Timed Multiparty Session Types ⋆

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida

Imperial College London

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 with time constraints ensures type safety and time-error freedom, namely processes respect the prescribed timing and causalities between interactions. A decidable condition, enforceable on timed global types, guarantees global 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 global 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 [20] and to the governance of large cyberinfrastructures [18] via the Scribble (a MPST-based tool chain) project [21].

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

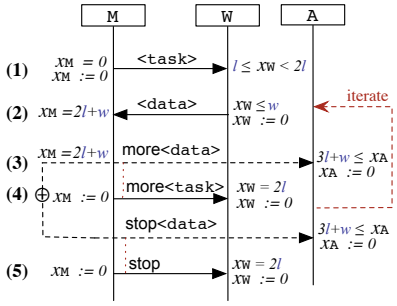
⋆ We would like to thank Viviana Bono and Mariangiola Dezani-Ciancaglini, and Massimo Bartoletti for the discussions and insightful comments on a previous version of this paper, which allowed us to improve our work.

additional time analysis of the types. In § 3 we give the semantics of timed global types (*TG*s) and prove soundness and completeness of the projection onto timed local types (*TL*s) (Theorem 3.3). 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 for the π-calculus enables a modular verification for time-error freedom (Theorem 4.4): 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) from happening: *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 5.1), 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 5.4). Transparency of properties is guaranteed not only between *TG*s and processes, but also between *TG*s and CTAs: we give a sound and complete characterisation (Theorem 5.6) yielding a new class of CTAs which enjoys progress and liveness (Theorem 5.7). Conclusions and related work are in § 6. For the reviewers convenience we attach an appendix with full proofs, detailed related work, and the definitions omitted in the paper.

2 Running example: a use-case of a distributed timed protocol

The motivating scenario developed with our partner, the Ocean Observatories Initiative (OOI) [18], 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 represented by combining global protocol descriptions from MPSTs and time from CTAs. We show, using a Message Sequence Chart (MSC)-like notation, 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. Delays l (average latency of the network) and w (sampling time) are expressed in milliseconds. Each clock can be reset many times.



(1): `M` sends `W` a message of type `task` and resets x_M ($x_M := 0$). 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 $x_W \le w$ and resets x_W. `M` receives the message at time $x_M = 2l + w$.

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

(4): If more was selected in **(3)** then `M` sends `W` a new task with label more, resets x_M and another iteration is executed.

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

We build on the synchronous semantics in [14] i.e., time elapses at the same pace for all the parts of the system. However, note that clocks can be reset at different times, hence e.g. the values of x_M, x_W, and x_A differ at some point.

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:

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

where c is a bound time constant taking values in $\mathbb{Q}^{\geq 0}$ (we derive `false`, $<$, $\leq$, $\geq$, $\vee$ in the standard way). The set of free clocks in δ, written $fn(\delta)$, is defined inductively as: $fn(\texttt{true}) = \emptyset$, $fn(x > c) = fn(x = c) = \{x\}$, $fn(\neg\delta) = fn(\delta)$, 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 predicate λ over X is a subset of X. When λ is $\emptyset$ then no reset occurs, 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 predicate. The clock constraint specifies when that action can be executed and the reset predicate specifies which clocks must be reset.

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

$$\begin{array}{lll} S & ::= & \texttt{bool} \mid \texttt{nat} \mid \ldots \mid G \mid (T, \delta) \\ G & ::= & \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{A_i\}.G_i\}_{i\in I} \mid \mu\mathbf{t}.G \mid \mathbf{t} \mid \texttt{end} \qquad A ::= \{\delta_0, \lambda_0, \delta_1, \lambda_1\} \\ T & ::= & \mathtt{p} \oplus \{l_i : \langle S_i\rangle\{B_i\}.T_i\}_{i\in I} \mid \mathtt{p}\&\{l_i : \langle S_i\rangle\{B_i\}.T_i\}_{i\in I} \mid \mu\mathbf{t}.T \mid \mathbf{t} \mid \texttt{end} \qquad B ::= \{\delta, \lambda\} \end{array}$$

The sorts S include base types (`bool`, `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* $A_i = \{\delta_{\mathtt{0}}, \lambda_{\mathtt{0}}, \delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\}$, where $\delta_{\mathtt{0}}$ and $\lambda_{\mathtt{0}}$ are the clock constraint and reset predicate for the output action, and $\delta_{\mathtt{I}}$ and $\lambda_{\mathtt{I}}$ 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 `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 predicate of an action performed by a participant are defined only over the clocks owned by that participant. The example below violates this assumption.

$$G_{\mathtt{1}} = \mathtt{p} \to \mathtt{q} : \langle \mathtt{int}\rangle\{x_{\mathtt{p}} < 10, x_{\mathtt{p}}, x_{\mathtt{p}} < 20, x_{\mathtt{p}}\}$$

since both the constraints of the (send) action of $\mathtt{p}$ and of the (receive) action of $\mathtt{q}$ are defined over $x_{\mathtt{p}}$, and $x_{\mathtt{p}}$ can be owned by either $\mathtt{p}$ or $\mathtt{q}$ (similarly for the reset predicates $\{x_{\mathtt{p}}\}$). 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 $\mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{\delta_{\mathtt{0}i}, \lambda_{\mathtt{0}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}.G_i\}_{i\in I} \in G$ implies $fn(\delta_{\mathtt{0}i}), \lambda_{\mathtt{0}i} \subseteq X(\mathtt{p},G)$ 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 onto $\mathtt{p} \in \mathcal{P}(G)$ by $G\downarrow_{\mathtt{p}}$; the definition is standard except each $\{\delta_{\mathtt{0}}, \lambda_{\mathtt{0}}, \delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\}$ is projected onto the sender (resp. receiver) by keeping only the output part $\{\delta_{\mathtt{0}}, \lambda_{\mathtt{0}}\}$ (resp. the input part $\{\delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\}$), 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 3.1 (Temperature calculation) We show below the global timed type G for the protocol in § 2 and its projection $G\downarrow_{\mathtt{M}}$ onto $\mathtt{M}$. We write $\_$ for empty reset predicates.

