $\mathcal{A}_2$ and $\mathcal{B}_2$ are essential in showing the implementation relationship.

Related work. The results of this paper constitute the first assume-guarantee style results for timed I/O automata. Assume-guarantee reasoning has been previously investigated in various formal frameworks, most commonly, in frameworks based on temporal logics [Pnu84,Sta85,AL93,AL95] and reactive modules [HQR00,HQR02]. Although some of these frameworks such as TLA and reactive modules can be extended to support modeling of timed system behavior [AL94,AH97], it is hard to understand whether all of the results and reasoning techniques obtained for their untimed versions generalize to the timed setting. The work presented in [TAKB96] considers a framework based on timed processes that underlies the language of the tool COSPAN [AK96]. The focus of that paper is timed simulation relations, how they relate to verification based on language inclusion and algorithmic aspects of checking for timed simulations. The topic of assume-guarantee reasoning is visited only for a single theorem, which is akin to the first theorem of this paper. Our other theorems that involve the use of auxiliary automata and liveness properties appear to incorporate novel and simple ideas that have not been investigated before in decomposing verification of timed systems.

Organization of the paper. Section 2 introduces the basic concepts of the TIOA framework, and gives the basic definitions and results relevant to what its presented in the rest of the paper. This section also states the notational conventions used in writing the TIOA specifications that appear in the examples. Section 3 gives a theorem and its corollary that can be used in decomposing verification of systems where the TIOAs express safety properties. Section 4 shows how the ideas of Section 3 can be applied to decomposition of verification where TIOAs express liveness properties as well as safety properties. Section 5 summarizes the contributions of the paper and discusses possible directions for future work.

2 Timed I/O Automata

In this section, we present briefly the basic definitions and results from the timed I/O modeling framework that are necessary to understand the material in this paper. The reader is referred to [KLSV03a] for the details.

2.1 Describing Timed System Behavior

We use the set R of real numbers as the domain (in [KLSV03a] other time domains are also considered). A time interval is a nonempty, convex subset of R. An interval is *left-closed (right-closed)* if it has a minimum (resp., maximum) element, and *left-open (right-open)* otherwise. It is *closed* if it is both left-closed and right-closed.

States of automata will consist of valuations of *variables*. Each variable has both a *static type*, which defines the set of values it may assume, and a *dynamic type*, which gives the set of trajectories it may follow. We assume that dynamic types are closed under some simple operations: shifting the time domain, taking subintervals and pasting together intervals. We call a variable *discrete* if its dynamic type equals the pasting-closure of a set of constant-valued functions (i.e., the step-functions), and *analog* if its dynamic type equals the pasting-closure of a set of continuous functions (i.e., the piecewise-continuous functions).

A *valuation* for a set V of variables is a function that associates with each variable $v \in V$ a value in its static type. We write $val(V)$ for the set of all valuations for V. A *trajectory* for a set V of variables describes the evolution of the variables in V over time; formally, it is a function from a time interval that starts with 0 to valuations of V, that is, a trajectory defines a value for each variable at each time in the interval. We write $trajs(V)$ for the set of all trajectories for V. A *point trajectory* is one with the trivial domain $\{0\}$. The *limit time* of a trajectory τ, $\tau.ltime$, is the supremum of the times in its domain. We say that a trajectory is *closed* if its domain is a closed interval. $\tau.fval$ is defined to be the first valuation of τ, and if τ is closed, $\tau.lval$ is the last valuation. Suppose τ and τ' are trajectories for V, with τ closed. The *concatenation* of τ and τ', denoted by $\tau \frown \tau'$, is the trajectory obtained by taking the union of the first trajectory and the function obtained by shifting the domain of the second trajectory until the start time agrees with the limit time of the first trajectory; the last valuation of the first trajectory, which may not be the same as the first valuation of the second trajectory, is the one that appears in the concatenation.

The notion of a *hybrid sequence* is used to model a combination of changes that occur instantaneously and changes that occur over intervals of time. Our definition is parameterized by a set A of discrete actions and a set V of variables. Thus, an (A, V)*-sequence* is a finite or infinite alternating sequence, $\tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \ldots$, of trajectories over V and actions in A. A *hybrid sequence* is any (A, V)-sequence. The *limit time* of a hybrid sequence α, denoted by $\alpha.ltime$, is defined by adding the limit times of all its trajectories. Hybrid sequence α is defined to be *admissible* if $\alpha.ltime = \infty$, and *closed* if it is a finite sequence and the domain of its final trajectory is a closed interval. Like trajectories, hybrid sequences can be concatenated, and one can be a prefix of another. If α is a closed (A, V)-sequence, where $V = \emptyset$ and $\beta \in trajs(\emptyset)$, we call $\alpha \frown \beta$ a *time-extension* of α. A hybrid sequence can also be restricted to smaller sets of actions and variables: the (A', V')*-restriction* of an (A, V)-sequence α is obtained by first projecting all trajectories of α on the variables in V', then removing the actions not in A', and finally concatenating all adjacent trajectories.

A set S of hybrid sequences is said to be *closed under limits* if for each chain (with respect to prefix ordering) of closed hybrid sequences in S, the limit of the chain is in S.

2.2 Timed I/O Automata Definition

Formally, a *timed I/O automaton (TIOA)* consists of:

- A set X of *internal variables.*
- A set $Q \subseteq val(X)$ of *states.*
- A nonempty set $\Theta \subseteq Q$ of *start states*.
- A set H of *internal actions*, a set I of *input* actions and a set O of *output actions*. We write $E \triangleq I \cup O$ for the set of external actions and $A \triangleq E \cup H$ for the set of all actions. Actions in $L \triangleq H \cup O$ are called *locally controlled*.
- A set $\mathcal{D} \subseteq Q \times A \times Q$ of *discrete transitions.*
  We use $\mathbf{x} \stackrel{a}{\to} \mathbf{x}'$ as shorthand for $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$. We say that a is *enabled* in $\mathbf{x}$ if $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$ for some $\mathbf{x}'$.
- A set $\mathcal{T}$ of trajectories for X such that $\tau(t) \in Q$ for every $\tau \in \mathcal{T}$ and every t in the domain of τ. Given a trajectory $\tau \in \mathcal{T}$ we denote $\tau.fval$ by $\tau.fstate$ and, if τ is closed, we denote $\tau.lval$ by $\tau.lstate$.

