Timed Multiparty Session Types*

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida

Imperial College London, London, UK

Abstract. We propose a typing theory, based on multiparty session types, for modular verification of real-time choreographic interactions. To model real-time implementations, we introduce a simple calculus with delays and a decidable static proof system. The proof system ensures type safety and time-error freedom, namely processes respect the prescribed timing and causalities between interactions. A decidable condition on timed global types guarantees time-progress for validated processes with delays, and gives a sound and complete characterisation of a new class of CTAs with general topologies that enjoys progress and liveness.

1 Introduction

Communicating timed automata (CTAs) [14] extend the theory of timed automata [3] to enable a precise specification and verification of real-time distributed protocols. A CTA consists of a finite number of timed automata synchronising over the elapsing of time and exchanging messages over unbound channels. In spite of its simplicity, the combination of timed automata [3] and communicating automata (CAs) [8] can represent many different temporal aspects from a local viewpoint. On the other hand, the model is known to be computationally hard, and it is difficult to directly link its idealised semantics to implementations of programming languages and distributed systems.

On a parallel line of research, *multiparty session types* (MPSTs) [13, 6] have been proposed to describe communication protocols among two or more participants from a global viewpoint. Global types are projected to local types, against which programs can be type-checked and verified to behave correctly without deadlocks. This framework is applied in industry projects [19] and to the governance of large cyberinfrastructures [17] via the Scribble project (a MPST-based tool chain) [20].

From the theoretical side, in the untimed setting recent work brings CAs into choreographic frameworks, by seeking a correspondence with projected local types [11]. We proceed along these lines by applying the idealised mathematical semantics of CTAs to the design of MPSTs with clocks, clock constraints, and resets, in order to fill the gap between the abstract specification by CTAs and the verification of real-time programs. Surprisingly, since MPSTs inherently capture relative temporal constraints by imposing an order on the communications, they enable effective verification without limitations on topology or buffer-boundedness, unlike existing work on CTAs.

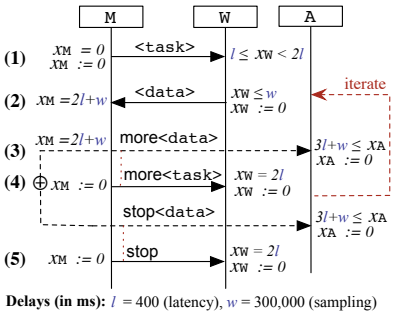
We organise our results in two parts. First we show that although time annotations increase the expressive power of global types, time-error freedom is guaranteed without additional time analysis of the types. In § 3 we give the semantics of timed global

* This work has been partially sponsored by EPSRC EP/K034413/1 and EP/K011715/1. We thank Viviana Bono and Mariangiola Dezani-Ciancaglini for their insightful comments.

types (*TG*s) and prove soundness and completeness of the projection onto timed local types (*TL*s) (Theorem 3). In § 4 we give a simple π-calculus for programs (running as *processes*) with delays that can be used to synchronise the communications in a session. A compositional proof system enables modular verification of time-error freedom (Theorem 7): if all programs in a system are validated, then the global conversation will respect the prescribed timing and causalities between interactions. In the second part we investigate the conditions for an advanced property – time-progress – ensuring that if a process deadlocks, then its untimed counter-part would also deadlock (i.e., deadlock is not caused by time constraints). The fact that untimed processes in single sessions are deadlock-free [13] yields progress for timed processes. Time-progress is related to two delicate issues: (1) some time constraints in a *TG* may be unsatisfiable and (2) there may exist some distributed implementation of the *TG* which deadlocks. We give two sufficient conditions on *TG*s (§5) to prevent (1) and (2): *feasibility* (for each partial execution allowed by a *TG* there is a correct complete one) and *wait-freedom* (if all senders respect their time constraints, then no receiver has to wait for a message). Feasibility and wait-freedom are decidable (Proposition 8), and if we start from feasible and wait-free *TG*s, then the proof system given in part one guarantees time-progress for processes (Theorem 11). We give a sound and complete characterisation (Theorem 13) yielding a new class of CTAs which enjoys progress and liveness (Theorem 14). Conclusion and related work are in § 6. Full definitions can be found in the technical report [22].

2 Running Example: a Use-Case of a Distributed Timed Protocol

The motivating scenario developed with our partner, the Ocean Observatories Initiative (OOI) [17], is directed towards deploying a network of sensors and ocean instruments used/controlled remotely via service agents. In many OOI use-cases requests are augmented with deadlines and services are scheduled to execute at certain time intervals. These temporal requirements can be expressed by combining global protocol descriptions from MPSTs and time from CTAs. We show a protocol to calculate the average water temperature via sensor sampling. The protocol involves three participants: a master `M` that initiates the sampling, a worker/sensor `W` with fixed response time w, and an aggregator `A` for accumulating the data; their time constraints are expressed using clocks $x_{\mathtt{M}}$, $x_{\mathtt{W}}$, and $x_{\mathtt{A}}$, initially set to 0. Each clocks can be reset many times. Delays l (average latency of the network) and w (sampling time) are expressed in milliseconds. As in [14] (synchronous semantics) time elapses at the same pace for all the parts of the system.



(1): `M` sends `W` a message of type `task` and resets $x_{\mathtt{M}}$ ($x_{\mathtt{M}}$:=0). After at least l and at most $2l$, `W` receives the message.

(2): `W` completes the task and replies to `M` with the temperature (of type `data`) at any time satisfying $x_{\mathtt{W}} \leq w$ and resets $x_{\mathtt{W}}$. `M` receives the message at time $x_{\mathtt{M}} = 2l + w$.

(3): `M` immediately sends `A` a message of type `data` with either label more (the sampling continues for another iteration) or stop (enough results have been aggregated).

(4): If more was selected then `M` sends `W` a new task with label more, resets $x_{\mathtt{M}}$ and another iteration is executed.

(5): If stop was selected, `M` notifies `W` and the session ends.

3 Timed Multiparty Session Types

Global types [6, 13] are specifications of the interactions (causalities and carried data types) of multiparty sessions. A global type can be automatically projected onto a set of local types describing the session from the perspective of each single participant and used for local verification of processes. We extend global and local types with constraints on clocks, yielding *timed global types* (*TG*s) and *local session types* (*TL*s).

We use the definitions from timed automata (see [3, § 3.3], [14, § 2]): let X be a set of *clocks* ranging over $x_1, \ldots, x_n$ and taking values in $\mathbb{R}^{\geq 0}$. A *clock assignment* $\nu : X \mapsto \mathbb{R}^{\geq 0}$ returns the time of the clocks in X. We write $\nu + t$ for the assignment mapping all $x \in X$ to $\nu(x) + t$. We write ν_0 for the initial assignment mapping all clocks to 0. The set $\Phi(X)$ of *clock constraints* over X is:

$$\delta ::= \mathtt{true} \mid x > c \mid x = c \mid \neg\delta \mid \delta_1 \wedge \delta_2$$

where c is a bound time constant taking values in $\mathbb{Q}^{\geq 0}$ (we derive $\mathtt{false}$, $<$, $\leq$, $\geq$, $\vee$ in the standard way). The set of free clocks in δ, written $fn(\delta)$, is defined inductively as: $fn(\mathtt{true}) = \emptyset$, $fn(x > c) = fn(x = c) = \{x\}$, $fn(\neg\delta) = fn(\delta)$, and $fn(\delta_1 \wedge \delta_2) = fn(\delta_1) \cup fn(\delta_2)$. We write $\delta(\vec{x})$ if $fn(\delta) = \vec{x}$ and let $\nu \models \delta$ denote that δ is satisfied by ν. A *reset* λ over X is a subset of X. When λ is $\emptyset$ then clocks are not reset, otherwise the assignment for each $x \in \lambda$ is set to 0. We write $[\lambda \mapsto 0]\nu$ for the clock assignment that is like ν except 0 is assigned to all clocks in λ.

Participants ($\mathtt{p}, \mathtt{q}, \mathtt{p}_1, \ldots \in \mathbb{N}$) interact via point-to-point asynchronous message passing. An interaction consists of a send action and a receive action, each annotated with a clock constraint and a reset. The clock constraint specifies when that action can be executed and the reset specifies which clocks must be set to 0.

Syntax. The syntax for sorts S, timed global types G, and timed local types T is:

$$\begin{array}{lll} S & ::= & \mathtt{bool} \mid \mathtt{nat} \mid \ldots \mid G \mid (T, \delta) \\ G & ::= & \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{A_i\}.G_i\}_{i\in I} \mid \mu\mathbf{t}.G \mid \mathbf{t} \mid \mathtt{end} \qquad\qquad A ::= \{\delta_{\mathtt{O}}, \lambda_{\mathtt{O}}, \delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\} \\ T & ::= & \mathtt{p} \oplus \{l_i : \langle S_i\rangle\{B_i\}.T_i\}_{i\in I} \mid \mathtt{p}\&\{l_i : \langle S_i\rangle\{B_i\}.T_i\}_{i\in I} \mid \mu\mathbf{t}.T \mid \mathbf{t} \mid \mathtt{end} \quad B ::= \{\delta, \lambda\} \end{array}$$

The sorts S include base types ($\mathtt{bool}$, $\mathtt{nat}$, etc.), G for shared name passing (used for the initiation of sessions of type G, cf. § 4), and (T, δ) for session delegation. Sort (T, δ) allows a participant involved in a session to delegate the remaining behaviour T; upon delegation the sender will no longer participate in the delegated session and receiver will execute the protocol described by T under any clock assignment satisfying δ. G and T in sorts do not include free type variables.