$$\begin{array}{l}
G \;\; = \mathtt{M} \to \mathtt{W} : \langle \mathtt{task}\rangle\{B^1_{\mathtt{0}}, B^1_{\mathtt{I}}\}.\mu\mathbf{t}.G' \\
G' \;\; = \mathtt{W} \to \mathtt{M} : \langle \mathtt{data}\rangle\{B^2_{\mathtt{0}}, B^2_{\mathtt{I}}\}. \\
\qquad \mathtt{M} \to \mathtt{A} : \{\mathsf{more}\langle \mathtt{data}\rangle\{B^3_{\mathtt{0}}, B^3_{\mathtt{I}}\}.\ \mathtt{M} \to \mathtt{W} : \mathsf{more}\langle \mathtt{task}\rangle\{B^4_{\mathtt{0}}, B^4_{\mathtt{I}}\}.\mathbf{t}, \\
\qquad\qquad \mathsf{stop}\langle \mathtt{data}\rangle\{B^3_{\mathtt{0}}, B^3_{\mathtt{I}}\}.\ \mathtt{M} \to \mathtt{W} : \mathsf{stop}\langle\rangle\{B^4_{\mathtt{0}}, B^4_{\mathtt{I}}\}.\mathtt{end}\} \\
G\downarrow_{\mathtt{M}} = \mathtt{W} \oplus \langle \mathtt{task}\rangle\{B^1_{\mathtt{0}}\}. \\
\qquad \mu\mathbf{t}.\ \mathtt{W}\&\langle \mathtt{data}\rangle\{B^2_{\mathtt{I}}\}. \\
\qquad \mathtt{A} \oplus \{\mathsf{more} : \langle \mathtt{data}\rangle\{B^3_{\mathtt{0}}\}.\mathtt{W} \oplus \mathsf{more} : \langle \mathtt{task}\rangle\{B^4_{\mathtt{0}}\}.\mathbf{t}, \\
\qquad\qquad \mathsf{stop} : \langle \mathtt{data}\rangle\{B^3_{\mathtt{0}}\}.\mathtt{W} \oplus \mathsf{stop} : \langle\rangle\{B^4_{\mathtt{0}}\}.\mathtt{end}\}
\end{array}$$

$$\begin{array}{l}
B^1_{\mathtt{0}} = \{x_{\mathtt{M}} = 0, x_{\mathtt{M}}\} \\
B^1_{\mathtt{I}} = \{l \le x_{\mathtt{W}} < 2l,\ _\} \\
B^2_{\mathtt{0}} = \{x_{\mathtt{W}} \le w, x_{\mathtt{W}}\} \\
B^2_{\mathtt{I}} = \{x_{\mathtt{M}} = 2l + w,\ _\} \\
B^3_{\mathtt{0}} = \{x_{\mathtt{M}} = 2l + w,\ _\} \\
B^3_{\mathtt{I}} = \{3l + w \le x_{\mathtt{A}}, x_{\mathtt{A}}\} \\
B^4_{\mathtt{0}} = \{x_{\mathtt{M}} = 2l + w, x_{\mathtt{M}}\} \\
B^4_{\mathtt{I}} = \{x_{\mathtt{W}} = 2l, x_{\mathtt{W}}\}
\end{array}$$

Remark 3.1 (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 § 2 and Example 3.1: the repetition of the same scenario across recursion instances (one for each sampling task) is modelled by having all participants to reset their clocks in their last action before starting a new recursion instance (e.g., $B^3_{\mathtt{I}}$, $B^4_{\mathtt{0}}$ and $B^4_{\mathtt{I}}$ on the second line of G' in Example 3.1).

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., $\mathtt{p}$ sends $l\langle S\rangle$ to $\mathtt{q}$), $\mathtt{pq}?l\langle S\rangle$ 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 $\mathit{subj}(\mathtt{pq}!l\langle S\rangle) = \mathtt{p}$, $\mathit{subj}(\mathtt{qp}?l\langle S\rangle) = \mathtt{p}$ and $\mathit{subj}(t) = \emptyset$.

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 $\mathtt{p}$ but not yet received by $\mathtt{q}$ (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 3.3).

*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 $\mathtt{rdy}(G, \emptyset)$.

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

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

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

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

Fig. 1. Labelled transitions for timed global types

$$(1)\ \begin{array}{c}\mathtt{rdy}(\mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{A_i\}.G_i\}_{i\in I}, D)\\ (\text{with } A_i = \{\delta_{0i}, \lambda_{0i}, \delta_{Ii}, \lambda_{Ii}\})\end{array} = \begin{cases}\{\{\delta_{0i}\}_{i\in I}\} \bigcup_{i\in I} \mathtt{rdy}(G_i, D \cup \{\mathtt{p}, \mathtt{q}\}) & \text{if } \mathtt{p} \notin D\\ \bigcup_{i\in I} \mathtt{rdy}(G_i, D \cup \{\mathtt{p}, \mathtt{q}\}) & \text{otherwise}\end{cases}$$

$$(2)\ \mathtt{rdy}(\mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S\rangle\{\delta_0, \lambda_0, \delta_I, \lambda_I\}.G, D) = \begin{cases}\{\{\delta_I\}\} \cup \mathtt{rdy}(G, D \cup \{\mathtt{q}\}) & \text{if } \mathtt{q} \notin D\\ \mathtt{rdy}(G, D \cup \{\mathtt{q}\}) & \text{otherwise}\end{cases}$$

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

In (1) the send action of $\mathtt{p}$ is ready, hence the singleton including the constraints $\{\delta_{0i}\}_{i\in I}$ 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 3.2 (Satisfiability of ready actions) *We write* $\nu \models^* \mathtt{rdy}(G)$ *when the constraints of all ready actions of G are satisfiable under* ν *or sometimes in the future. Formally,* $\nu \models^* \mathtt{rdy}(G)$ *iff* $\forall \{\{\delta_i\}_{i\in I}\} \in \mathtt{rdy}(G)\ \exists t \geq 0,\ j \in I.\ \nu + t \models \delta_j$.

By requiring the satisfiability of *some* $j \in I$ (i.e., for some branches of a ready action) we allow each action to be executed at any time allowed by its constraints, not necessarily at the earliest possible time. In this way, the semantics of *TG*s specifies the full range of correct behaviours.