We require that the set of trajectories be closed under the operations of prefix, suffix, and concatenation and that there is a point trajectory for every state of the automaton. Moreover, the following axioms are satisfied:

E1 (Input action enabling)
For every $\mathbf{x} \in Q$ and every $a \in I$, there exists $\mathbf{x}' \in Q$ such that $\mathbf{x} \stackrel{a}{\to} \mathbf{x}'$.

E2 (Time-passage enabling)
For every $\mathbf{x} \in Q$, there exists $\tau \in \mathcal{T}$ such that $\tau.fstate = \mathbf{x}$ and either $\tau.ltime = \infty$, or τ is closed and some $l \in L$ is enabled in $\tau.lstate$.

Executions and traces. An *execution fragment* of a TIOA $\mathcal{A}$ is an (A, V)-sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots$, where A and V are all the actions and variables of $\mathcal{A}$, respectively, where each τ_i is a trajectory of $\mathcal{A}$, and for every i, $\tau_i.lval \stackrel{a_{i+1}}{\to} \tau_{i+1}.fval$. An execution fragment records what happens during a particular run of a system, including all the discrete state changes and all the changes that occur while time advances. An *execution* is an execution fragment whose first state is a start state of $\mathcal{A}$. We write $frags_{\mathcal{A}}$ for the set of all execution fragments of $\mathcal{A}$ and $execs_{\mathcal{A}}$ for the set of all executions of $\mathcal{A}$.

The external behavior of a TIOA is captured by the set of "traces" of its execution fragments, which record external actions and the intervening passage of time. Formally, the *trace* of an execution fragment α is the $(E, \emptyset)$-restriction of α. Thus, a trace is a hybrid sequence consisting of external actions of $\mathcal{A}$ and trajectories over the empty set of variables. The only interesting information contained in these trajectories is the amount of time that elapses. A *trace fragment* of $\mathcal{A}$ is the trace of an execution fragment of $\mathcal{A}$, and a *trace* of $\mathcal{A}$ is the trace of an execution of $\mathcal{A}$. We write $tracefrags_{\mathcal{A}}(\mathbf{x})$ for the set of trace fragments of $\mathcal{A}$ from $\mathbf{x}$ and $traces_{\mathcal{A}}$ for the set of traces of $\mathcal{A}$.

Implementation relationships. Timed I/O automata $\mathcal{A}$ and $\mathcal{B}$ are *comparable* if they have the same external actions. If $\mathcal{A}$ and $\mathcal{B}$ are comparable then $\mathcal{A}$ *implements* $\mathcal{B}$, denoted by $\mathcal{A} \leq \mathcal{B}$, if $traces_{\mathcal{A}} \subseteq traces_{\mathcal{B}}$.

Composition. We say that TIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if, for $i \neq j$, $X_i \cap X_j = H_i \cap A_j = O_i \cap O_j = \emptyset$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible TIOAs then their *composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is defined to be the tuple $\mathcal{A} = (X, Q, \Theta, H, I, O, \mathcal{D}, \mathcal{T})$ where

- $X = X_1 \cup X_2$.
- $Q = \{\mathbf{x} \in val(X) \mid \mathbf{x} \lceil X_1 \in Q_1 \wedge \mathbf{x} \lceil X_2 \in Q_2\}$. [1]
- $\Theta = \{\mathbf{x} \in Q \mid \mathbf{x} \lceil X_1 \in \Theta_1 \wedge \mathbf{x} \lceil X_2 \in \Theta_2\}$.
- $H = H_1 \cup H_2$.
- $I = (I_1 \cup I_2) - (O_1 \cup O_2)$
- $O = O_1 \cup O_2$.
- For each $\mathbf{x}, \mathbf{x}' \in Q$ and each $a \in A$, $\mathbf{x} \xrightarrow{a}_{\mathcal{A}} \mathbf{x}'$ iff for $i \in \{1,2\}$, either (1) $a \in A_i$ and $\mathbf{x} \lceil X_i \xrightarrow{a}_i \mathbf{x}' \lceil X_i$, or (2) $a \notin A_i$ and $\mathbf{x} \lceil X_i = \mathbf{x}' \lceil X_i$.
- $\mathcal{T} = \{\tau \in trajs(X) \mid \tau \downarrow X_1 \in \mathcal{T}_1 \wedge \tau \downarrow X_2 \in \mathcal{T}_2\}$. [2]

The following theorem is a fundamental theorem that relates the set of traces of a composed automaton to the sets of traces of its components. Set inclusion in one direction expresses the idea that a trace of a composition "projects" to yield traces of the components. Set inclusion in the other direction expresses the idea that traces of components can be "pasted" to yield a trace of the composition.

Theorem 1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be comparable TIOAs, and let $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$. Then $traces_{\mathcal{A}}$ is exactly the set of $(E, \emptyset)$-sequences whose restrictions to $\mathcal{A}_1$ and $\mathcal{A}_2$ are traces of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. That is, $traces_{\mathcal{A}} = \{\beta \mid \beta$ is an $(E, \emptyset)$-sequence and $\beta \lceil (E_i, \emptyset) \in traces_{\mathcal{A}_i}, i = \{1,2\}\}$.*

2.3 Properties

A *property* P for a timed I/O automaton $\mathcal{A}$ is defined to be any subset of the execution fragments of $\mathcal{A}$. We write $execs_{(\mathcal{A},P)}$ for the set of executions of $\mathcal{A}$ in P and $traces_{(\mathcal{A},P)}$ for the set of traces of executions of $\mathcal{A}$ in P.

A property P for a TIOA $\mathcal{A}$ is said to be a *safety* property if it is closed under prefix and limits of execution fragments. A property P for $\mathcal{A}$ is defined to be a *liveness* property provided that for any closed execution fragment α of $\mathcal{A}$, there exists an execution fragment β such that $\alpha \frown \beta \in P$.

In the TIOA modeling framework safety properties are typically specified as all of the execution fragments of a TIOA. That is, no extra machinery other than the automaton specification itself is necessary to specify a safety property. The set of execution fragments of a TIOA is closed under prefix and limits. When the external behavior sets (trace sets), rather than execution fragments are taken as a basis for specification, an automaton is said to specify a safety property only if its trace set is closed under limits (trace sets are closed under prefixes by definition).