In G, type $\mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{A_i\}.G_i\}_{i\in I}$ models an *interaction*: $\mathtt{p}$ chooses a branch $i \in I$, where I is a finite set of indices, and sends $\mathtt{q}$ the branching label l_i along with a message of sort S_i. The session then continues as prescribed by G_i. Each branch is annotated with a *time assertion* $A_i = \{\delta_{\mathtt{O}i}, \lambda_{\mathtt{O}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}$, where $\delta_{\mathtt{O}i}$ and $\lambda_{\mathtt{O}i}$ are the clock constraint and reset for the output action, and $\delta_{\mathtt{I}i}$ and $\lambda_{\mathtt{I}i}$ are for the input action. We will write $\mathtt{p} \to \mathtt{q} : \langle S\rangle\{A\}.G'$ for interactions with one branch. *Recursive type* $\mu\mathbf{t}.G$ associates a *type variable* $\mathbf{t}$ to a recursion body G; we assume that type variables are guarded in the standard way and $\mathtt{end}$ occurs at least once in G (this is a common assumption e.g., [9]). We denote by $\mathcal{P}(G)$ the set of participants of G and write $G' \in G$ when G' appears in G.

As in [14] we assume that the sets of clocks 'owned' (i.e., that can be read and reset) by different participants in a *TG* are pair-wise disjoint, and that the clock constraint and reset of an action performed by a participant are defined only over the clocks owned by that participant. The example below violates this assumption.

$$G_1 = \mathtt{p} \to \mathtt{q} : \langle \mathtt{int} \rangle \{x_\mathtt{p} < 10, x_\mathtt{p}, x_\mathtt{p} < 20, x_\mathtt{p}\}$$

since both the constraints of the (send) action of p and of the (receive) action of q are defined over $x_\mathtt{p}$, and $x_\mathtt{p}$ can be owned by either p or q (similarly for the resets $\{x_\mathtt{p}\}$). Formally, we require that for all G there exists a partition $\{X(\mathtt{p}, G)\}_{\mathtt{p} \in \mathcal{P}(G)}$ of X such that $\mathtt{p} \to \mathtt{q} : \{l_i \langle S_i \rangle \{\delta_{\mathtt{O}i}, \lambda_{\mathtt{O}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}.G_i\}_{i \in I} \in G$ implies $fn(\delta_{\mathtt{O}i}), \lambda_{\mathtt{O}i} \subseteq X(\mathtt{p}, G)$ and $fn(\delta_{\mathtt{I}i}), \lambda_{\mathtt{I}i} \subseteq X(\mathtt{q}, G)$ for all $i \in I$.

In T, interactions are modelled from a participant's viewpoint either as *selection* types $\mathtt{p} \oplus \{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I}$ or *branching* types $\mathtt{p}\&\{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I}$. We denote the *projection* of G on $\mathtt{p} \in \mathcal{P}(G)$ by $G \downarrow_\mathtt{p}$; the definition is standard except that each $\{\delta_{\mathtt{O}i}, \lambda_{\mathtt{O}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}$ is projected on the sender (resp. receiver) by keeping only the output part $\{\delta_{\mathtt{O}i}, \lambda_{\mathtt{O}i}\}$ (resp. the input part $\{\delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}$), e.g., if $G = \mathtt{p} \to \mathtt{q} : \{l_i \langle S_i \rangle \{B_i, B_i'\}.G_i\}_{i \in I}$ then $G \downarrow_\mathtt{p} = \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{B_i\}.G_i \downarrow_\mathtt{p}\}_{i \in I}$ and $G \downarrow_\mathtt{q} = \mathtt{p}\&\{l_i : \langle S_i \rangle \{B_i'\}.G_i \downarrow_\mathtt{q}\}_{i \in I}$.

Example 1 (Temperature calculation) We show below the global timed type G for the protocol in § 2 and its projection $G \downarrow_\mathtt{M}$ onto M. We write $\_$ for empty resets.

$$\begin{array}{lll}
G & = \mathtt{M} \to \mathtt{W} : \langle \mathtt{task} \rangle \{B_\mathtt{O}^1, B_\mathtt{I}^1\}.\mu \mathbf{t}.G' & B_\mathtt{O}^1 = \{x_\mathtt{M} = 0, x_\mathtt{M}\} \\
G' & = \mathtt{W} \to \mathtt{M} : \langle \mathtt{data} \rangle \{B_\mathtt{O}^2, B_\mathtt{I}^2\}. & B_\mathtt{I}^1 = \{l \le x_\mathtt{W} < 2l, \ _\} \\
 & \quad \mathtt{M} \to \mathtt{A} : \{\mathsf{more}\langle \mathtt{data} \rangle \{B_\mathtt{O}^3, B_\mathtt{I}^3\}.\ \mathtt{M} \to \mathtt{W} : \mathsf{more}\langle \mathtt{task} \rangle \{B_\mathtt{O}^4, B_\mathtt{I}^4\}.\mathbf{t}, & B_\mathtt{O}^2 = \{x_\mathtt{W} \le w, x_\mathtt{W}\} \\
 & \qquad \mathsf{stop}\langle \mathtt{data} \rangle \{B_\mathtt{O}^3, B_\mathtt{I}^3\}.\ \mathtt{M} \to \mathtt{W} : \mathsf{stop}\langle\rangle \{B_\mathtt{O}^4, B_\mathtt{I}^4\}.\mathtt{end}\} & B_\mathtt{I}^2 = \{x_\mathtt{M} = 2l + w, \ _\} \\
G \downarrow_\mathtt{M} & = \mathtt{W} \oplus \langle \mathtt{task} \rangle \{B_\mathtt{O}^1\}. & B_\mathtt{O}^3 = \{x_\mathtt{M} = 2l + w, \ _\} \\
 & \quad \mu \mathbf{t}.\ \mathtt{W}\&\langle \mathtt{data} \rangle \{B_\mathtt{I}^2\}. & B_\mathtt{I}^3 = \{3l + w \le x_\mathtt{A}, x_\mathtt{A}\} \\
 & \quad \mathtt{A} \oplus \{\mathsf{more} : \langle \mathtt{data} \rangle \{B_\mathtt{O}^3\}.\mathtt{W} \oplus \mathsf{more} : \langle \mathtt{task} \rangle \{B_\mathtt{O}^4\}.\mathbf{t}, & B_\mathtt{O}^4 = \{x_\mathtt{M} = 2l + w, x_\mathtt{M}\} \\
 & \qquad \mathsf{stop} : \langle \mathtt{data} \rangle \{B_\mathtt{O}^3\}.\mathtt{W} \oplus \mathsf{stop} : \langle\rangle \{B_\mathtt{O}^4\}.\mathtt{end}\} & B_\mathtt{I}^4 = \{x_\mathtt{W} = 2l, x_\mathtt{W}\}
\end{array}$$

Remark 1 (On the importance of resets). Resets in timed global types play an important role to model the same notion of time as the one supported by CTAs, yielding a more direct comparison between types and CTAs. Resets give a concise representation of several scenarios, e.g., when time constraints must be repeatedly satisfied for an unbounded number of times. This is clear from Example 1: the repetition of the same scenario across recursion instances (one for each sampling task) is modelled by resetting all clocks before starting a new recursion instance (e.g., $B_\mathtt{I}^3$, $B_\mathtt{O}^4$ and $B_\mathtt{I}^4$ on the second line of G' in Example 1).

Semantics of timed global types. The LTS for *TG*s is defined over states of the form (ν, G) and labels $\ell ::= \mathtt{pq}!l\langle S \rangle \mid \mathtt{pq}?l\langle S \rangle \mid t$ where $\mathtt{pq}!l\langle S \rangle$ is a send action (i.e., p sends $l\langle S \rangle$ to q), $\mathtt{pq}?l\langle S \rangle$ is the dual receive action, and $t \in \mathbb{R}^{\geq 0}$ is a time action modelling time passing. We denote the set of labels by $\mathcal{L}$ and let $subj(\mathtt{pq}!l\langle S \rangle) = \mathtt{p}$, $subj(\mathtt{qp}?l\langle S \rangle) = \mathtt{p}$ and $subj(t) = \emptyset$.

We extend the syntax of G with $\mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S \rangle \{A\}.G$ to describe the state in which message $l\langle S \rangle$ has been sent by p but not yet received by q (as in [11, § 2]). The separation of send and receive actions is used to model the asynchronous behaviour in distributed systems, as illustrated by the following example.

$$\begin{aligned}
&\mathtt{p} \to \mathtt{q} : \langle \mathtt{int} \rangle \{x_\mathtt{p} < 10,\ _ , x_\mathtt{q} \geq 10,\ _ \}.\mathtt{p} \to \mathtt{r} : \langle \mathtt{int} \rangle \{x_\mathtt{p} \geq 10,\ _ , \mathtt{true},\ _ \} \\
\xrightarrow{\mathtt{pq}!\langle \mathtt{int} \rangle} \ &\mathtt{p} \rightsquigarrow \mathtt{q} : \langle \mathtt{int} \rangle \{x_\mathtt{p} < 10,\ _ , x_\mathtt{q} > 20,\ _ \}.\mathtt{p} \to \mathtt{r} : \langle \mathtt{int} \rangle \{x_\mathtt{p} < 10,\ _ , \mathtt{true},\ _ \} \\
\xrightarrow{\mathtt{pr}!\langle \mathtt{int} \rangle} \ &\mathtt{p} \rightsquigarrow \mathtt{q} : \langle \mathtt{int} \rangle \{x_\mathtt{p} < 10,\ _ , x_\mathtt{q} > 20,\ _ \}.\mathtt{p} \rightsquigarrow \mathtt{r} : \langle \mathtt{int} \rangle \{x_\mathtt{p} \geq 10,\ _ , \mathtt{true},\ _ \}
\end{aligned}$$

After the first action $\mathtt{pq}!\langle \mathtt{int} \rangle$ the *TG* above can reduce by one of the following actions: a send $\mathtt{pr}!\langle \mathtt{int} \rangle$ (as illustrated), a receive of $\mathtt{q}$, or a time step. By using intermediate states, a send action and its corresponding receive action (e.g., $\mathtt{pq}!\langle \mathtt{int} \rangle$ and $\mathtt{pq}?\langle \mathtt{int} \rangle$) are separate, hence could be interleaved with other actions, as well as occur at different times. This fine-grained semantics corresponds to local type semantics where asynchrony is modelled as message exchange through channels (see Theorem 3).