The transition rules for *TG*s are given in Figure 1. We assume the execution always begins with initial assignment ν_0. Rule $\lfloor \textsc{select} \rfloor$ 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 predicate λ_0. Rule $\lfloor \textsc{branch} \rfloor$ is the dual for branching. Rules $\lfloor \textsc{async1} \rfloor$ and $\lfloor \textsc{async2} \rfloor$ model interactions that appear later (syntactically), but are not causally dependent on the first interaction. Rule $\lfloor \textsc{time} \rfloor$ 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 $\lfloor \textsc{time} \rfloor$ can always be applied to $(\nu, \mathtt{end})$ since $\nu + t \models^* \mathtt{rdy}(\mathtt{end})$ for all t.

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

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

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

$$(\nu, T[\mu \mathbf{t}.T/\mathbf{t}]) \xrightarrow{\ell} (\nu', T') \;\; \mathit{imply} \;\; (\nu, \mu \mathbf{t}.T) \xrightarrow{\ell} (\nu', T') \qquad \lfloor \textsc{Lrec} \rfloor$$

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

Rule $\lfloor \textsc{Lsel} \rfloor$ is for send actions and its dual $\lfloor \textsc{Lbra} \rfloor$ for receive actions. In rule $\lfloor \textsc{Ltime} \rfloor$ 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) $\ell = t \Rightarrow \forall \mathtt{p} \neq \mathtt{q}.\ (\nu_\mathtt{p}, T_\mathtt{p}) \xrightarrow{\ell} (\nu_\mathtt{p} + t, T_\mathtt{p}) \;\wedge\; w_\mathtt{pq} = w_\mathtt{pq}' \;\wedge$
$T_\mathtt{p} = \mathtt{r}\&\{l_k : \langle S_k \rangle \{\delta_k, \lambda_k\}.T_k\}_{k \in I} \wedge w_\mathtt{rp} = l_m\langle S \rangle \cdot w_\mathtt{rp} \;\; (m \in I) \quad \Rightarrow \quad \nu + t \models^* \{\delta_m\}$

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 3.3 (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	
$\mid \; u[\mathtt{i}](y).P$	Accept	
$\mid \; c[\mathtt{p}] \triangleleft l\langle e \rangle; P$	Select	
$\mid \; c[\mathtt{p}] \triangleright \{l_i(z_i).P_i\}_{i \in I}$	Branching	
$\mid \; \mathtt{delay}(t).P$	Delay	
$\mid \; \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional	
$\mid \; P \mid Q$	Parallel	
$\mid \; \mathbf{0}$	Inaction	
$\mid \; \mu X.P$	Recursion	
$\mid \; X$	Variable	

$\mid \; (\nu a)P$	Hide Shared	
$\mid \; (\nu s)P$	Hide Session	
$\mid \; s : h$	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, 17]). The other processes are standard.

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

Fig. 2. Reduction for processes

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:

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

In the first row: $(\nu s)s:\emptyset=\mathbf{0}$ removes queues of ended sessions, the second rule permutes causally unrelated messages. In the second row: the first rule breaks delays into 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 4.1 (Temperature calculation) Process $P_{\mathtt{M}}$ is a possible implementation of participant $\mathtt{M}$ of the protocol in Example 3.1, 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.

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

Proof rules. We validate programs against specifications based on *TL*s, using judgements of the form $\Gamma\vdash P\rhd\Delta$ 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,

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

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

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

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

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

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

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

Fig. 3. Proof rules for programs

e.g., $\Delta(s[\mathrm{p}]) = (\nu, T)$ when the process being validated is acting as p in session s specified by T; ν is a 'virtual' clock assignment built during the validation.

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 4.2 (Infinitely satisfiable) *G is infinitely satisfiable if either: (1) constraints in recursion bodies have no resets, no equalities nor upper bounds (i.e., $x < c$ or $x \leq c$), 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 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 $\lfloor$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 $\lfloor$vAcc$\rfloor$ for session accept is similar but initiates a new session for participant i. Rule $\lfloor$vBra$\rfloor$ for branching processes checks all the branches in 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 $\lfloor$vSel$\rfloor$ for selection processes checks that the clock constraint δ_j of the selected branch j satisfies the current assignment ν, and updates, in the session environment of the premise, the clock assignment as prescribed by λ_j. Rule $\lfloor$vDel$\rfloor$ for delegation requires δ_d to be satisfied in the assignment ν_d of the delegated session, which is removed from the premise. Rule $\lfloor$vTime$\rfloor$ increments the clock assignments of all sessions in Δ. Rule $\lfloor$vEnd$\rfloor$ validates $\mathbf{0}$ if there are no more actions prescribed by Δ. Rule $\lfloor$vDef$\rfloor$ extends Γ with the assignment

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

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

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

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

Fig. 4. Extended reduction for processes with errors and clocks (selected rules)

for process variable X. Rules $\lfloor$VPar$\rfloor$ and $\lfloor$VCond$\rfloor$ are standard. Rule $\lfloor$VCall$\rfloor$ validates, as usual, recursive call X against $\Gamma(X)$ (and possibly some terminated sessions Δ').

Theorem 4.3 (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 reset predicates (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 in Figure 4: $\lfloor$Link$\rfloor$ introduces a clock process $(s[\mathtt{i}],\nu_0)$ with initial assignment for each participant `i` in the new session; $\lfloor$Delay$\rfloor$ increments all clock assignments, $\lfloor$Sel$\rfloor$ checks the clock constraints against clock assignments and appropriately resets (the rule for branching is extended similarly); $\lfloor$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 $\lfloor$Sel$\rfloor$ only resets the clocks associated to participant `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 4.4 (Time-error freedom) *If $\Gamma \vdash P \triangleright \Delta$, and $P \rightarrow^* P'$ 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. In Figure 5: in (1) if `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. $\mathtt{p} \to \mathtt{q} : \langle \mathtt{int} \rangle \{ x_\mathtt{p} > 3,\ \_ , x_\mathtt{q} = 4,\ \_ \}$
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. $\mu \mathbf{t}.\mathtt{p} \to \mathtt{q} : \langle \mathtt{int} \rangle \{ x_\mathtt{p} < 1, x_\mathtt{p}, x_\mathtt{q} = 2, x_\mathtt{q} \}.\mathtt{p} \to \mathtt{r} : \langle \mathtt{int} \rangle \{ x_\mathtt{p} < 1,\ \_ , \mathtt{true}, x_\mathtt{r} \}.\mathbf{t}$

Fig. 5. Examples of non-feasible (1,5) and feasible (2,3,4,6) global types

(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 *TG*s, a constraint may become unsatisfiable by the restrictions posed 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 the restriction $x_\mathtt{p} < 5$ form the first iteration (e.g., p may send 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*.