Liveness properties are typically specified by coupling a TIOA $\mathcal{A}$ with a property P where P is a liveness property. A pair $(\mathcal{A}, P)$ can be viewed as a two-part specification: a safety condition expressed by the automaton $\mathcal{A}$ and a liveness condition expressed by P. It is in general desirable that P does not itself impose safety constraints, beyond those already imposed by the execution

[1] If f is a function and S is a set, then we write $f \lceil S$ for the restriction of f to S, that is, the function g with $dom(g) = dom(f) \cap S$ such that $g(c) = f(c)$ for each $c \in dom(g)$.

[2] If f is a function whose range is a set of functions and S is a set, then we write $f \downarrow S$ for the function g with $dom(g) = dom(f)$ such that $g(c) = f(c) \lceil S$ for each $c \in dom(g)$.

fragments of $\mathcal{A}$. To achieve this, P should be defined so that every closed execution in P can be extended to some execution that is in both $execs_{\mathcal{A}}$ and P. The notion of machine-closure is used to formalize this condition. A detailed discussion can be found in [KLSV03a].

Implementation relationships. In analogy with basic TIOAs, we define another preorder for automata with properties: $(\mathcal{A}_1, P_1) \leq (\mathcal{A}_2, P_2)$ provided that $traces_{(\mathcal{A}_1,P_1)} \subseteq traces_{(\mathcal{A}_2,P_2)}$.

Composition. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are two compatible timed I/O automata and P_1 and P_2 are properties for $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, then we define $P_1 \| P_2$ to be $\{\alpha \in frags_{\mathcal{A}_1 \| \mathcal{A}_2} \mid \alpha \lceil (A_i, X_i) \in P_i, i \in \{1, 2\}\}$. Using this, we define composition of automata with properties $(\mathcal{A}_1, P_1) \| (\mathcal{A}_2, P_2)$ as $(\mathcal{A}_1 \| \mathcal{A}_2, P_1 \| P_2)$.

Theorem 2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two compatible TIOAs and P_1 and P_2 be properties for $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Then $traces_{(\mathcal{A}_1 \| \mathcal{A}_2, P_1 \| P_2)}$ is exactly the set of $(E, \emptyset)$-sequences whose restrictions to $\mathcal{A}_1$ and $\mathcal{A}_2$ are $traces_{(\mathcal{A}_1,P_1)}$ and $traces_{(\mathcal{A}_2,P_2)}$, respectively. That is,*
$traces_{(\mathcal{A}_1 \| \mathcal{A}_2, P_1 \| P_2)} = \{\beta \mid \beta$ *is an* $(E, \emptyset)$*-sequence and* $\beta \lceil (E_i, \emptyset) \in traces_{(\mathcal{A}_i,P_i)}, i \in \{1, 2\}\}$.

2.4 Conventions for Writing TIOA Specifications

We typically specify sets of trajectories using differential and algebraic equations and inclusions. Suppose the time domain T is R, τ is a (fixed) trajectory over some set of variables V, and $v \in V$. With some abuse of notation, we use the variable name v to denote the function that gives the value of v at all times during trajectory τ. Similarly, we view any expression e containing variables from V as a function with $dom(\tau)$ (the domain of τ). Suppose that v is a variable and e is a real-valued expression containing variables from V. We say that τ satisfies the algebraic equation $v = e$ which means that, for every $t \in dom(\tau)$, $v(t) = e(t)$, that is, the constraint on the variables expressed by the equation $v = e$ holds for each state on trajectory τ. Suppose also that e, when viewed as a function, is integrable. Then we say that τ satisfies $d(v) = e$ if, for every $t \in dom(\tau)$, $v(t) = v(0) + \int_0^t e(t')dt'$. This way of specifying trajectories generalizes to differential inclusions as explained in [KLSV03a].

In the rest of the paper, we use examples to illustrate our theorems. These examples are based on TIOA specifications written using certain notational conventions (see Appendix B). The transitions are specified in precondition-effect style. A **precondition** clause specifies the enabling condition for an action. The **effect** clause contains a list of statements that specify the effect of performing that action on the state. All the statements in an effect clause are assumed to be executed sequentially in a single indivisible step. The absence of a specified precondition for an action means that the action is always enabled and the absence of a specified effect means that performing the action does not change the state.

The trajectories are specified by using a variation of the language presented in [MWLF03]. A **satisfies** clause contains a list of predicates that must be satisfied by all the trajectories. This clause is followed by a **stops when** clause. If the predicate in this clause becomes true at a point t in time, then t must be the limit time of the trajectory. When there is no stopping condition for trajectories we omit the **stops when** clause. In our examples, we write $\mathbf{d}(v) = e$ for $d(v) = e$. If the value of a variable is constant throughout a trajectory then we write $\mathbf{constant}(v)$.

3 Decomposition Theorem for TIOAs

In this section we present two assume/guarantee style results, Theorem 3 and Corollary 1, which can used for proving that a system specified as a composite automaton $\mathcal{A}_1\|\mathcal{B}_1$ implements a specification represented by a composite automaton $\mathcal{A}_2\|\mathcal{B}_2$.

The main idea behind Theorem 3 is to assume that $\mathcal{A}_1$ implements $\mathcal{A}_2$ in a context represented by $\mathcal{B}_2$, and symmetrically that $\mathcal{B}_1$ implements $\mathcal{B}_2$ in a context represented by $\mathcal{A}_2$ where $\mathcal{A}_2$ and $\mathcal{B}_2$ are automata whose trace sets are closed under limits. The requirement about limit-closure implies that $\mathcal{A}_2$ and $\mathcal{B}_2$ specify trace safety properties. Moreover, we assume that $\mathcal{A}_2$ and $\mathcal{B}_2$ allow arbitrary time-passage. This is the most general assumption one could make to ensure that $\mathcal{A}_2\|\mathcal{B}_2$ does not impose stronger constraints on time-passage than $\mathcal{A}_1\|\mathcal{B}_1$.