*TG*s are used as a model of the correct behaviour for distributed implementations in § 4. Therefore their semantics should only include desirable executions. We need to take special care in the definition of the semantics of time actions: if an action is ready to be executed and the associated constraint has an upper bound, then the semantics should prevent time steps that are too big and would make that clock constraint unsatisfiable. For instance in $\mathtt{p} \to \mathtt{q} : \langle \mathtt{int} \rangle \{x_\mathtt{p} \leq 20,\ \_ , \mathtt{true},\ \_ \}$ (assuming $x_\mathtt{p} = 0$) the LTS should allow, before the send action of $\mathtt{p}$ occurs, only time steps that preserve $x_\mathtt{p} \leq 20$.

More generally, we need to ensure that time actions do not invalidate the constraint of any action that is ready to be executed, or *ready action*. A ready action is an action that has no causal relationship with other actions that occur earlier, syntactically. A *TG* may have more than one ready action, as shown by the following example.

$$\mathtt{p} \to \mathtt{q} : \langle \mathtt{int} \rangle \{x_\mathtt{p} \leq 20,\ \_ , \mathtt{true},\ \_ \}.\mathtt{k} \to \mathtt{r} : \langle \mathtt{int} \rangle \{x_\mathtt{k} < 10,\ \_ , x_\mathtt{r} = 10,\ \_ \}$$

The *TG* above has two ready actions, namely the send actions of $\mathtt{p}$ and of $\mathtt{k}$ which can happen in any order due to asynchrony (i.e., an order cannot be enforced without extra communications between $\mathtt{p}$ and $\mathtt{k}$). In this case a desirable semantics should prevent the elapsing of time intervals that would invalidate either $\{x_\mathtt{p} \leq 20\}$ or $\{x_\mathtt{k} < 10\}$.

Below, function $\mathtt{rdy}(G, D)$ returns the set, for each ready actions in G, of elements of the form $\{\delta_i\}_{i \in I}$ which are the constraints of the branches of that ready action. D is a set of participants, initially empty, used to keep track of the causal dependencies between actions. We write $\mathtt{rdy}(G)$ for $\mathtt{rdy}(G, \emptyset)$.

$$(1)\ \begin{array}{c}\mathtt{rdy}(\mathtt{p} \to \mathtt{q} : \{l_i \langle S_i \rangle \{A_i\}.G_i\}_{i \in I}, D) \\ (\text{with } A_i = \{\delta_{\mathtt{O}i}, \lambda_{\mathtt{O}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\})\end{array} = \begin{cases} \{\{\delta_{\mathtt{O}i}\}_{i \in I}\} \bigcup_{i \in I} \mathtt{rdy}(G_i, D \cup \{\mathtt{p}, \mathtt{q}\}) \text{ if } \mathtt{p} \notin D \\ \bigcup_{i \in I} \mathtt{rdy}(G_i, D \cup \{\mathtt{p}, \mathtt{q}\}) \quad \text{otherwise} \end{cases}$$

$$(2)\ \mathtt{rdy}(\mathtt{p} \rightsquigarrow \mathtt{q} : l \langle S \rangle \{\delta_\mathtt{O}, \lambda_\mathtt{O}, \delta_\mathtt{I}, \lambda_\mathtt{I}\}.G, D) = \begin{cases} \{\{\delta_\mathtt{I}\}\} \cup \mathtt{rdy}(G, D \cup \{\mathtt{q}\}) \quad \text{if } \mathtt{q} \notin D \\ \mathtt{rdy}(G, D \cup \{\mathtt{q}\}) \quad \text{otherwise} \end{cases}$$

$$(3)\ \mathtt{rdy}(\mu \mathbf{t}.G, D) = \mathtt{rdy}(G, D) \qquad (4)\ \mathtt{rdy}(\mathbf{t}, D) = \mathtt{rdy}(\mathtt{end}, D) = \emptyset$$

In (1) the send action of $\mathtt{p}$ is ready, hence the singleton including the constraints $\{\delta_{\mathtt{O}i}\}_{i \in I}$ are added to the solution and each G_i is recursively checked. Any action in G_i involving $\mathtt{p}$ or $\mathtt{q}$ is not ready. Adding $\{\mathtt{p}, \mathtt{q}\}$ to D ensures that the constraints of actions that causally depend from the first interaction are not included in the solution. (2) is similar.

Definition 2 (Satisfiability of ready actions) *We write* $\nu \models^* \mathtt{rdy}(G)$ *when the constraints of all ready actions of G are eventually satisfiable under* ν. *Formally,* $\nu \models^* \mathtt{rdy}(G)$ *iff* $\forall \{\{\delta_i\}_{i \in I}\} \in \mathtt{rdy}(G) \exists t \geq 0,\ j \in I.\ \nu + t \models \delta_j$.

$$\frac{j \in I \quad A_j = \{\delta_0, \lambda_0, \delta_I, \lambda_I\} \quad \nu \models \delta_0 \quad \nu' = [\lambda_0 \mapsto 0]\nu}{(\nu, \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{A_i\}.G_i\}_{i\in I}) \xrightarrow{\mathtt{pq}!l_j\langle S_j\rangle} (\nu', \mathtt{p} \rightsquigarrow \mathtt{q} : l_j\langle S_j\rangle\{A_j\}.G_j)} \quad \lfloor \textsc{select} \rfloor$$

$$\frac{\nu \models \delta_I \quad \nu' = [\lambda_I \mapsto 0]\nu}{(\nu, \mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S\rangle\{\delta_0, \lambda_0, \delta_I, \lambda_I\}.G) \xrightarrow{\mathtt{pq}?l\langle S\rangle} (\nu', G)} \quad \frac{(\nu, G[\mu\mathbf{t}.G/\mathbf{t}]) \xrightarrow{\ell} (\nu', G')}{(\nu, \mu\mathbf{t}.G) \xrightarrow{\ell} (\nu', G')} \quad \lfloor \textsc{branch} \rfloor / \lfloor \textsc{rec} \rfloor$$

$$\frac{\forall k \in I \quad (\nu, G_k) \xrightarrow{\ell} (\nu', G_k') \quad \mathtt{p}, \mathtt{q} \notin subj(\ell) \quad \ell \neq t}{(\nu, \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{A_i\}.G_i\}_{i\in I}) \xrightarrow{\ell} (\nu', \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{A_i\}.G_i'\}_{i\in I})} \quad \lfloor \textsc{async1} \rfloor$$

$$\frac{(\nu, G) \xrightarrow{\ell} (\nu', G') \quad \mathtt{q} \notin subj(\ell)}{(\nu, \mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S\rangle\{A\}.G) \xrightarrow{\ell} (\nu', \mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S\rangle\{A\}.G')} \quad \frac{\nu + t \models^* \mathtt{rdy}(G)}{(\nu, G) \xrightarrow{t} (\nu + t, G)} \quad \lfloor \textsc{async2} \rfloor / \lfloor \textsc{time} \rfloor$$

Fig. 1. Labelled transitions for timed global types

The transition rules for *TG*s are given in Figure 1. We assume the execution always begins with initial assignment ν_0. Rule $\lfloor$SELECT$\rfloor$ models selection as usual, except that the clock constraint of the selected branch j is checked against the current assignment (i.e., $\nu \models \delta_0$) which is updated with reset λ_0. Rules $\lfloor$ASYNC1$\rfloor$ and $\lfloor$ASYNC2$\rfloor$ model interactions that appear later (syntactically), but are not causally dependent on the first interaction. Rule $\lfloor$TIME$\rfloor$ models time passing by incrementing all clocks; the clause in the premise prevents time steps that would make the clock constraints of some ready action unsatisfiable. Note that $\lfloor$TIME$\rfloor$ can always be applied to $(\nu, \mathtt{end})$ since $\nu + t \models^* \mathtt{rdy}(\mathtt{end})$ for all t. By Definition 2, $\nu + t \models^* \mathtt{rdy}(G)$ requires the satisfiability of the constraints of *some* of the branches of (each ready action of) G, while some other branches may become unsatisfiable. In this way, the semantics of *TG*s specifies the full range of correct behaviours. For instance in $\mathtt{p} \to \mathtt{q} : \{l_1 : \{x_\mathtt{p} < c,\ \_, \mathtt{true},\ \_\},\ l_2 : \{x_\mathtt{p} > c,\ \_, \mathtt{true},\ \_\}\}$ one can, in some executions, let time pass until $x_\mathtt{p} > c$ so that l_2 can be chosen.