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

P implements $G \downarrow_\mathtt{p}$: it waits 6 units of time, then sends q a message and waits for the reply. Q implements $G \downarrow_\mathtt{q}$: it receives a message from 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 4.4, 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 reset predicates, and whose edges model the causal dependencies between actions in (the one-time unfolding of) G. The algorithm yields Proposition 5.1.

Proposition 5.1 (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., $\texttt{erase}(P)$). The *erasure* $\texttt{erase}(P)$ of a timed processes P is defined inductively by removing the delays in P (i.e., $\texttt{erase}(\texttt{delay}(t).P') = \texttt{erase}(P')$), while leaving unchanged the untimed parts (e.g., $\texttt{erase}(\overline{u}[\texttt{n}](y).P') = \overline{u}[\texttt{n}](y).\texttt{erase}(P')$); the other rules are homomorphic.

Proposition 5.2 (Conformance) *If $P \longrightarrow P'$, then $\texttt{erase}(P) \longrightarrow^* \texttt{erase}(P')$.*

Processes implementing multiple sessions may get stuck because of a bad timing of their attempts to initiate new sessions. Consider $P = \texttt{delay}(5).\overline{a}[2](v).P_1 \mid a[2](y).P_2$; $\texttt{erase}(P)$ 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 [18]) conform session delay.

Definition 5.3 (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 5.4 (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 $P_0 = \overline{a}[\texttt{n}](y).P_1 \mid \prod_{i \in \{2,..,n\}} a[\texttt{i}](y).P_i$ where $P_1, \ldots, P_n$ do not contain any name hiding, request/accept, and session receive/delegate.

Corollary 5.5 (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 reset predicates, $\Phi(X)$ 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 $\vec{w} = \{w_{ij}\}_{i \neq j \in \{1,..,n\}}$ 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 reset predicates 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 5.6 (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 $\mathcal{C} \approx \mathcal{A}(G)$.*

[1] Note that: (1) multiparty compatibility allows scenarios with unbounded channels e.g., the channel from $\mathtt{p}$ to $\mathtt{q}$ in $\mu\mathbf{t}.\mathtt{p} \to \mathtt{q} : l\langle S\rangle\{A\}.\mathbf{t}$, and (2) considering 1-bounded executions (for a simpler theory) preserves generality due to a property called *stability* in [11] and directly applicable to our scenario. By stability, if $\mathcal{C}$ is basic and multiparty compatible, then for all its reachable states s there exists an execution $s \longrightarrow^* s'$ from s to a stable state s', and there exists a 1-bounded execution $s_0 \longrightarrow^* s'$ from the initial state s_0 of $\mathcal{C}$ to s'. Namely, after an appropriate execution *any* reachable state can be reached by a 1-bounded execution.

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 5.7 (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.*

6 Conclusion and related work

This work designs a choreographic timed specification based on the semantics of CTAs and MPSTs, and attests its theory in the π-calculus. The table below recalls the results for the untimed setting we build upon (first row), and summarises the results in this work: a decidable proof system for π-calculus processes ensuring time-error freedom and a sound and complete characterisation of CTAs (second row), and two conditions, with decidable algorithms, 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 have verified the practicability of our approach in an implementation of a timed conversation API for Python. The prototype [1] is currently being integrated into the OOI infrastructure [18].

*TG*s	π-calculus	session CTAs
untimed	type safety, error-freedom, progress [13]	Sound, complete characterisation, progress [11]
timed	type safety (Thm 4.3) error-freedom (Thm 4.4)	Sound, complete characterisation, (Thm 5.6)
feasible, wait-free	progress (Thm 5.4, Thm 5.5)	progress (Thm 5.7)

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

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

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. [22] 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: [19] includes time constraints inspired by timed automata into the π-calculus, [5, 17] 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. J. Y. Lee and J. Zic. On modeling real-time mobile processes. *Aust. Comput. Sci. Commun.*, 24(1):139–147, Jan. 2002.
17. 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.
18. Ocean Observatories Initiative (OOI). `http://oceanobservatories.org/`.
19. N. Saeedloei and G. Gupta. Timed π-calculus. In *TGC*, LNCS, 2013. to appear.
20. Savara JBoss Project. `http://www.jboss.org/savara`.
21. Scribble Project homepage. `www.scribble.org`.
22. S. Tripakis. Verifying progress in timed systems. In *Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *LNCS*, pages 299–314. 1999.

Appendix

This appendix includes additional definitions and proofs, as well as detailed related work, and can be read at the committee's discretion.

Table of Contents

A Extended related work (calculi with time)

The different notions of time in existing process calculi could be classified along three main characteristics [**?**]: (1) absolute (time flows at the same pace in all the parts or the system) or relative, (2) continuous (time takes values in $\mathbb{R}^{\geq 0}$) or discrete, and (3) separate (two-phase scheme: time actions and other actions are performed in different phases) or combined (time-stamp scheme). As to (1) and (2), absolute and continuous time, is the combination chosen in our work. We adopted the two-phase scheme as customary in process algebra (whereas timed automata use timestamps).

[19] extends the π-calculus with absolute and continuous times, and includes timed constraints inspired by timed automata in the syntax level of the π-calculus. Timed COWS [15] extends COWS with absolute, continuous and separate time. COWS is enriched with a 'wait' construct similar to our delay. Also the work in [**?**], which is focused on testing, introduces a 'wait' construct similar to our delay, along with delay annotations for actions. [**?**] introduces a declarative framework for reasoning on service oriented systems based on a timed extension of Concurrent Constraint Programming. Activities are organised in sessions, and sessions are explicitly managed via primitives for requesting and accepting a new instance. Some other extensions introduce timeouts, e.g., πRT-calculus [16] (absolute, continuous, separate), and [5] where time elapses by discrete ticks and delays propagate asynchronously through the system. Web-π [**?**] and C^3 [17] extend the π-calculus and the Conversation Calculus (CC), respectively, to enable the reasoning on the interplay between time and exceptions. They are both inspired by the notion of times in [5] and, whereas Web-π focuses on orchestrations of asynchronous services, C^3 focuses on multiparty conversations in a synchronous scenario. [12] introduces a (dense-time) calculus with a time guard construct $[t_{min}, t_{max}]$ specifying a range for the delay for reaching the next state. The authors define a timed labelled transitions systems (TLTS) and a system of equations returning the times in which a process is active.