Theorem 3. *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{B}_1$, $\mathcal{B}_2$ are TIOAs such that $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, $\mathcal{B}_1$ and $\mathcal{B}_2$ are comparable, and $\mathcal{A}_i$ is compatible with $\mathcal{B}_i$ for $i \in \{1,2\}$. Suppose further that:*

1. *The sets $traces_{\mathcal{A}_2}$ and $traces_{\mathcal{B}_2}$ are closed under limits.*
2. *The sets $traces_{\mathcal{A}_2}$ and $traces_{\mathcal{B}_2}$ are closed under time-extension.*
3. *$\mathcal{A}_1\|\mathcal{B}_2 \leq \mathcal{A}_2\|\mathcal{B}_2$ and $\mathcal{A}_2\|\mathcal{B}_1 \leq \mathcal{A}_2\|\mathcal{B}_2$.*

Then $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_2\|\mathcal{B}_2$.

Theorem 3 has a corollary, Corollary 1 below, which can be used in the decomposition of proofs even when $\mathcal{A}_2$ and $\mathcal{B}_2$ neither admit arbitrary time-passage nor have limit-closed trace sets. The main idea behind this corollary is to assume that $\mathcal{A}_1$ implements $\mathcal{A}_2$ in a context $\mathcal{B}_3$ that is a variant of $\mathcal{B}_2$, and symmetrically that $\mathcal{B}_1$ implements $\mathcal{B}_2$ in a context that is a variant of $\mathcal{A}_2$. That is, the correctness of implementation relationship between $\mathcal{A}_1$ and $\mathcal{A}_2$ does not depend on all the environment constraints, just on those expressed by $\mathcal{B}_3$ (symmetrically for $\mathcal{B}_1$,$\mathcal{B}_2$, and $\mathcal{A}_3$). In order to use this corollary to prove $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_2\|\mathcal{B}_2$ one needs to be able to find appropriate variants of $\mathcal{A}_2$ and $\mathcal{B}_2$ that meet the required closure properties. This corollary prompts one to pin down what is essential about the behavior of the environment in proving the intended implementation relationship, and also allows one to avoid the unnecessary details of the environment in proofs.

Corollary 1. *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$ are TIOAs such that $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ are comparable, $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ are comparable, and $\mathcal{A}_i$ is compatible with $\mathcal{B}_i$ for $i \in \{1,2,3\}$. Suppose further that:*

1. *The sets $traces_{\mathcal{A}_3}$ and $traces_{\mathcal{B}_3}$ are closed under limits.*
2. *The sets $traces_{\mathcal{A}_3}$ and $traces_{\mathcal{B}_3}$ are closed under time-extension.*
3. *$\mathcal{A}_2\|\mathcal{B}_3 \leq \mathcal{A}_3\|\mathcal{B}_3$ and $\mathcal{A}_3\|\mathcal{B}_2 \leq \mathcal{A}_3\|\mathcal{B}_3$.*
4. *$\mathcal{A}_1\|\mathcal{B}_3 \leq \mathcal{A}_2\|\mathcal{B}_3$ and $\mathcal{A}_3\|\mathcal{B}_1 \leq \mathcal{A}_3\|\mathcal{B}_2$.*

Then $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_2\|\mathcal{B}_2$.

The proofs for Theorem 3 and Corollary 1 can be found in [KLSV03a].

Example 1. This example illustrates that, in cases where specifications $\mathcal{A}_2$ and $\mathcal{B}_2$ satisfy certain closure properties, it is possible to decompose the proof of $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_2\|\mathcal{B}_2$ by using Theorem 3, even if it is not the case that $\mathcal{A}_1 \leq \mathcal{A}_2$ or $\mathcal{B}_1 \leq \mathcal{B}_2$.

The automata *AlternateA* and *AlternateB* in Figure 1 are timing-independent automata in which no consecutive outputs occur without inputs happening in between. *AlternateA* and *AlternateB* perform a handshake, outputting an alternating sequence of a and b actions when they are composed. The automata *CatchUpA* and *CatchUpB* in Figure 2 are timing-dependent automata that do not necessarily alternate inputs and outputs as *AlternateA* and *AlternateB*. *CatchUpA* can perform an arbitrary number of b actions, and can perform an a provided that $counta \leq countb$. It allows $counta$ to increase to one more than $countb$. *CatchUpB* can perform an arbitrary number of a actions, and can perform a b provided that $counta \geq countb + 1$. It allows $countb$ to reach $counta$. Timing constraints require each output to occur exactly one time unit after the last action. *CatchUpA* and *CatchUpB* perform an alternating sequence of a actions and b actions when they are composed.

Suppose that we want to prove that $\mathit{CatchUpA} \parallel \mathit{CatchUpB} \leq \mathit{AlternateA} \parallel \mathit{AlternateB}$. We cannot apply the basic substitutivity theorem since the assertions $\mathit{CatchUpA} \leq \mathit{AlternateA}$ and $\mathit{CatchUpB} \leq \mathit{AlternateB}$ are not true. Consider the trace $\tau_0\, b\, \tau_1\, a\, \tau_2\, a\, \tau_3$ of *CatchUpA* where τ_0, τ_1, τ_2 and τ_3 are trajectories with limit time 1. After having performed one b and one a, *CatchUpA* can perform another a. But, this is impossible for *AlternateA* which needs an input to enable the second a. *AlternateA* and *CatchUpA* behave similarly only when put in a context that imposes alternation.

It is easy to check that *AlternateA* and *AlternateB* satisfy the closure properties required by Assumptions 1 and 2 of Theorem 3 and, hence can be substituted for $\mathcal{A}_2$ and $\mathcal{B}_2$ respectively. Similarly, we can easily check that Assumption 3 is satisfied if we substitute *CatchUpA* for $\mathcal{A}_1$ and *CatchUpB* for $\mathcal{B}_1$.

Example 2. This example illustrates that it may be possible to decompose verification, using Corollary 1, in cases where Theorem 3 is not applicable. If the aim is to show $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$ where $\mathcal{A}_2$ and $\mathcal{B}_2$ do not satisfy the assumptions of Theorem 3, then we find appropriate context automata A_3 and B_3 that abstract from those details of $\mathcal{A}_2$ and $\mathcal{B}_2$ that are not essential in proving $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$.