Semantics for timed local types. The LTS for *TL*s is defined with states (ν, T), labels $\mathcal{L}$ and by the following rules:

$$(\nu, \mathtt{q} \oplus \{l_i : \langle S_i\rangle\{B_i\}.T_i\}_{i\in I}) \xrightarrow{\mathtt{pq}!l_j\langle S_j\rangle} (\nu', T_j) \ \ (j \in I \ \ B_j = \{\delta, \lambda\} \ \ \nu \models \delta \ \ \nu' = [\lambda \mapsto 0]\nu) \ \ \lfloor \textsc{lSel} \rfloor$$

$$(\nu, \mathtt{q} \& \{l_i : \langle S_i\rangle\{B_i\}.T_i\}_{i\in I}) \xrightarrow{\mathtt{qp}?l_j\langle S_j\rangle} (\nu', T_j) \ \ (j \in I \ \ B_j = \{\delta, \lambda\} \ \ \nu \models \delta \ \ \nu' = [\lambda \mapsto 0]\nu) \ \ \lfloor \textsc{lBra} \rfloor$$

$$(\nu, T[\mu\mathbf{t}.T/\mathbf{t}]) \xrightarrow{\ell} (\nu', T') \ \ \textit{implies} \ \ (\nu, \mu\mathbf{t}.T) \xrightarrow{\ell} (\nu', T') \qquad \lfloor \textsc{lRec} \rfloor$$

$$\nu + t \models^* \mathtt{rdy}(T) \ \ \textit{implies} \ \ (\nu, T) \xrightarrow{t} (\nu', T) \qquad \lfloor \textsc{lTime} \rfloor$$

Rule $\lfloor$LSEL$\rfloor$ is for send actions and its dual $\lfloor$LBRA$\rfloor$ for receive actions. In rule $\lfloor$LTIME$\rfloor$ for time passing, the constraints of the ready action of T must be satisfiable after t in ν. Note that T always has only one ready action. The definitions of $\mathtt{rdy}(T)$ and $\nu + t \models^* \mathtt{rdy}(T)$ are the obvious extensions of the definitions we have given for *TG*s.

Given a set of participants $\{1, \ldots, n\}$ we define configurations $(T_1, \ldots, T_n, \vec{w})$ where $\vec{w} ::= \{w_{ij}\}_{i\neq j\in\{1,\ldots,n\}}$ are unidirectional, possibly empty (denoted by ε), unbounded channels with elements of the form $l\langle S\rangle$. The LTS of $(T_1, \ldots, T_n, \vec{w})$ is defined as follows, with ν being the overriding union (i.e., $\oplus_{i\in\{1,\ldots,n\}}\nu_i$) of the clock assignments ν_i

of the participants. $(\nu, (T_1, \ldots, T_n, \vec{w})) \xrightarrow{\ell} (\nu', (T_1', \ldots, T_n', \vec{w'}))$ iff:

(1) $\ell = \mathtt{pq}!l\langle S\rangle \Rightarrow (\nu_\mathtt{p}, T_\mathtt{p}) \xrightarrow{\ell} (\nu'_\mathtt{p}, T'_\mathtt{p}) \wedge w'_\mathtt{pq} = w_\mathtt{pq} \cdot l\langle S\rangle \wedge (ij \neq \mathtt{pq} \Rightarrow w_{ij} = w'_{ij} \wedge T_i = T'_i)$
(2) $\ell = \mathtt{pq}?l\langle S\rangle \Rightarrow (\nu_\mathtt{q}, T_\mathtt{q}) \xrightarrow{\ell} (\nu'_\mathtt{q}, T'_\mathtt{q}) \wedge l\langle S\rangle \cdot w'_\mathtt{pq} = w_\mathtt{pq} \wedge (ij \neq \mathtt{pq} \Rightarrow w_{ij} = w'_{ij} \wedge T_j = T'_j)$
(3) $\ell = t \Rightarrow \forall i \neq j \in \{1, \ldots, n\}.(\nu_i, T_i) \xrightarrow{\ell} (\nu_i + t, T_i) \wedge w_{ij} = w'_{ij}$

with $\mathtt{p}, \mathtt{q}, i, j \in \{1, \ldots, n\}$.

We write $TR(G)$ for the set of visible traces obtained by reducing G under the initial assignment ν_0. Similarly for $TR(T_1, \ldots, T_n, \vec{\varepsilon})$. We denote trace equivalence by $\approx$.

Theorem 3 (Soundness and completeness of projection) *Let G be a timed global type and $\{T_1, \ldots, T_n\} = \{G \downarrow_\mathtt{p}\}_{\mathtt{p} \in \mathcal{P}(G)}$ be the set of its projections, then $G \approx (T_1, \ldots, T_n, \vec{\varepsilon})$.*

4 Multiparty Session Processes with Delays

We model processes using a timed extension of the asynchronous session calculus [6]. The syntax of the session calculus with delays is presented below.

P	::=	$\overline{u}[\mathtt{n}](y).P$	Request		
		$u[\mathtt{i}](y).P$	Accept		
		$c[\mathtt{p}] \triangleleft l\langle e\rangle; P$	Select		
		$c[\mathtt{p}] \triangleright \{l_i(z_i).P_i\}_{i \in I}$	Branching		
		$\mathtt{delay}(t).P$	Delay		
		$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional		
		$P \mid Q$	Parallel		
		$\mathbf{0}$	Inaction		
		$\mu X.P$	Recursion		
		X	Variable		
		$(\nu a)P$	Hide Shared		
		$(\nu s)P$	Hide Session		
		$s : h$	Queue		
h	::=	$\emptyset \mid h \cdot (\mathtt{p}, \mathtt{q}, m)$		(queue content)	
m	::=	$l\langle v\rangle \mid (s[\mathtt{p}], \nu)$		(messages)	
c	::=	$s[\mathtt{p}] \mid y$		(session names)	
u	::=	$a \mid z$		(shared names)	
e	::=	$v \mid \neg e \mid e' \mathtt{op}\ e'$		(expressions)	
v	::=	$c \mid u \mid \mathtt{true} \mid \ldots$		(values)	

$\overline{u}[\mathtt{n}](y).P$ sends, along u, a request to start a new session y with participants $\mathtt{1}, \ldots, \mathtt{n}$, where it participates as $\mathtt{1}$ and continues as P. Its dual $u[\mathtt{i}](y).P$ engages in a new session as participant $\mathtt{i}$. Select $c[\mathtt{p}] \triangleleft l\langle e\rangle; P$ sends message $l\langle e\rangle$ to participant $\mathtt{p}$ in session c and continues as P. Branching is dual. Request and accept bind y in P, and branching binds z_i in P_i. We introduce a new primitive $\mathtt{delay}(t).P$ that executes P after waiting exactly t units of time. Note that t is a constant (as in [5, 16]). The other processes are standard. We often omit inaction $\mathbf{0}$, and the label in a singleton selection or branching, and denote with $\mathtt{fn}(P)$ the set of free variables and names of P.

We define *programs* as processes that have not yet engaged in any session, namely that have no queues, no session name hiding, and no free session names/variables.

Structural equivalence for processes is the least equivalence relation satisfying the standard rules from [6] – we recall below (first row) those for queues – plus the following rules for delays:

$(\nu s)s : \emptyset \equiv \mathbf{0} \quad s : h \cdot (\mathtt{p}, \mathtt{q}, m) \cdot (\mathtt{p}', \mathtt{q}', m') \cdot h' \equiv s : h \cdot (\mathtt{p}', \mathtt{q}', m') \cdot (\mathtt{p}, \mathtt{q}, m) \cdot h' \quad \text{if } \mathtt{p} \neq \mathtt{p}' \text{ or } \mathtt{q} \neq \mathtt{q}'$
$\mathtt{delay}(t + t').P \equiv \mathtt{delay}(t).\mathtt{delay}(t').P \qquad \mathtt{delay}(0).P \equiv P$
$\mathtt{delay}(t).(\nu a)P \equiv (\nu a)\mathtt{delay}(t).P \qquad \mathtt{delay}(t).(P \mid Q) \equiv \mathtt{delay}(t).P \mid \mathtt{delay}(t).Q$

In the first row: $(\nu s)s : \emptyset \equiv \mathbf{0}$ removes queues of ended sessions, the second rule permutes causally unrelated messages. In the second row: the first rule breaks delays into

$$\begin{array}{rcll}
\overline{a}[\mathtt{n}](y).P_1 \mid \prod_{i\in\{2,..,n\}} a[\mathtt{i}](y).P_i & \longrightarrow & (\nu s)(\prod_{i\in\{1,..,n\}} P_i[s[\mathtt{i}]/y] \mid s:\emptyset) \ (s\notin \mathtt{fn}(P_i)) & \lfloor \textsc{Link} \rfloor \\
s[\mathtt{p}][\mathtt{q}] \triangleleft l\langle e\rangle; P \mid s:h & \longrightarrow & P \mid s:h\cdot(\mathtt{p},\mathtt{q},l\langle v\rangle) \quad (e\downarrow v) & \lfloor \textsc{Sel} \rfloor \\
s[\mathtt{p}][\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in J} \mid s:(\mathtt{p},\mathtt{q},l_j\langle v\rangle)\cdot h & \longrightarrow & P_j[v/z_j] \mid s:h \quad (j\in J) & \lfloor \textsc{Bra} \rfloor \\
\mathtt{delay}(t).P \mid \prod_{j\in J} s_j:h_j & \longrightarrow & P \mid \prod_{j\in J} s_j:h_j & \lfloor \textsc{Delay} \rfloor \\
P\longrightarrow P' \ \text{(not by } \lfloor \textsc{Delay} \rfloor) & \textit{imply} & P\mid Q\longrightarrow P'\mid Q & \lfloor \textsc{Com} \rfloor \\
\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q\longrightarrow P \quad (e\downarrow \mathtt{true}) & & \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q\longrightarrow Q \quad (e\downarrow \mathtt{false}) & \lfloor \textsc{IfT/IfF} \rfloor \\
P\equiv P' \ P'\longrightarrow Q' \ Q\equiv Q' \ \textit{imply}\ P\longrightarrow Q & & P\longrightarrow P' \ \textit{imply}\ (\nu n)P\longrightarrow(\nu n)P' & \lfloor \textsc{Str/Hide} \rfloor
\end{array}$$

Fig. 2. Reduction for processes

smaller intervals, and $\mathtt{delay}(0).P \equiv P$ allows time to pass for idle processes. The rules in the third row distribute delays in hiding and parallel processes.