The main focus of our work is different from the above work: we use timed specifications *as types*, rather than enriching timed primitives in the π-calculus syntax level and reason on examples using the timed LTS. Our aim is to define, on the basis of a timed semantics guided by timed automata, a static checker for time-error freedom and global time-progress, and to design a scalable language for timed multiparty interactions. With this respect, our calculus is a minimal syntactic extension of the π-calculus and is simpler than the above calculi. By integrating its primitives with interrupt constructs in Scribble and Python [**?**], we can write most of the use-cases related with time in [18]. Extensions to other mechanisms such as dynamic time-passing [19] and [16] at the π-calculus level are possible along the lines of [7] and [**?**]: we plan to apply these ideas to future Scribble [21] and Python developments based on our prototype implementation [1].

B Types - extended definitions and proofs

This appendix includes: (§ B.1) further explanation of the rules in the LTS for *TG*s, (§ B.2) definition of projection, (§ B.3) definitions omitted related to LTS of *TL*s, (§ B.5) the proof of Theorem 3.3, and (§ B.4) the definition of infinitely satisfiable *TG*s.

B.1 More on the LTS of timed global types (rules $\lfloor$ASYNC1$\rfloor$ and $\lfloor$ASYNC2$\rfloor$)

Rules $\lfloor$ASYNC1$\rfloor$ and $\lfloor$ASYNC2$\rfloor$ model interactions that appear later (syntactically) but are not causally dependent from the first interaction. For instance, in (1) below

$$\mathtt{p} \to \mathtt{q} : \langle \mathtt{int} \rangle \{x_{\mathtt{p}} < 20,\ \_\ , \mathtt{true},\ \_\ \}.\mathtt{r} \to \mathtt{k} : \langle \mathtt{int} \rangle \{x_{\mathtt{r}} < 10,\ \_\ , \mathtt{true},\ \_\ \} \quad (1)$$

the second interaction could happen before the first one as $\mathtt{p}$ and $\mathtt{r}$ are not causally related; in this case (1) would move by $\lfloor$ASYNC1$\rfloor$ to

$$\mathtt{p} \to \mathtt{q} : \langle \mathtt{int} \rangle \{x_{\mathtt{p}} < 20,\ \_\ , \mathtt{true},\ \_\ \}.\mathtt{r} \rightsquigarrow \mathtt{k} : \langle \mathtt{int} \rangle \{x_{\mathtt{r}} < 10,\ \_\ , \mathtt{true},\ \_\ \} \quad (2)$$

The case for multiple branches is more delicate. Rule $\lfloor$ASYNC1$\rfloor$ can be applied only if (1) the continuation of all branches (i.e., the clause $\forall k \in I$ in the premise of $\lfloor$ASYNCH1$\rfloor$) can make the same action, (2) the action is not causally dependent with other that occur before syntactically (i.e., the clause $\mathtt{p},\mathtt{q} \notin \mathit{subj}(\ell)$ in the premise of $\lfloor$ASYNC1$\rfloor$). Consider G in (3) (we omit the time assertions as they are not relevant for our discussion):

$$\begin{array}{l} G \ = \mathtt{p} \to \mathtt{q} : \{l_1\langle \mathtt{int} \rangle \{\ldots\}.G', l_2\langle \mathtt{int} \rangle \{\ldots\}.G'\} \\ G' = \mathtt{r} \to \mathtt{k} : \{l\langle \mathtt{int} \rangle \{\ldots\}.G''\} \end{array} \quad (3)$$

$\lfloor$ASYNCH1$\rfloor$ can be applied to G in (3). In fact the continuation G' in both branch l_1 and l_2 can make a (causally unrelated) step $\mathtt{rk}!l\langle \mathtt{int} \rangle$ hence

$$G \xrightarrow{\mathtt{rk}!l\langle \mathtt{int} \rangle} \mathtt{p} \to \mathtt{q} : \{l_1\langle \mathtt{int} \rangle \{\ldots\}.G''', l_2\langle \mathtt{int} \rangle \{\ldots\}.G'''\}$$

where $G''' = \mathtt{r} \rightsquigarrow \mathtt{k} : \{l\langle \mathtt{int} \rangle \{\ldots\}.G''\}$.

Scenarios as the *TG* in (4) where the continuations of the branches l_1 and l_2 do not allow the same action, is ruled out because they are not projectable.

$$\begin{array}{l} \mathtt{p} \to \mathtt{q} : \{\ l_1\langle \mathtt{int} \rangle \{\ldots\}.\mathtt{r} \to \mathtt{k} : \{l_3\langle \mathtt{int} \rangle \{\ldots\}.G_3\}, \\ \qquad\qquad\ \ l_2\langle \mathtt{int} \rangle \{\ldots\}.\mathtt{r} \to \mathtt{k} : \{l_4\langle \mathtt{int} \rangle \{\ldots\}.G_4\}\} \end{array} \quad (4)$$

(4) is not projectable because the projections of $\mathtt{r} \to \mathtt{k} : \{l_3\langle \mathtt{int} \rangle \{\ldots\}.G_3\}$ and $\mathtt{r} \to \mathtt{k} : \{l_4\langle \mathtt{int} \rangle \{\ldots\}.G_4\}$ on the sender $\mathtt{r}$ cannot be merged (see the definition of merge operator $\sqcup$ after Definition B.1). We can, instead, project the *TG* in (5):

$$\begin{array}{l} \mathtt{p} \to \mathtt{q} : \{\ l_1\langle \mathtt{int} \rangle \{\ldots\}.\mathtt{q} \to \mathtt{k} : \{l_3\langle \mathtt{int} \rangle \{\ldots\}.G_3\}, \\ \qquad\qquad\ \ l_2\langle \mathtt{int} \rangle \{\ldots\}.\mathtt{q} \to \mathtt{k} : \{l_4\langle \mathtt{int} \rangle \{\ldots\}.G_4\}\} \end{array} \quad (5)$$

but $\lfloor$ASYNCH$\rfloor$ cannot be applied as the second interaction is causally dependent from the first one, i.e., clause $\mathtt{p},\mathtt{q} \notin \mathit{subj}(\ell)$ (with $\ell = \mathtt{qk}!l_3\langle \mathtt{int} \rangle$ in this case) in the premise of $\lfloor$ASYNC1$\rfloor$ is not satisfied.