Consider the automata *UseOldInputA* and *UseOldInputB* in Figure 3. *UseOldInputA* keeps track of whether or not it is *UseOldInputA*'s turn, and when it is *UseOldInputA*'s turn, it keeps track of the next time it is supposed to perform an output. The number of outputs that *UseOldInputA* can perform is bounded by a natural number. In the case of repeated b inputs, it is the oldest input that determines when the next output will occur. The automaton *UseOldInputB* is the same as *UseOldInputA* (inputs and outputs reversed) except that the turn variable of *UseOldInputB* is set to false initially. Note that *UseOldInputA* and *UseOldInputA* are not timing-independent and their trace sets are not limit-closed. For each automaton, there are infinitely many start states, one for each natural number. We can build an infinite chain of traces, where each element in the chain corresponds to an execution starting from a distinct start state. The limit of such a chain, which contains infinitely many outputs, cannot be a trace of *UseOldInputA* or *UseOldInputA* since the number of outputs they can perform is bounded by a natural number. The automaton *UseNewInputA* in Figure 4 behaves similarly to *UseOldInputA* except for the handling of inputs. In the case of repeated b inputs, it is the most recent input that determines when the next output will occur. The automaton *UseNewInputB* in Figure 4 is the same as *UseNewInputA* (inputs and outputs reversed) except that the turn variable of *UseNewInputB* is set to false initially.

Suppose that we want to prove that $\mathit{UseNewInputA} \parallel \mathit{UseNewInputB} \leq \mathit{UseOldInputA} \parallel \mathit{UseOldInputB}$. Theorem 3 is not applicable here because the high-level automata *UseOldInputA* and *UseOldInputB* do not satisfy the required closure properties. However, we can use Corollary 1 to decompose verification. It requires us to find auxiliary automata that are less restrictive than *UseOldInputA* and

UseOldInputB but that are restrictive enough to express the constaints that should be satisfied by the environment, for *UseNewInputA* to implement *UseOldInputA* and for *UseNewInputB* to implement *UseOldInputB*.

The automata *AlternateA* and *AlternateB* in Figure 1 can be used as auxiliary automata in this example. They satisfy the closure properties required by Corollary 1 and impose alternation, which is the only additional condition to ensure the needed trace inclusion.

We can define a forward simulation relation (see [KLSV03a]) from *UseNewInputA* $\|$ *UseNewInputB* to *UseOldInputA* $\|$ *UseOldInputB*, which is based on the equality of the turn variables of the implementation and the specification automata. The fact that this simulation relation only uses the equality of turn variables reinforces the idea that the auxiliary contexts, which only keep track of their turn, capture exactly what is needed for the proof of *UseNewInputA* $\|$ *UseNewInputB* $\leq$ *UseOldInputA* $\|$ *UseOldInputB*. We can observe that a direct proof of this assertion would require one to deal with state variables such as *maxout* and *next* of both *UseOldInputA* and *UseOldInputB*, which do not play any essential role in the proof. On the other hand, by decomposing the proof along the lines of Corollary 1 some of the unnecessary details can be avoided. Even though, this is a toy example with an easy proof it should not be hard to observe how this simplification would scale to large proofs.

4 Decomposition Theorem with TIOAs with Properties

Theorem 3 and its corollary presented in Section 3 assume specification automata whose trace sets are closed under limits, and hence express safety constraints. In this section we present a theorem that can be used in the decomposition of verification where the specification automata may also express liveness properties.

The decomposition of a proof of the assertion $(\mathcal{A}_1, P_1)\|(\mathcal{B}_1, Q_1) \leq (\mathcal{A}_2, P_2)\|(\mathcal{B}_2, Q_2)$ can be viewed as consisting of two parts. The first part involves the decomposition of the proof that $(\mathcal{A}_1, P_1)$ and $(\mathcal{B}_1, Q_1)$ satisfy their safety properties and the second part involves the decomposition of the proof that $(\mathcal{A}_1, P_1)$ and $(\mathcal{B}_1, Q_1)$ satisfy their liveness properties. Theorem 4 uses Corollary 1 for the safety part of proofs; the first four hypotheses of Theorem 4 imply those of Corollary 1. The remaining two hypotheses involve the liveness part of proofs. It requires one to find auxiliary automata with properties, $(\mathcal{A}_3, P_3)$ and $(\mathcal{B}_3, Q_3)$, such that $(\mathcal{A}_1, P_1)$ implements $(\mathcal{A}_3, P_3)$ in the context of $\mathcal{B}_3$ without relying on the liveness property of $\mathcal{B}_3$, and $(\mathcal{B}_1, Q_1)$ implements $(\mathcal{B}_3, Q_3)$ in the context of $\mathcal{A}_3$ without relying on the liveness property of $\mathcal{A}_3$. Moreover, $(\mathcal{A}_1, P_1)$ must implement $(\mathcal{A}_2, P_2)$ in the context of $(\mathcal{B}_3, Q_3)$ and $(\mathcal{B}_1, Q_1)$ must implement $(\mathcal{B}_2, Q_2)$ in the context of $(\mathcal{A}_3, P_3)$. That is, the implementation relation between $(\mathcal{A}_1, P_1)$ and $(\mathcal{A}_2, P_2)$ depend on the liveness property Q_3 of the auxiliary context, and the implementation relation between $(\mathcal{B}_1, Q_1)$ and $(\mathcal{B}_2, Q_2)$ depend on the liveness property P_3 of the auxiliary context.

Theorem 4. *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$ are TIOAs such that $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ are comparable, $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ are comparable, and $\mathcal{A}_i$ is compatible with $\mathcal{B}_i$ for $i \in \{1, 2, 3\}$. Suppose that P_i is a property for A_i and Q_i is a property for $\mathcal{B}_i$ for $i \in \{1, 2, 3\}$. Suppose further that:*

1. *The sets $traces_{\mathcal{A}_3}$ and $traces_{\mathcal{B}_3}$ are closed under limits.*
2. *The sets $traces_{\mathcal{A}_3}$ and $traces_{\mathcal{B}_3}$ are closed under time-extension.*
3. *$\mathcal{A}_2 \leq \mathcal{A}_3$ and $\mathcal{B}_2 \leq \mathcal{B}_3$.*
4. *$\mathcal{A}_1\|\mathcal{B}_3 \leq \mathcal{A}_2\|\mathcal{B}_3$ and $\mathcal{A}_3\|\mathcal{B}_1 \leq \mathcal{A}_3\|\mathcal{B}_2$.*