The reduction rules are given in Figure 2. In $\lfloor \textsc{Sel} \rfloor$ we write $e\downarrow v$ when expression e evaluates to value v. Rule $\lfloor \textsc{Delay} \rfloor$ models time passing for P. By combining $\lfloor \textsc{Delay} \rfloor$ with rule $\mathtt{delay}(t).(P \mid Q) \equiv \mathtt{delay}(t).P \mid \mathtt{delay}(t).Q$ we allow a delay to elapse simultaneously for parallel processes. The queues in parallel with P always allow time passing, unlike other kinds of processes (as shown in rule $\lfloor \textsc{Com} \rfloor$ which models the synchronous semantics of time in [14]). Rule $\lfloor \textsc{Com} \rfloor$ enables part of the system to reduce as long as the reduction does not involve $\lfloor \textsc{Delay} \rfloor$ on P. If P reduces by $\lfloor \textsc{Delay} \rfloor$ then also all other parallel processes must make the same time step, i.e. the whole system must move by $\lfloor \textsc{Delay} \rfloor$. The other rules are standard (n stands for s or a in $\lfloor \textsc{Hide} \rfloor$).

Example 4 (Temperature calculation) Process $P_{\mathtt{M}}$ is a possible implementation of participant $\mathtt{M}$ of the protocol in Example 1, e.g., $G\downarrow_{\mathtt{M}}$. Assuming that at least one task is needed in each session, we let $\mathtt{task}()$ be a local function returning the next task and $\mathtt{more\_tasks}()$ return $\mathtt{true}$ when more tasks have to be submitted and $\mathtt{false}$ otherwise.

$$\begin{array}{l}
P_{\mathtt{M}} = s[\mathtt{M}][\mathtt{W}] \triangleleft \langle \mathtt{task}() \rangle; \mu X.\mathtt{delay}(2l+w).\ s[\mathtt{M}][\mathtt{W}] \triangleright (y); \mathtt{if}\ \mathtt{more_tasks}() \\
\quad \mathtt{then}\ s[\mathtt{M}][\mathtt{A}] \triangleleft \mathsf{more}\langle y\rangle; s[\mathtt{M}][\mathtt{W}] \triangleleft \mathsf{more}\langle \mathtt{task}() \rangle; X\ \mathtt{else}\ s[\mathtt{M}][\mathtt{A}] \triangleleft \mathsf{stop}\langle y\rangle; s[\mathtt{M}][\mathtt{W}] \triangleleft \mathsf{stop}\langle\rangle; \mathtt{end}
\end{array}$$

Proof rules. We validate programs against specifications based on *TL*s, using judgements of the form $\Gamma \vdash P \triangleright \Delta$ and $\Gamma \vdash e : S$ defined on the following environments:

$$\Gamma ::= \emptyset \mid \Gamma, u:S \mid \Gamma, X:\Delta \qquad\qquad \Delta ::= \emptyset \mid \Delta, c:(\nu, T)$$

The *type environment* Γ maps shared variables/names to sorts and process variables to their types, and the *session environment* Δ holds information on the ongoing sessions, e.g., $\Delta(s[\mathtt{p}]) = (\nu, T)$ when the process being validated is acting as $\mathtt{p}$ in session s specified by T; ν is a virtual clock assignment built during the validation (virtual in the sense that it mimics the assignment associated to T by the LTS) .

Resets can generate infinite time scenarios in recursive protocols. To ensure sound typing we introduce a condition, *infinite satisfiability*, that guarantees a regularity across different instances of a recursion.

Definition 5 (Infinitely satisfiable) *G is infinitely satisfiable if either: (1) constraints in recursion bodies have no equalities nor upper bounds (i.e., $x < c$ or $x \leq c$) and no resets occur, or (2) all participants reset at each iteration.*

In the rest of this section we assume that *TG*s are infinitely satisfiable. As usual (e.g., [13]), in the validation of P we check $\Gamma \vdash P' \triangleright \Delta$ where P' is obtained by unfolding once all

$$\lfloor \textsc{vReq} \rfloor \frac{\Gamma, u : G \vdash P \triangleright \Delta, y[1] : (\nu_0, G\downarrow_1) \quad dom(\nu_0) = \{x_1\}}{\Gamma, u : G \vdash \overline{u}[\mathtt{n}](y).P \triangleright \Delta} \qquad \lfloor \textsc{vAcc} \rfloor \frac{\Gamma, u : G \vdash P \triangleright \Delta, y[\mathtt{i}] : (\nu_0, G\downarrow_\mathtt{i}) \quad dom(\nu_0) = \{x_\mathtt{i}\} \quad \mathtt{i} \neq 1}{\Gamma, u : G \vdash u[\mathtt{i}](y).P \triangleright \Delta}$$

$$\lfloor \textsc{vBra} \rfloor \frac{\forall i \in I \quad \nu \models \delta_i \qquad \begin{cases} \Gamma, z_i : S_i \vdash P_i \triangleright \Delta, c : ([\lambda_i \mapsto 0]\nu, T_i) & (S_i \neq (T_d, \delta_d)) \\ \Gamma \vdash P_i \triangleright \Delta, c : ([\lambda_i \mapsto 0]\nu, T_i), z_i : (\nu_d, T_d) \quad \nu_d \models \delta_d & (S_i = (T_d, \delta_d)) \end{cases}}{\Gamma \vdash c[\mathtt{p}] \triangleright \{l_i(z_i).P_i\}_{i \in I} \triangleright \Delta, c : (\nu, \mathtt{p}\&\{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I})}$$

$$\lfloor \textsc{vSel} \rfloor \frac{j \in I \quad \Gamma \vdash e : S_j \quad \nu \models \delta_j \quad \Gamma \vdash P \triangleright \Delta, c : ([\lambda_j \mapsto 0]\nu, T_j) \quad (S_j \neq (T_d, \delta_d))}{\Gamma \vdash c[\mathtt{p}] \triangleleft l_j\langle e \rangle; P \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I})}$$

$$\lfloor \textsc{vDel} \rfloor \frac{j \in I \quad \Gamma \vdash e : S_j \quad \nu \models \delta_j \quad \nu_d \models \delta_d \quad \Gamma \vdash P \triangleright \Delta, c : ([\lambda_j \mapsto 0]\nu, T_j) \quad (S_j = (T_d, \delta_d))}{\Gamma \vdash c[\mathtt{p}] \triangleleft l_j\langle e \rangle; P \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I}), c' : (\nu_d, T_d)}$$

$$\lfloor \textsc{vPar} \rfloor \frac{dom(\Delta_1) \cap dom(\Delta_2) = \emptyset \quad \Gamma \vdash P_i \triangleright \Delta_i \quad i \in \{1,2\}}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1, \Delta_2} \qquad \lfloor \textsc{vCond} \rfloor \frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash P_i \triangleright \Delta \quad i \in \{1,2\}}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \triangleright \Delta}$$

$$\lfloor \textsc{vTime} \rfloor \frac{\Gamma \vdash P \triangleright \{c_i : (\nu_i + t, T_i)\}_{i \in I}}{\Gamma \vdash \mathtt{delay}(t).P \triangleright \{c_i : (\nu_i, T_i)\}_{i \in I}} \qquad \lfloor \textsc{vEnd} \rfloor \frac{\forall c \in dom(\Delta) \quad \Delta(c) = (\nu, \mathtt{end})}{\Gamma \vdash \mathbf{0} \triangleright \Delta}$$

$$\lfloor \textsc{vDef} \rfloor \frac{\Gamma, X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \mu X.P \triangleright \Delta} \quad \lfloor \textsc{vCall} \rfloor \frac{\forall c \in dom(\Delta') \quad \Delta'(c) = (\nu, \mathtt{end})}{\Gamma, X : \Delta \vdash X \triangleright \Delta, \Delta'}$$

Fig. 3. Proof rules for programs

recursions $\mu X.P''$ occurring in P. This ensures that both the first instance of a recursion and the successive ones (all similar by infinite satisfiability) satisfy the specification.