B.2 Projection of *TG*s onto *TL*s

Definition B.1 describes how to project algorithmically a *TG* onto its participants.

Definition B.1 (Projection) Given G the *projection of G on a* $\mathtt{p} \in G$, written $G \downarrow_{\mathtt{p}}$ is recursively defined as follows assuming $\mathtt{p} \neq \mathtt{q}$:

$$(1)\ \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{B_{Oi}, B_{Ii}\}.G_i\}_{i\in I} \downarrow_{\mathtt{r}} = \begin{cases} \mathtt{q} \oplus \{l_i : \langle S_i\rangle\{B_{Oi}\}.G_i \downarrow_{\mathtt{r}}\}_{i\in I} & \text{if } \mathtt{r} = \mathtt{p} \\ \mathtt{p}\&\{l_i : \langle S_i\rangle\{B_{Ii}\}.G_i \downarrow_{\mathtt{r}}\}_{i\in I} & \text{if } \mathtt{r} = \mathtt{q} \\ \sqcup_{i\in I} G_i \downarrow_{\mathtt{r}} & \text{otherwise} \end{cases}$$

$$(2)\ (\mu \mathbf{t}.G) \downarrow_{\mathtt{r}} = \begin{cases} \mu \mathbf{t}.(G \downarrow_{\mathtt{r}}) & \text{if } G \downarrow_{\mathtt{r}} \neq \mathbf{t} \\ \mathtt{end} & \text{otherwise} \end{cases} \qquad (3)\ \ \mathbf{t} \downarrow_{\mathtt{r}} = \mathbf{t} \qquad (4)\ \ \mathtt{end} \downarrow_{\mathtt{r}} = \mathtt{end}$$

If no side condition applies then G is *not projectable* on $\mathtt{p}$. The time assertions are projected straightforwardly: $\{\delta_{\mathtt{O}}, \lambda_{\mathtt{O}}, \delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\}$ is projected onto the sender by keeping only the output part $\{\delta_{\mathtt{O}}, \lambda_{\mathtt{O}}\}$ and onto the receiver by keeping only the input part $\{\delta_{\mathtt{I}}, \lambda_{\mathtt{I}}\}$. The rest of the definition is standard. Case (1) uses the merge operator $\sqcup$ [**?**,**?**] to ensure that if the locally observable behaviour of the projected local type is not independent of the chosen branch then it is identifiable by $\mathtt{r}$ via a unique label. If the branches cannot be merged then the side condition of case (1) is not satisfied. In this paper we consider *TG*s that are projectable on all their participants.

The merge operator $\sqcup$ [**?**,**?**] is defined below

$$\begin{array}{l} \mathtt{p}\&\{l_i : \langle S_i\rangle\{B_i\}.T_i\}_{i\in I} \sqcup \mathtt{p}'\&\{l_j : \langle S'_j\rangle\{B'_j\}.T'_j\}_{j\in J} = \\ \mathtt{p}\&(\{l_k : \langle S_k\rangle\{B_k\}.T_k\}_{k\in I\setminus J} \cup \{l_k : \langle S'_k\rangle\{B'_k\}.T'_k\}_{k\in J\setminus I} \cup \{l_k : \langle S_k\rangle\{B_k\}.T_k \sqcup T'_k\}_{k\in I\cap J}) \\ \text{where for each } k \in I \cap J,\ \mathtt{p}_k = \mathtt{p}'_k,\ S_k = S'_k \text{ and } B_k = B'_k. \end{array}$$

and ensures that either (a) the locally observable behaviour of the projected local type is either independent of the chosen branch (i.e., $G_i \downarrow_{\mathtt{r}} = G_j \downarrow_{\mathtt{r}}$ for all $i, j \in I$), or (b) the chosen branch is identifiable by $\mathtt{r}$ via a unique label. If the side condition of the merge operator is not satisfied then the two timed local types are not mergeable; in this case the side condition of case (1) of Definition B.1 of projection is not satisfied.

Definition B.2 (Projectable *TG*) *We say that G is* projectable *if there exist* $\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}(G)}$ *such that for all* $\mathtt{p} \in \mathcal{P}(G)$, $G \downarrow_{\mathtt{p}} = T_{\mathtt{p}}$.

In this paper we implicitly consider all G projectable.

B.3 LTS for timed local types

in the LTS for timed local types given in § 3, rule $\lfloor$LTIME$\rfloor$ uses the function $\nu + t \models^* \mathtt{rdy}(T)$ (omitted in the paper due to space constraints as it is similar to the one used in the LTS of timed global types). We give in Definition B.3 its formal definition for timed local types.

Definition B.3 (Ready local clock constraints)

$$\begin{array}{ll} \mathtt{rdy}(\mathtt{p} \oplus \{l_i : \langle S_i\rangle\{\delta_i, \lambda_i\}.T_i\}_{i\in I}) = \mathtt{rdy}(\mathtt{p}\&\{l_i : \langle S_i\rangle\{\delta_i, \lambda_i\}.T_i\}_{i\in I}) = \{\delta_j\}_{i\in I} \\ \mathtt{rdy}(\mu \mathbf{t}.T) & = \mathtt{rdy}(T) \\ \mathtt{rdy}(\mathbf{t}) & = \mathtt{rdy}(\mathtt{end}) = \emptyset \end{array}$$

B.4 Infinitely satisfiable timed global types

First, we give a general characterisation of constraints that make recursive types possibly unsatisfiable, if clocks are not reset. These constraints, which we will call *bad constraints*, are either unrepeatable constraints (i.e., include equalities) and upper-binding constraints.