5. $(\mathcal{A}_1, P_1)\|(\mathcal{B}_3, \mathit{frags}_{\mathcal{B}_3}) \leq (\mathcal{A}_3, P_3)\|(\mathcal{B}_3, \mathit{frags}_{\mathcal{B}_3})$ *and* $(\mathcal{A}_3, \mathit{frags}_{\mathcal{A}_3})\|(\mathcal{B}_1, Q_1) \leq (\mathcal{A}_3, \mathit{frags}_{\mathcal{A}_3})\|(\mathcal{B}_3, Q_3)$.
6. $(\mathcal{A}_1, P_1)\|(\mathcal{B}_3, Q_3) \leq (\mathcal{A}_2, P_2)\|(\mathcal{B}_3, Q_3)$ *and* $(\mathcal{A}_3, P_3)\|(\mathcal{B}_1, Q_1) \leq (\mathcal{A}_3, P_3)\|(\mathcal{B}_2, Q_2)$.

Then $(\mathcal{A}_1, P_1)\|(\mathcal{B}_1, Q_1) \leq (\mathcal{A}_2, P_2)\|(\mathcal{B}_2, Q_2)$.

The proof sketch for Theorem 4 is given in Appendix A.

Example 3. This example illustrates the use of Theorem 4 in decomposing the proof of an implementation relationship where the implementation and specification are not merely composition of automata but composition of automata that satisfy some liveness property.

Let *UseOldInputA'*, *UseOldInputB'*, *UseNewInputA'*, and *UseNewInputB'* be automata which are defined exactly as automata *UseOldInputA*, *UseOldInputB*, *UseNewInputA*, and *UseNewInputB* from Example 2 except that there is no bound on the number of outputs that the automata can perform. That is, *maxout* is removed from their sets of state variables. Let P_1, P_2, Q_1 and Q_2 be properties for, respectively, *UseNewInputA'*, *UseOldInputA'*, *UseNewInputB'* and *UseOldInputB'* defined as follows:

- P_1 consists of the admissible execution fragments of *UseNewInputA'*.
- Q_1 consists of the admissible execution fragments of *UseNewInputB'*.
- P_2 consists of the execution fragments of *UseOldInputA'* that contain infinitely many a actions.
- Q_2 consists of the execution fragments of *UseOldInputB'* that contain infinitely many b actions.

Suppose that we want to prove that:

$(\mathit{UseNewInputA'}, P_1) \parallel (\mathit{UseNewInputB'}, Q_1) \leq (\mathit{UseOldInputA'}, P_2) \parallel (\mathit{UseOldInputB'}, Q_2)$.

The automata *UseNewInputA'* $\parallel$ *UseNewInputB'* and *UseOldInputA'* $\parallel$ *UseOldInputB'* perform an alternating sequence of a and b actions. The properties express the additional condition that as time goes to infinity the composite automaton *UseNewInputA'* $\parallel$ *UseNewInputB'* performs infinitely many a and infinitely many b actions where a and b actions alternate.

As in Example 2 automata *AlternateA* and *AlternateB* from Figure 1 satisfy the required closure properties for auxiliary automata and capture what is essential about the safety part of the proof, namely that the environments of *UseNewInputA'* and *UseNewInputB'* impose alternation. The essential point in the proof of the liveness part is that each automaton responds to each input it receives from its environment. Therefore, we need to pair *AlternateA* and *AlternateB* with properties that eliminate non-responding behavior. The properties P_3 and Q_3 defined below satisfy this condition:

- P_3 consists of execution fragments α of *AlternateA* that satisfy the following condition: if α has finitely many actions then the last action in α is a.
- Q_3 consists of execution fragments α of *AlternateB* that satisfy the following condition: if α has finitely many actions and contains at least one a then the last action in α is b.

In order to see why the first part of Assumption 5 is satisfied we can inspect the definition of *UseNewInputA* and observe that *UseNewInputA* performs an output a one time unit after each input b, when it is composed with *AlternateB*. This implies that in any admissible execution fragment of *UseNewInputA* $\parallel$ *AlternateB* with finitely many actions the last action must be a. This is exactly the liveness constraint expressed by P_3. The second part of Assumption 5 can be seen to hold using a symmetric argument.

In order to see why the first part of Assumption 6 holds consider any execution fragment β of *UseNewInputA* $\|$ *AlternateB*. For β to satisfy P_1 and Q_3 at the same time, it must consist of an infinite sequence in which a and b actions alternate. It is not possible for *UseNewInputA* $\|$ *AlternateB* to have an admissible execution fragment with finitely many actions because the definition of *UseNewInputA* requires such a sequence to end in a while this is ruled out by Q_3, which requires *AlternateB* to respond to a. The second part of Assumption 6 can be seen to hold using a symmetric argument.

Note that in our explanations we refer to execution fragments rather than traces of execution fragments. This is because our examples do not include any internal actions and our arguments for execution fragments extend to trace fragments in a straightforward way.

5 Conclusions and future work

In this paper we have focused on compositionality for timed I/O automata. In particular, we have presented three assume-guarantee style substitutivity results for the composition operation of timed I/O automata. We believe that these results are simple and easy to understand; they build upon the basic concepts about TIOAs and the fundamental results for the composition operation. Unlike many of the related results obtained for other formal frameworks, no complex logical operators or induction principles are needed for our theorem statements and their proofs.

Theorem 4 suggests a useful way of separating out proof obligations for safety and liveness in decomposing verification tasks that involve TIOAs paired with liveness properties. A proof based on Theorem 1 that shows that an implementation satisfies a safety specification can in large part be reused when liveness conditions are added to the specification.

Our main goal in this line of work was to obtain simple and general compositionality results. As future work we intend to explore the implications on our results of considering special kinds of automata and properties that we have defined in [KLSV03a]. For example, it would be interesting to know if any of the assumptions of our theorems would be implied if we considered receptive TIOAs, I/O feasible TIOAs or I/O liveness properties.

Our current results apply to trace inclusion preorder. Another interesting direction for future work would be to extend these results to other preorders based on various notions of simulations relations defined in [KLSV03a].

Acknowledgements. This work is supported by DARPA/AFOSR MURI Contract F49620-02-1-0325. We thank Frits Vaandrager and Roberto Segala for their comments on the preliminary versions of our results.