We show in Figure 3 selected proof rules for programs. Rule $\lfloor \textsc{vReq} \rfloor$ for session request adds a new instance of session for participant `1` to Δ in the premise. The newly instantiated session is associated with an initial assignment ν_0 for the clock of participant `1`. Rule $\lfloor \textsc{vAcc} \rfloor$ for session accept is similar but initiates a new session for participant `i` with $i > 1$. Rule $\lfloor \textsc{vBra} \rfloor$ is for branching processes. For all $i \in I$, σ_i must hold under ν and the virtual clock assignments used to validate P_i is reset according to λ_i. If the received message is a session (i.e., $S_i = (T_d, \delta_d)$) a new assignment $z_i : (\nu_d, T_d)$ is added to Δ in the premise. This can be any assignment such that $\nu_d \models \delta_d$. Rule $\lfloor \textsc{vSel} \rfloor$ for selection processes checks the constraint δ_j of the selected branch j against ν. In the premise, ν is reset as prescribed by λ_j. Rule $\lfloor \textsc{vDel} \rfloor$ for delegation requires δ_d to be satisfied under ν_d (of the delegated session) which is removed from the premise. Rule $\lfloor \textsc{vTime} \rfloor$ increments the assignments of all sessions in Δ. Rule $\lfloor \textsc{vEnd} \rfloor$ validates $\mathbf{0}$ if there are no more actions prescribed by Δ. Rule $\lfloor \textsc{vDef} \rfloor$ extends Γ with the assignment for process variable X. Rules $\lfloor \textsc{vPar} \rfloor$ and $\lfloor \textsc{vCond} \rfloor$ are standard. Rule $\lfloor \textsc{vCall} \rfloor$ validates, as usual, recursive call X against $\Gamma(X)$ (and possibly some terminated sessions Δ').

Theorem 6 (Type preservation) *If $\Gamma \vdash P \triangleright \emptyset$ and $P \longrightarrow P'$, then $\Gamma \vdash P' \triangleright \emptyset$.*

In the above theorem, P is a process reduced from a program (hence Δ is $\emptyset$). A standard corollary of type preservation is error freedom. An error state is reached when a process performs an action at a time that violates the constraints prescribed by its type. To formulate this property, we extend the syntax of processes as follows: selection and branching are annotated with clock constraints and resets (i.e., $c[\mathtt{p}] \triangleleft l\langle e \rangle \{\delta, \lambda\}; P$ and $c[\mathtt{p}] \triangleright \{l_i(z_i)\{\delta_i, \lambda_i\}.P_i\}_{i \in I}$); two new processes, `error` and clock process $(s[\mathtt{p}], \nu)$, are introduced. Process `error` denotes a state in which a violation has occurred, and

$(s[\mathtt{p}], \nu)$ associates a clock assignment ν to ongoing session $s[\mathtt{p}]$. The reduction rules for processes are extended as shown below.

$$\frac{\forall i \in \{1,..,n\} \quad s \notin \mathtt{fn}(P_i)}{\overline{a}[\mathtt{n}](y).P_1 \mid \prod_{i\in\{2,..,n\}} a[\mathtt{i}](y).P_i \longrightarrow (\nu s)(\prod_{i\in\{1,..,n\}} (P_i[s[\mathtt{i}]/y] \mid (s[\mathtt{i}], \nu_0)) \mid s : \emptyset)} \lfloor \textsc{Link} \rfloor$$

$$\mathtt{delay}(t).P \mid \prod_{j\in J}(s_j : h_j \mid \prod_{k\in K_j}(s_j[\mathtt{p}_k], \nu_k)) \longrightarrow P \mid \prod_{j\in J}(s_j : h_j \mid \prod_{k\in K_j}(s_j[\mathtt{p}_k], \nu_k + t)) \lfloor \textsc{Delay} \rfloor$$

$$\frac{e \downarrow v \qquad \nu' = [\lambda \mapsto 0]\nu \qquad \delta \models \nu}{s[\mathtt{p}][\mathtt{q}] \triangleleft \{\delta, \lambda\} l\langle e\rangle; P \mid s : h \mid (s[\mathtt{p}], \nu) \longrightarrow P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v\rangle) \mid (s[\mathtt{p}], \nu')} \lfloor \textsc{Sel} \rfloor$$

$$\frac{\neg\delta \models \nu}{s[\mathtt{p}][\mathtt{q}] \triangleleft \{\delta, \lambda\} l\langle e\rangle; P \mid s : h \mid (s[\mathtt{p}], \nu) \longrightarrow \mathtt{error} \mid s : h \mid (s[\mathtt{p}], \nu)} \lfloor \textsc{ESel} \rfloor$$

$\lfloor \textsc{Link} \rfloor$ introduces a clock process $(s[\mathtt{i}], \nu_0)$ with initial assignment for each participant $\mathtt{i}$ in the new session; $\lfloor \textsc{Delay} \rfloor$ increments all clock assignments, $\lfloor \textsc{Sel} \rfloor$ checks the clock constraints against clock assignments and appropriately resets (the rule for branching is extended similarly); $\lfloor \textsc{ESel} \rfloor$ is an additional rule which moves to `error` when a process tries to perform a send action at a time that does not satisfy the constraint (a similar rule is added for violating receive actions). Note that $\lfloor \textsc{Sel} \rfloor$ only resets the clocks associated to participant $\mathtt{p}$ in session s and never affects clocks of other participants and sessions. The proof rules are adapted straightforwardly, with `error` not validated against any Δ.

Theorem 7 (Time-error freedom) *If* $\Gamma \vdash P \triangleright \Delta$, *and* $P \to^* P'$ *then* $P' \not\equiv \mathtt{error}$.

5 Time-progress of timed processes and CTAs

This section studies a subclass of timed global types characterised by two properties, *feasibility* and *wait-freedom* and states their decidability; it then shows that these are sufficient conditions for progress of validated processes and CTAs.

Feasibility. A *TG* G is *feasible* iff $(\nu_0, G_0) \longrightarrow^* (\nu, G)$ implies $(\nu, G) \longrightarrow^* (\nu', \mathtt{end})$ for some ν'. Intuitively, G_0 is feasible if every partial execution can be extended to a terminated session. Not all *TGs* are feasible. The specified protocol may get stuck because a constraint is unsatisfiable, for example it is `false`, or the restrictions posed by previously occurred constraints are too strong. We give below a few examples of non-feasible (1,5) and feasible (2,3,4,6) global types:

1. $\mathtt{p} \to \mathtt{q} : \langle \mathtt{int}\rangle\{x_\mathtt{p} > 3, \_, x_\mathtt{q} = 4, \_\}$
2. $\mathtt{p} \to \mathtt{q} : \langle \mathtt{int}\rangle\{x_\mathtt{p} > 3 \wedge x_\mathtt{p} \leq 4, \_, x_\mathtt{q} = 4, \_\}$ 3. $\mathtt{p} \to \mathtt{q} : \langle \mathtt{int}\rangle\{x_\mathtt{p} > 3, \_, x_\mathtt{q} \geq 4, \_\}$
4. $\mathtt{q} \to \mathtt{r} : \{l_1 : \{x_\mathtt{q} > 3, \_, \mathtt{true}, \_\}, \; l_2 : \{x_\mathtt{q} < 3, \_, \mathtt{true}, \_\}\}$
5. $\mu\mathbf{t}.\mathtt{p} \to \mathtt{q} : \langle \mathtt{int}\rangle\{x_\mathtt{p} < 1, x_\mathtt{p}, x_\mathtt{q} = 2, x_\mathtt{q}\}.\mathtt{p} \to \mathtt{r} : \langle \mathtt{int}\rangle\{x_\mathtt{p} < 5, \_, \mathtt{true}, x_\mathtt{r}\}.\mathbf{t}$
6. $\mu\mathbf{t}.\mathtt{p} \to \mathtt{q} : \langle \mathtt{int}\rangle\{x_\mathtt{p} < 1, x_\mathtt{p}, x_\mathtt{q} = 2, x_\mathtt{q}\}.\mathtt{p} \to \mathtt{r} : \langle \mathtt{int}\rangle\{x_\mathtt{p} < 1, \_, \mathtt{true}, x_\mathtt{r}\}.\mathbf{t}$

In (1) if $\mathtt{p}$ sends $\langle \mathtt{int}\rangle$ at time 5, which satisfies $x_\mathtt{p} > 3$, then there exists no $x_\mathtt{q}$ satisfying $x_\mathtt{q} = 4$ (considering that $x_\mathtt{q}$ must be greater than or equal to 5 to respect the global flowing of time); (2) amends (1) by restricting the earlier constraint; (3) amends (1) by relaxing the unsatisfiable constraint. In branching and selection at least one constraint associated to the branches must be satisfiable, e.g., we accept (4). In recursive *TGs*, a constraint may become unsatisfiable by constraints that occur after, syntactically, in the same recursion body. In the second iteration of (5) $x_\mathtt{q} = 2$ is made unsatisfiable by $x_\mathtt{p} < 5$ occurring in the first iteration (e.g., $\mathtt{p}$ may send $\mathtt{q}$ the message when $x_\mathtt{q} > 2$); in (6) this problem is solved by restricting the second constraint on $x_\mathtt{p}$.

Wait-freedom. In distributed implementations, a party can send a message at any time satisfying the constraint. Another party can choose to execute the corresponding receive action at any specific time satisfying the constraint without knowing when the message has been or will be sent. If the constraints in a *TG* allow a receive action before the corresponding send, a complete correct execution of the protocol may not be possible at run-time (as we will illustrate later with an example). We introduce a condition on *TG*s called wait-freedom, ensuring that in all the distributed implementations of a *TG*, a receiver checking the queue at any prescribed time never has to wait for a message. Formally (and using $\supset$ for logic implication): G_0 is *wait-free* iff $(\nu_0, G_0) \longrightarrow^* \xrightarrow{\mathtt{pq}!l\langle S\rangle} (\nu, G)$ and $\mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S\rangle\{\delta_0, \lambda_0, \delta_\mathtt{I}, \lambda_\mathtt{I}\}.G' \in G$ imply $\delta_\mathtt{I} \supset \nu(x) \leq x$ for all $x \in fn(\delta_\mathtt{I})$.