Definition B.4 (Unrepeatable constraint) δ *is* unrepeatable *iff* $\mathcal{U}(\delta, \texttt{true})$*, where*

$$\mathcal{U}(\delta, b) = \begin{cases} b & \text{if } \delta \text{ is of the form } x = e \\ \mathcal{U}(\delta_1, b) \wedge \mathcal{U}(\delta_2, b) & \text{if } \delta = \delta_1 \wedge \delta_2 \\ \mathcal{U}(\delta', \neg b) & \text{if } \delta = \neg \delta' \end{cases}$$

Definition B.5 (Upper binding constraint) *A constraint* δ *is* upper binding *iff one of the following condition holds:*

δ *is of the form* $x < e$
$\delta = \delta_1 \wedge \delta_2$ *and* δ_i *is upper binding for some* $i \in \{1, 2\}$
$\delta = \neg \delta'$ *and* δ' *is upper binding*

Definition B.6 (Bad constraint) *A constraint* δ *is bad, written* $\mathcal{B}(\delta)$*, if it is unrepeatable or upper binding. We write* $\neg\mathcal{B}(\delta)$ *if* δ *is not bad.*

We say that $\mathtt{p}$ resets x in G iff $\mathtt{p} \in G$ implies one of the following:

1. $G = \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{\delta_{\mathtt{O}i}, \lambda_{\mathtt{O}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}.G_i\}_{i \in I}$ and

$$\exists J \subseteq I.\ (\forall i \in J.\ x \in \lambda_{\mathtt{O}i}) \wedge (\forall i \in (I \setminus J).\ \mathtt{p} \text{ resets } x \text{ in } G_i)$$

2. $G = \mu \mathbf{t}.G'$ and $\mathtt{p}$ resets x in G'

In (1), $\mathtt{p}$ resets x in G iff $\mathtt{p}$ immediately resets x in some branches J and will reset later in the remaining branches $I \setminus J$.

Infinite satisfiability requires that either all constraints are not bad (1) or each participant resets all its clocks at least once within each recursion.

Definition B.7 (Infinitely satisfiable G**)** G_0 *is* infinitely satisfiable *if either:*

1. $\forall \mu \mathbf{t}.G \in G_0$, $\forall \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{\delta_{\mathtt{O}i}, \lambda_{\mathtt{O}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}.G_i\}_{i \in I} \in G$, $\forall i \in I$,

$$\mathcal{B}(\delta_{\mathtt{O}i}) \wedge \mathcal{B}(\delta_{\mathtt{I}i}) \wedge \lambda_{\mathtt{O}i} = \lambda_{\mathtt{I}i} = \emptyset$$

2. $\forall \mu \mathbf{t}.G \in G_0$, $\forall \mathtt{p} \in \mathcal{P}(G)$, $\forall x \in X(\mathtt{p}, G_0)$, $\mathtt{p}$ *resets* x *in* G.

Recall that $X(\mathtt{p}, G_0)$ denote the clocks of $\mathtt{p}$ in G_0. Infinite satisfiability is applied to checking feasibility and wait-freedom in § E.3 (step 5).

B.5 Proof of Theorem 3.3 (soundness, completeness of projection)

The proof uses the following lemma.

Lemma B.8 *Let G be a (projectable, see Definition B.2) TG, and ν be the overriding union of the assignments $\nu_{\mathtt{p}}$ of all $\mathtt{p} \in \mathcal{P}(G)$, then for all $\mathtt{p} \in \mathcal{P}(G)$:*

$$\nu + t \models^* \mathtt{rdy}(G) \quad \textit{iff} \quad \nu_{\mathtt{p}} + t \models^* \mathtt{rdy}(G\downarrow_{\mathtt{p}})$$

The proof of Theorem 3.3 proceeds as the proof of Theorem 3.1 [**?**, Appendix A.1]. First we define (as in [**?**]) a behaviour-preserving subtyping relation $T < T'$ between timed local types which allows to add branches in branching types but not selection types. This subtyping is extended to configurations as follows: $(\nu, \vec{T}, \vec{w}) < (\nu', \vec{T'}, \vec{w'})$ if $\vec{w} = \vec{w'}$, $\nu = \nu'$ and $\forall \mathtt{p} \in \mathcal{P}$, $T_{\mathtt{p}} < T'_{\mathtt{p}}$ and we obtain the following property: if $(\nu, \vec{T}, \vec{w}) < (\nu', \vec{T'}, \vec{w'})$ then $(\nu, \vec{T}, \vec{w}) \approx (\nu', \vec{T'}, \vec{w'})$.

Second, to match global and local types step by step, as in [**?**] we (1) extend projection to intermediary steps and (2) define projected configuration of a global type G, written $[[G]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)}$ to take into account the content of channels by adding into the channels the messages sent in intermediary states i.e.,

$$[[\mathtt{p}' \rightsquigarrow \mathtt{q}' : l\langle S\rangle\{A\}.G]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)} = [[G]]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)}[w_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'} \cdot l\langle S\rangle]$$

Third, as in [**?**] we prove the following step equivalence which directly yields the theorem: if $[[G]] < (\nu, \vec{T}, \vec{w})$ then $G \xrightarrow{\ell} G' \Leftrightarrow (\nu, \vec{T}, \vec{w}) \xrightarrow{\ell} (\nu', \vec{T'}, \vec{w'})$ and $[[G']] < (\nu', \vec{T'}, \vec{w'})$.

The step equivalence is proved by induction on the structure of the transition of G. For soundness we assume $(\nu, G) \xrightarrow{\ell} (\nu', G')$ and show that the corresponding configuration $(\nu, \vec{T}, \vec{w})$ can then make a step ℓ into $(\nu'', \vec{T''}, \vec{w''}) < (\nu', \vec{T'}, \vec{w'})$ where $(\nu', \vec{T'}, \vec{w'})$ is the projection $[[G']]\{w_{\mathtt{pq}}\}_{\mathtt{pq}\in\mathcal{P}(G)}$ of G'. All the cases are identical to [**?**], observing that, by projection, if the clock constraints of G are satisfied then also the constraints in the corresponding configuration are satisfied since both cock constraints and reset predicates of G are exactly the same as those of its projections. The only step that is not treated in [**?**] is the time transition by $\lfloor$Time$\rfloor$. This case follows by Lemma B.8. In fact, if $(\nu, G) \xrightarrow{t} (\nu', G')$ then the premise of $\lfloor$Time$\rfloor$ – $\nu + t \models^* \mathtt{rdy}(G)$ – is satisfied. By Lemma B.8, $\nu + t \models^* \mathtt{rdy}(G)$ yields $\nu + t \models^* \mathtt{rdy}(T_i)$ for all projections T_i hence the premise of $\lfloor$LTime$\rfloor$ for all T_i is satisfied and the configuration can make a time step t.