References

[AH97] R. Alur and T. Henzinger. Modularity for timed and hybrid systems. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *LNCS*, pages 74–88. Springer-Verlag, 1997.

[AK96] R. Alur and R. P. Kurshan. Timing analysis in COSPAN. In *Hybrid Systems III: Verification and Control*. Springer-Verlag, 1996.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 1(15):73–132, 1993.

[AL94] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.
[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.
[HQR00] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 245–252. IEEE Computer Society Press, 2000.
[HQR02] T. Henzinger, S. Qadeer, and S. K. Rajamani. An assume-guarantee rule for checking simulation. *ACM Transactions on Programming Languages and Systems*, 24:51–64, 2002.
[Jon83] C. B. Jones. Specification and design of parallel programs. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. North-Holland, 1983.
[KLSV03a] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Technical Report MIT/LCS/TR-917, MIT Laboratory for Computer Science, 2003. Available at `http://theory.lcs.mit.edu/tds/reflist.html`.
[KLSV03b] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 166–177, Cancun, Mexico, 2003. IEEE Computer Society. Full version available as Technical Report MIT/LCS/TR-917.
[MWLF03] S. Mitra, Y. Wang, N. Lynch, and E. Feron. Safety verification of pitch controller for model helicopter. In O. Maler and A. Pnueli, editors, *Proc. of Hybrid Systems: Computation and Control*, volume 2623 of *Lecture Notes in Computer Science*, pages 343–358, Prague, the Czech Republic April 3-5, 2003.
[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logis and Models of Concurret Systems*, NATO ASI, pages 123–144. Springer-Verlag, 1984.
[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *LNCS*, pages 369–391. Springer-Verlag, 1985.
[TAKB96] S. Tasiran, R. Alur, R.P. Kurshan, and R.K. Brayton. Verifying abstractions of timed systems. In *Proceedings of the Seventh Conference on Concurrency Theory (CONCUR)*, volume 1119 of *LNCS*, 1996.

A Proofs

Proof of Theorem 4: Let $\beta \in traces_{(\mathcal{A}_1,P_1)\|(\mathcal{B}_1,Q_1)}$. By definition of composition for automata with properties, $\beta \in traces_{(\mathcal{A}_1\|\mathcal{B}_1)}$. By Assumptions 1, 2, 3 and 4 and Theorem 1, we have $\beta \in traces_{(\mathcal{A}_2\|\mathcal{B}_2)}$. By projection using Theorem 1, $\beta\lceil(E_{\mathcal{A}_2},\emptyset) \in traces_{\mathcal{A}_2}$ and $\beta\lceil(E_{\mathcal{B}_2},\emptyset) \in traces_{\mathcal{B}_2}$. By Assumption 3, $\beta\lceil(E_{\mathcal{A}_2},\emptyset) \in traces_{\mathcal{A}_3}$ and $\beta\lceil(E_{\mathcal{B}_2},\emptyset) \in traces_{\mathcal{B}_3}$. Since $\mathcal{A}_2$ and $\mathcal{A}_3$ are comparable, $\beta\lceil(E_{\mathcal{A}_2},\emptyset) = \beta\lceil(E_{\mathcal{A}_3},\emptyset)$ and $\beta\lceil(E_{\mathcal{B}_2},\emptyset) = \beta\lceil(E_{\mathcal{B}_3},\emptyset)$. Therefore, $\beta\lceil(E_{\mathcal{A}_3},\emptyset) \in traces_{\mathcal{A}_3}$ and $\beta\lceil(E_{\mathcal{B}_3},\emptyset) \in traces_{\mathcal{B}_3}$.

By projection using Theorem 2, we have $\beta\lceil(E_{\mathcal{A}_1},\emptyset) \in traces_{(\mathcal{A}_1,P_1)}$ and $\beta\lceil(E_{\mathcal{B}_1},\emptyset) \in traces_{(\mathcal{B}_1,Q_1)}$. By pasting using Theorem 2, we have $\beta \in traces_{(\mathcal{A}_1,P_1)\|(\mathcal{B}_3,frags_{\mathcal{B}_3})}$ and $\beta \in traces_{(\mathcal{B}_1,Q_1)\|(\mathcal{A}_3,frags_{\mathcal{A}_3})}$. By Assumption 5, we have $\beta \in traces_{(\mathcal{A}_3,P_3)\|(\mathcal{B}_3,frags_{\mathcal{B}_3})}$ and $\beta \in traces_{(\mathcal{B}_3,Q_3)\|(\mathcal{A}_3,frags_{\mathcal{A}_3})}$. By projection using Theorem 2, we get $\beta\lceil(E_{\mathcal{A}_3},\emptyset) \in traces_{(\mathcal{A}_3,P_3)}$ and $\beta\lceil(E_{\mathcal{B}_3},\emptyset) \in traces_{(\mathcal{B}_3,Q_3)}$. Since $\beta\lceil(E_{\mathcal{A}_1},\emptyset) \in traces_{(\mathcal{A}_1,P_1)}$, by pasting using Theorem 2, we have $\beta \in traces_{(\mathcal{A}_1,P_1)\|(\mathcal{B}_3,Q_3)}$, similarly since $\beta\lceil(E_{\mathcal{B}_1},\emptyset) \in traces_{(\mathcal{B}_1,Q_1)}$, we have $\beta \in traces_{(\mathcal{B}_1,Q_1)\|(\mathcal{A}_3,P_3)}$. By Assumption 6, we have $\beta \in (\mathcal{A}_2,P_2)\|(\mathcal{B}_3,Q_3)$ and $\beta \in (\mathcal{A}_3,P_3)\|(\mathcal{B}_2,Q_2)$. By projection pasting using Theorem 2, $\beta\lceil(E_{\mathcal{A}_2},\emptyset) \in traces_{(\mathcal{A}_2,P_2)}$ and $\beta\lceil(E_{\mathcal{B}_2},\emptyset) \in traces_{(\mathcal{B}_2,Q_2)}$. By pasting using Theorem 2, it follows that $\beta \in (\mathcal{A}_2,P_2)\|(\mathcal{B}_2,Q_2)$, as needed.