We show below a process $P \mid Q$ whose correct execution cannot complete despite $P \mid Q$ is the well-typed implementation of a feasible (but not wait-free) *TG*.

$$
\begin{array}{ll}
G & = \mathtt{p} \to \mathtt{q} : \langle \mathtt{int}\rangle\{x_\mathtt{p} < 3 \vee x_\mathtt{p} > 3,\ \_ , x_\mathtt{q} < 3 \vee x_\mathtt{q} > 3,\ \_\}. \\
 & \quad \mathtt{q} \to \mathtt{p} : \{l_1 : \{x_\mathtt{q} > 3,\ \_ , x_\mathtt{p} > 3,\ \_\},\ l_2 : \{x_\mathtt{q} < 3,\ \_ , x_\mathtt{p} < 3,\ \_\}\} \\
G\downarrow_\mathtt{p} & = \mathtt{q} \oplus \langle \mathtt{int}\rangle\{x_\mathtt{p} < 3 \vee x_\mathtt{p} > 3,\ \_\}.\mathtt{q}\&\{l_1 : \{x_\mathtt{p} > 3,\ \_\},\ l_2 : \{x_\mathtt{p} < 3,\ \_\}\} \\
G\downarrow_\mathtt{q} & = \mathtt{p}\&\langle \mathtt{int}\rangle\{x_\mathtt{q} < 3 \vee x_\mathtt{q} > 3,\ \_\}.\mathtt{p} \oplus \{l_1 : \{x_\mathtt{q} > 3,\ \_\},\ l_2 : \{x_\mathtt{q} < 3,\ \_\}\} \\
P & = \mathtt{delay}(6).s[\mathtt{p}][\mathtt{q}] \triangleleft \langle 10\rangle; s[\mathtt{q}][\mathtt{p}] \triangleright \{l_1.0,\ l_2.0\} \quad Q = s[\mathtt{p}][\mathtt{q}] \triangleright (x).s[\mathtt{q}][\mathtt{p}] \triangleleft l_2\langle\rangle; 0
\end{array}
$$

P implements $G\downarrow_\mathtt{p}$: it waits 6 units of time, then sends $\mathtt{q}$ a message and waits for the reply. Q implements $G\downarrow_\mathtt{q}$: it receives a message from $\mathtt{p}$ and then selects label l_2; both interactions occur at time 0 which satisfies the clock constraints of $G\downarrow_\mathtt{q}$. By Theorem 7, since $\emptyset \vdash P \mid Q \triangleright s[\mathtt{p}] : (\nu_0, G\downarrow_\mathtt{p}), s[\mathtt{q}] : (\nu_0, G\downarrow_\mathtt{q})$, no violating interactions will occur in $P \mid Q$. However $P \mid Q$ cannot make any step and the session it stuck. This scenario, unlike errors in § 4 representing violations, models the fact that a non wait-free specification allows participants to have incompatible views of the timings of action.

Decidability. If G is infinitely satisfiable (as also assumed by the typing in § 4), then there exists a terminating algorithm for checking that it is feasible and wait-free. The algorithm is based on a direct acyclic graph annotated with clock constraints and resets, and whose edges model the causal dependencies between actions in (the one-time unfolding of) G. The algorithm yields Proposition 8.

Proposition 8 (Decidability) *Feasibility and wait-freedom of infinitely satisfiable TGs are decidable.*

Time-progress for processes. We study the conditions under which a validated program P is guaranteed to proceed until the completion of all activities of the protocols it implements, assuming progress of its untimed counterpart (i.e., $\mathtt{erase}(P)$). The *erasure* $\mathtt{erase}(P)$ of a timed processes P is defined inductively by removing the delays in P (i.e., $\mathtt{erase}(\mathtt{delay}(t).P') = \mathtt{erase}(P')$), while leaving unchanged the untimed parts (e.g., $\mathtt{erase}(\overline{u}[\mathtt{n}](y).P') = \overline{u}[\mathtt{n}](y).\mathtt{erase}(P')$); the other rules are homomorphic.

Proposition 9 (Conformance) *If $P \longrightarrow P'$, then $\mathtt{erase}(P) \longrightarrow^* \mathtt{erase}(P')$.*

Processes implementing multiple sessions may get stuck because of a bad timing of their attempts to initiate new sessions. Consider $P = \mathtt{delay}(5).\overline{a}[2](v).P_1 \mid a[2](y).P_2$;

$\texttt{erase}(P)$ can immediately start the session, whereas P is stuck. Namely, the delay of 5 time units introduces a deadlock in a process that would otherwise progress. This scenario is ruled out by requiring processes to only initiate sessions before any delay occurs, namely we assume processes to be *session delay*. All examples we have examined in practice (e.g., OOI use cases [17]) conform session delay.

Definition 10 (Session delay) *P is session delay if for each process occurring in P of the form $\texttt{delay}(t).P'$ (with $t > 0$), there are no session request and session accept in P'.*

We show that feasibility and wait-freedom, by regulating the exchange of messages *within* established sessions, are sufficient conditions for progress of session delay processes. We say that P is a *deadlock process* if $P \longrightarrow^* P'$ where $P' \not\longrightarrow$ and $P' \not\equiv \mathbf{0}$, and that Γ is feasible (resp. wait-free) if $\Gamma(u)$ is feasible (resp. wait-free) for all $u \in dom(\Gamma)$.

Theorem 11 (Timed progress in interleaved sessions) *Let Γ be a* feasible *and* wait-free *mapping, $\Gamma \vdash P_0 \triangleright \emptyset$, and $P_0 \longrightarrow^+ P$. If P_0 is session delay, $\texttt{erase}(P)$ is not a deadlock process and if $\texttt{erase}(P) \longrightarrow$ then $P \longrightarrow$.*

Several typing systems guarantee deadlock-freedom, e.g. [6]. We use one instance from [13] where a single session ensures deadlock-freedom. We characterise processes implementing single sessions, or *simple*, as follows: P is simple if $P_0 \longrightarrow^* P$ for some program P_0 such that $a : G \vdash P_0 \triangleright \emptyset$, and $P_0 = \overline{a}[\mathtt{n}](y).P_1 \mid \prod_{i \in \{2,\ldots,n\}} a[\mathtt{i}](y).P_i$ where $P_1, \ldots, P_n$ do not contain any name hiding, request/accept, and session receive/delegate.

Corollary 12 (Time progress in single sessions) *Let G be* feasible *and* wait-free*, and P be a* simple process *with $a : G \vdash P \triangleright \emptyset$. If $\texttt{erase}(P) \longrightarrow$, then there exist P' and P'' such that $\texttt{erase}(P) \longrightarrow P'$, $P \longrightarrow^+ P''$ and $\texttt{erase}(P'') = P'$.*

Progress for CTAs. Our *TG*s (§ 3) are a natural extension of global types with timed notions from CTAs. This paragraph clarifies the relationships between *TG*s and CTAs. We describe the exact subset of CTAs that corresponds to *TG*s. We also give the conditions for progress and liveness that characterise a new class of CTAs.

We first recall some definitions from [3, 14]. A *timed automaton* is a tuple $\mathcal{A} = (Q, q_0, \mathcal{A}ct, X, E, F)$ such that Q are the states, $q_0 \in Q$ is the initial state, $\mathcal{A}ct$ is the alphabet, X are the clocks, and $E \subseteq (Q \times Q \times \mathcal{A}ct \times 2^X \times \Phi(X))$ are the transitions, where 2^X are the resets, $\Phi(X)$ the clock constraints, and F the final states. A *network of CTAs* is a tuple $\mathcal{C} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, \vec{w})$ where $\vec{w} = \{w_{ij}\}_{i \neq j \in \{1,\ldots,n\}}$ are unidirectional unbounded channels. The LTS for CTAs is defined on states $s = ((q_1, \nu_1), \ldots, (q_n, \nu_n), \vec{w})$ and labels $\mathcal{L}$ and is similar to the semantics of configurations except that each $\mathcal{A}_i$ can make a time step even if it violates a constraint. For instance, assume that $\mathcal{A}_1$ can only perform transition $(q_1, q_1', ij!l\langle S\rangle, \emptyset, x_i \leq 10)$ from a non-final state q_1, and that $\nu_1 = 10$, then the semantics in [14] would allow a time transition with label 10. However, after such transition $\mathcal{A}_1$ would be stuck in a non-final state and the corresponding trace would not be accepted by the semantics of [14].

In order to establish a natural correspondence between *TG*s and CTAs we introduce an additional condition on the semantics of $\mathcal{C}$, similar to the constraint on ready actions

in the LTS for TG (rule $\lfloor$TIME$\rfloor$ in § 3). We say that a time transition with label t is *specified* if $\forall i \in \{1,..,n\}$, $\nu_i + t \models^* \mathtt{rdy}(q_i)$ where $\mathtt{rdy}(q_i)$ is the set $\{\delta_j\}_{j\in J}$ of constraints of the outgoing actions from q_i. We say that a semantics is specified if it only allows specified time transitions. With a specified semantics, $\mathcal{A}_1$ from the example above could not make any time transition before action $ij!l\langle S\rangle$ occurs.

The correspondence between *TG*s and CTAs is given as a sound and complete encoding. The *encoding* from T into $\mathcal{A}$, denoted by $\mathcal{A}(T)$, follows exactly the definition in [11, § 2], but adds clock constraints and resets to the corresponding edges, and sets the final states to $\{\mathtt{end}\}$. The encoding of a set of *TLs* $\{T_i\}_{i\in I}$ into a network of CTAs, written $\mathcal{A}(\{T_i\}_{i\in I})$, is the tuple $(\mathcal{A}(T_1),\ldots,\mathcal{A}(T_n),\vec{\varepsilon})$. Let G have projections $\{T_i\}_{i\in I}$, we write $\mathcal{A}(G)$ for as $(\mathcal{A}(T_1),\ldots,\mathcal{A}(T_n),\vec{\varepsilon})$.

Before stating soundness and completeness, we recall and adapt (to the timed setting), two conditions from [11]: the basic property (timed automata have the same shape as *TL*s) and multiparty compatibility (timed automata perform the same actions as a set of projected *TG*). A state s is *stable* when all its channels are empty. More precisely: $\mathcal{C}$ is *basic* when all its timed automata are deterministic, and the outgoing actions from each (q_i, C_i) are all sending or all receiving actions, and all to/from the same co-party. $\mathcal{C}$ is *multiparty compatible* when in all its reachable stable states, all possible (input/output) action of each timed automaton can be matched with a corresponding complementary (output/input) actions of the rest of the system after some 1-bounded executions (i.e., executions where the size of each buffer contains at most 1 message).[1]

A *session CTA* is a basic and multiparty compatible CTA with specified semantics.

Theorem 13 (Soundness and completeness) **(1)** *Let G be a (projectable) TG then $\mathcal{A}(G)$ is basic and multiparty compatible. Furthermore with a specified semantics $G \approx \mathcal{A}(G)$.* **(2)** *If $\mathcal{C}$ is a session CTA then there exists G such that $\mathcal{C} \approx \mathcal{A}(G)$.*

Our characterisation does not directly yields transparency of properties, differently from the untimed setting [11] and similarly to timed processes (§ 4). In fact, a session CTA itself does not satisfy progress. In the following we give the conditions that guarantee progress and liveness of CTAs. Let $s = ((q_1,\nu_1),\ldots,(q_n,\nu_n),\vec{w})$ be a reachable state of $\mathcal{C}$: s is a *deadlock state* if (i) $\vec{w} = \vec{\varepsilon}$, (ii) for all $i \in \{1,\ldots,n\}$, (q_i,ν_i) does not have outgoing send actions, and (iii) for some $i \in \{1,\ldots,n\}$, (q_i,ν_i) has incoming receiving action; s satisfies *progress* if for all s' reachable from s: (1) s' is not a deadlock state, and (2) $\forall t \in \mathbb{N}$, $((q_1,\nu_1+t),\ldots,(q_n,\nu_n+t),\vec{w})$ is reachable from s in $\mathcal{C}$. We say $\mathcal{C}$ satisfies *liveness* if for every reachable state s in $\mathcal{C}$, $s \longrightarrow^* s'$ with s' final.

Progress entails deadlock freedom (1) and, in addition, requires (2) that it is always possible to let time to diverge; namely the only possible way forward cannot be by actions occurring at increasingly short intervals of time (i.e., Zeno runs).[2]

We write $TR(\mathcal{C})$ for the set of visible traces that can be obtained by reducing $\mathcal{C}$. We extend to CTAs the trace equivalence $\approx$ defined in § 3.

Theorem 14 (Progress and liveness for CTAs) *If $\mathcal{C}$ is a session CTA and there exists a* feasible *G s.t. $\mathcal{C} \approx \mathcal{A}(G)$, then $\mathcal{C}$ satisfies progress and liveness.*

[1] Note that multiparty compatibility allows scenarios with unbounded channels e.g., the channel from $\mathtt{p}$ to $\mathtt{q}$ in $\mu\mathbf{t}.\mathtt{p} \to \mathtt{q} : l\langle S\rangle\{A\}.\mathbf{t}$

[2] The time divergence condition is common in timed setting and is called time-progress in [3].

6 Conclusion and Related Work

We design choreographic timed specifications based on the semantics of CTAs and MPSTs, and attest our theory in the π-calculus. The table below recalls the results for the untimed setting we build upon (first row), and summarises our results: a decidable proof system for π-calculus processes ensuring time-error freedom and a sound and complete characterisation of CTAs (second row), and two decidable conditions ensuring progress of processes (third row). These conditions also characterise a new class of CTAs, without restrictions on the topologies, that satisfy progress and liveness. We verified the practicability of our approach in an implementation of a timed conversation API for Python. The prototype [1] is being integrated into the OOI infrastructure [17].

*TG*s	π-calculus	session CTAs
untimed	type safety, error-freedom, progress [13]	Sound, complete characterisation, progress [11]
timed	type safety (Theorem 6) error-freedom (Theorem 7)	Sound, complete characterisation, (Theorem 13)
feasible, wait-free	progress (Theorems 11 and 12)	progress (Theorem 14)

Literature on MPSTs. The extension of the semantics of types with time is delicate as it may introduce unwanted executions (as discussed in § 3). To capture only the correct executions (corresponding to accepted traces in timed automata) we have introduced a new condition on time reductions of *TG*s and *TL*s: *satisfiability of ready actions* (e.g., $\lfloor \text{TIME} \rfloor$ in Figure 1). Our main challenge was extending the progress properties of untimed types [6] and CAs [11] to timed interactions. We introduced two additional necessary conditions for the timed setting, feasibility and wait-freedom, whose decidability is non trivial, and their application to time-progress is new. The theory of assertion-enhanced MPSTs [7] (which do not include progress) could not be applied to the timed scenario due to resets and the need to ensure consistency w.r.t. absolute time flowing.

Reachability and verification. In our work, if a CTA derives from a feasible *TG* then error and deadlock states will not be reached. Decidability of reachability for CTAs has been proven for specific topologies: those of the form $(\mathcal{A}_1, \mathcal{A}_2, w_{1,2})$ [14] and polyforests [10]. A related approach [2] extends MSCs with timed events and provides verification method that is decidable when the topology is a single strongly connected component, which ensures that channels have an upper bound. Our results do not depend on the topology nor require a limitation of the buffer size (e.g., the example in § **??** is not a polyforest and the buffer of `A` is unlimited). On the other hand, our approach relies on the additional restrictions induced by the conversation structure of *TG*s.

Feasibility. Feasibility was introduced in a different context (i.e., defining a not too stringent notion of fairness) in [4]. This paper gives a concrete definition in the context of real-time interactions, and states its decidability for infinitely satisfiable *TG*s. [21] gives an algorithm to check deadlock freedom for timed automata. The algorithm, based on syntactic conditions on the states relying on invariant annotations, is not directly applicable to check feasibility e.g., on the timed automaton derived from a *TG*.

Calculi with time. Recent work proposes calculi with time, for example: [18] includes time constraints inspired by timed automata into the π-calculus, [5, 16] add timeouts, [12] analyses the active times of processes, and [15] for service-oriented systems. The aim of our work is different from the work above: we use timed specifications *as types* to check time properties of the interactions, rather than enriching the π-calculus syntax with time primitives and reason on examples using timed LTS (or check channels linearity as [5]). Our aim is to define a static checker for time-error freedom and progress on the basis of a semantics guided by timed automata. With this respect, our calculus is a small syntactic extension from the π-calculus and is simpler than the above calculi.

References

1. Timed conversation API for Python. `www.doc.ic.ac.uk/~lbocchi/TimeApp.html`.
2. S. Akshay, P. Gastin, M. Mukund, and K. N. Kumar. Model checking time-constrained scenario-based specifications. In *FSTTCS*, volume 8 of *LIPIcs*, pages 204–215, 2010.
3. R. Alur and D. L. Dill. A theory of timed automata. *TCS*, 126:183–235, 1994.
4. K. R. Apt, N. Francez, and S. Katz. Appraising fairness in distributed languages. In *POPL*, pages 189–198. ACM, 1987.
5. M. Berger and N. Yoshida. Timed, distributed, probabilistic, typed processes. In *APLAS*, volume 4807 of *LNCS*, pages 158–174. 2007.
6. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.
7. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *LNCS*, pages 162–176, 2010.
8. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30:323–342, 1983.
9. G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multi-party session. *Logical Methods in Computer Science*, 8(1), 2012.
10. L. Clemente, F. Herbreteau, A. Stainer, and G. Sutre. Reachability of communicating timed processes. In *FOSSACS*, volume 7794 of *LNCS*, pages 81–96. Springer, 2013.
11. P.-M. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186, 2013.
12. M. Fischer et al. A new time extension to π-calculus based on time consuming transition semantics. In *Languages for System Specification*, pages 271–283. 2004.
13. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008.
14. P. Krcal and W. Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In *CAV*, volume 4144 of *LNCS*, pages 243–257, 2006.
15. A. Lapadula, R. Pugliese, and F. Tiezzi. Cows: A timed service-oriented calculus. In *ICTAC*, volume 4711 of *LNCS*, pages 275–290, 2007.
16. H. A. López and J. A. Pérez. Time and exceptional behavior in multiparty structured interactions. In *WS-FM*, volume 7176 of *LNCS*, pages 48–63. 2012.
17. Ocean Observatories Initiative (OOI). `http://oceanobservatories.org/`.
18. N. Saeedloei and G. Gupta. Timed π-calculus. In *TGC*, LNCS, 2013. to appear.
19. Savara JBoss Project. `http://www.jboss.org/savara`.
20. Scribble Project homepage. `www.scribble.org`.
21. S. Tripakis. Verifying progress in timed systems. In *Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *LNCS*, pages 299–314. 1999.
22. Technical report, department of computing, imperial college london, May 2014. 2014/3.