B Specifications of Automata Used in Examples

Automaton $AlternateA$

Variables X : **discrete** $myturn \in Bool$ **initially** true

States Q : $val(X)$

Actions A : **input** b, **output** a

Transitions $\mathcal{D}$:

input b
  effect
    $myturn := true$

output a
  precondition
    $myturn$
  effect
    $myturn := false$

Trajectories $\mathcal{T}$: **satisfies**
  constant$(myturn)$

Automaton $AlternateB$

Variables X : **discrete** $myturn \in Bool$ **initially** false

States Q : $val(X)$

Actions A : **input** a, **output** b

Transitions $\mathcal{D}$:

input a
  effect
    $myturn := true$

output b
  precondition
    $myturn$
  effect
    $myturn := false$

Trajectories $\mathcal{T}$: **satisfies**
  constant$(myturn)$

Fig. 1. Example automata for $\mathcal{A}_2$ and $\mathcal{B}_2$ in Theorem 3

Automaton $CatchUpA$

Variables X : **discrete** $counta, countb \in \mathsf{N}$ **initially** 0
analog $now \in \mathsf{R}^{\geq 0}$ **initially** 0
analog $next \in \mathsf{R}^{\geq 0} \cup \{\infty\}$ **initially** 0

States Q : $val(X)$

Actions A : **input** b, **output** a

Transitions $\mathcal{D}$:

input b
  effect
    $countb := countb + 1$
    $next := now + 1$

output a
  precondition
    $counta \leq countb \wedge now = next$
  effect
    $counta := counta + 1$
    $next := now + 1$

Trajectories $\mathcal{T}$: **satisfies**
  constant(counta,countb)
stops when
  $now = next$

Automaton $CatchUpB$

Variables X : **discrete** $counta, countb \in \mathsf{N}$ **initially** 0
analog $now \in \mathsf{R}^{\geq 0}$ **initially** 0
analog $next \in \mathsf{R}^{\geq 0} \cup \{\infty\}$ **initially** 0

States Q : $val(X)$

Actions A : **input** a, **output** b, **internal** c

Transitions $\mathcal{D}$:

input a
  effect
    $counta := counta + 1$
    $next := now + 1$

output b
  precondition
    $countb + 1 \leq counta \wedge now = next$
  effect
    $countb := countb + 1$
    $next = now + 1$

Trajectories $\mathcal{T}$: **satisfies**
  constant(counta,countb)
stops when
  $now = next$

Fig. 2. Example automata $\mathcal{A}_1$ and $\mathcal{B}_1$ for Theorem 3

Automaton $UseOldInputA$

Variables X : **discrete** $myturn \in Bool$ **initially** true
discrete $maxout \in \mathsf{N}$ **initially** arbitrary
analog $now \in \mathsf{R}^{\geq 0}$ **initially** 0
analog $next \in \mathsf{R}^{\geq 0} \cup \{\infty\}$ **initially** 0

States Q : $val(X)$

Actions A : **input** b, **output** a

Transitions $\mathcal{D}$:

input b
  effect
    $myturn := true$
    if $next = \infty$
    then $next := now + 1$

output a
  precondition
    $myturn \ \wedge \ (maxout > 0) \ \wedge \ (now = next)$
  effect
    $myturn := false$
    $maxout := maxout - 1$
    $next := \infty$

Trajectories $\mathcal{T}$: **satisfies**
  constant$(myturn, maxout, next)$
  $\mathbf{d}(now) = 1$
stops when
  $now = next$

Automaton $UseOldInputB$

Variables X : **discrete** $myturn \in Bool$ **initially** false
discrete $maxout \in \mathsf{N}$ **initially** arbitrary
analog $now \in \mathsf{R}^{\geq 0}$ **initially** 0
analog $next \in \mathsf{R}^{\geq 0} \cup \{\infty\}$ **initially** 0

States Q : $val(X)$

Actions A : **input** a, **output** b

Transitions $\mathcal{D}$:

input a
  effect
    $myturn := true$
    if $next = \infty$
    then $next := now + 1$

output b
  precondition
    $myturn \ \wedge \ (maxout > 0) \ \wedge \ (now = next)$
  effect
    $myturn := false$
    $maxout := maxout - 1$
    $next := \infty$

Trajectories $\mathcal{T}$: **satisfies**
  constant$(myturn, maxout, next)$
  $\mathbf{d}(now) = 1$
stops when
  $now = next$

Fig. 3. Example automata for $\mathcal{A}_2$ and $\mathcal{B}_2$ in Theorem 1

Automaton $UseNewInputA$

Variables X : **discrete** $myturn \in Bool$ **initially** true
discrete $maxout \in \mathsf{N}$ **initially** arbitrary
analog $now \in \mathsf{R}^{\geq 0}$ **initially** 0
analog $next \in \mathsf{R}^{\geq 0} \cup \{\infty\}$ **initially** 0

States Q : $val(X)$

Actions A : **input** b, **output** a

Transitions $\mathcal{D}$: **input** b
effect
$myturn := true$
$next := now + 1$

output a
precondition
$myturn \;\wedge\; (maxout > 0) \;\wedge\; (now = next)$
effect
$myturn := false$
$maxout := maxout - 1$
$next := \infty$

Trajectories $\mathcal{T}$: **satisfies**
constant$(myturn, maxout, next)$
$\mathbf{d}(now) = 1$
stops when
$now = next$

Automaton $UseNewInputA$

Variables X : **discrete** $myturn \in Bool$ **initially** false
discrete $maxout \in \mathsf{N}$ **initially** arbitrary
analog $now \in \mathsf{R}^{\geq 0}$ **initially** 0
analog $next \in \mathsf{R}^{\geq 0} \cup \{\infty\}$ **initially** 0

States Q : $val(X)$

Actions A : **input** a, **output** b

Transitions $\mathcal{D}$: **input** a
effect
$myturn := true$
$next := now + 1$

output b
precondition
$myturn \;\wedge\; (count > 0) \;\wedge\; (now = next)$
effect
$myturn := false$
$maxout := maxout - 1$
$next := \infty$

Trajectories $\mathcal{T}$: **satisfies**
constant$(myturn, maxout, next)$
$\mathbf{d}(now) = 1$
stops when
$now = next$

Fig. 4. Example automata for $\mathcal{A}_1$ and $\mathcal{B}_1$ in Theorem 1