Completeness proceeds similarly to [**?**] except the step $\lfloor$Time$\rfloor$ which also follows by Lemma B.8.

C Processes - extended definitions

This appendix includes: (§ C.1) the full rules of structural equivalence for processes, and (§ C.2) the syntax, reduction rules for processes and proof rules for programs with *explicit constructs for delegation*,

C.1 Structural equivalence

The structural equivalence relation for processes is the least equivalence relation satisfying the following rules where we denote n for s and a:

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\nu n)P \mid Q \equiv (\nu n)(P \mid Q) \quad \text{if } n \notin \mathtt{fn}(Q)$$
$$(\nu n)(\nu n')P \equiv (\nu n')(\nu n)P \qquad (\nu n)\mathbf{0} \equiv \mathbf{0}$$
$$\mu X.P \equiv P[\mu X.P/X]$$
$$(\nu s)s : \emptyset \equiv \mathbf{0}$$
$$s : (\mathtt{p},\mathtt{q},\mathtt{n}) \cdot (\mathtt{p}',\mathtt{q}',\mathtt{n}') \cdot h \equiv s : (\mathtt{p}',\mathtt{q}',\mathtt{n}') \cdot (\mathtt{p},\mathtt{q},\mathtt{n}) \cdot h \qquad \text{if } \mathtt{p} \neq \mathtt{p}' \text{ or } \mathtt{q} \neq \mathtt{q}'$$

$$\mathtt{delay}(t+t').P \equiv \mathtt{delay}(t).\mathtt{delay}(t').P \qquad \mathtt{delay}(0).P \equiv P$$
$$\mathtt{delay}(t).(\nu a)P \equiv (\nu a)\mathtt{delay}(t).P \qquad \mathtt{delay}(t).(P \mid Q) \equiv \mathtt{delay}(t).P \mid \mathtt{delay}(t).Q$$

The rules at the top are standard. We just recall that rule $(\nu s)s : \emptyset \equiv \mathbf{0}$ is for garbage collection of queues of terminated sessions, and the rule for queues in the last line (on the top) is for permuting messages that are causally unrelated (i.e., can be received in any order due to asynchrony). At the bottom, $\mathtt{delay}(t).\mathtt{delay}(t').P \equiv \mathtt{delay}(t+t').P$ breaks a delay into smaller intervals, $\mathtt{delay}(0).P \equiv P$ always allows time to elapse for idle processes, and the rules in the second line distribute a delay within a shared name restriction and in parallel compositions.

C.2 Processes with explicit delegation

For simplicity of presentations of the proofs, the syntax in Figure 6 divides the syntax in the main section into explicit constructs for session delegation and constants [6]. We also set delegation interactions (both outgoing and incoming) have only one branch. The extension to multiple branches is mechanical.

P ::=	$\overline{u}[\mathtt{n}](y).P$	Request	$\mid P \mid Q$	Parallel
$\mid$	$u[\mathtt{i}](y).P$	Accept	$\mid \mathbf{0}$	Inaction
$\mid$	$c[\mathtt{p}] \triangleleft l\langle e\rangle;P$	Select	$\mid \mu X.P$	Recursion
$\mid$	$c[\mathtt{p}] \triangleright \{l_i(z_i).P_i\}_{i\in I}$	Branching	$\mid X$	Variable
$\mid$	$c[\mathtt{p}] \triangleleft \langle\langle c', \nu\rangle\rangle;P$	Delegate	$\mid (\nu a)P$	Hide Shared
$\mid$	$c[\mathtt{p}] \triangleright ((y)).P$	Session receive	$\mid (\nu s)P$	Hide Session
$\mid$	$\mathtt{delay}(t).P$	Delay	$\mid s : h$	Queue
$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional		

Fig. 6. Syntax of processes with explicit constructs for delegation

Hereafter in this appendix we will use the following reduction rules for processes: the rules presented in Figure 2 where $\lfloor$SEL$\rfloor$ and $\lfloor$BRA$\rfloor$ are only used when the sort of the exchanged message is not of the form (T,δ), and the rules for delegation in Figure 7 for the delegation cases.

The proof rules for programs for the syntax with explicit delegation are: the rules in Figure 8 without $\lfloor$VBRA$\rfloor$ and $\lfloor$VSEL$\rfloor$ (which are substituted with the new rules in Figure 8), and the rules in Figure 8.

$$s[\mathtt{p}][\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \quad \lfloor \textsc{Del} \rfloor$$
$$s[\mathtt{p}][\mathtt{q}] \triangleright ((y)).P \mid s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h \longrightarrow P[s'[\mathtt{p}']/y] \mid s : h \qquad \lfloor \textsc{SRec} \rfloor$$

Fig. 7. Additional reduction rules for processes (for explicit modelling of delegation)

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

$$\lfloor \textsc{VDel} \rfloor \ \frac{\Gamma \vdash P \triangleright \Delta, c : ([\lambda \mapsto 0]\nu, T') \quad \nu \models \delta \quad \nu_d \models \delta_d}{\Gamma \vdash c[\mathtt{p}] \triangleleft \langle\langle c'\rangle\rangle; P \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \langle (T_d, \delta_d)\rangle\{\delta, \lambda\}.T'), c' : (\nu_d, T_d)}$$

$$\lfloor \textsc{VBra} \rfloor \ \frac{\forall i \in I \quad \Gamma, z_i : S_i \vdash P_i \triangleright \Delta, c : ([\lambda_i \mapsto 0]\nu, T_i) \quad \nu \models \delta_i}{\Gamma \vdash \ c[\mathtt{p}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \triangleright \Delta, c : (\nu, \mathtt{p}\&\{l_i : \langle S_i\rangle\{\delta_i, \lambda_i\}.T_i\}_{i\in I})}$$

$$\lfloor \textsc{VSRec} \rfloor \ \frac{\Gamma \vdash P \triangleright \Delta, c : ([\lambda \mapsto 0]\nu, T'), y : (\nu_d, T_d) \quad \nu \models \delta \quad \nu_d \models \delta_d}{\Gamma \vdash \ c[\mathtt{p}] \triangleright ((y)).P \triangleright \Delta, c : (\nu, \mathtt{p}\&\langle (T_d, \delta_d)\rangle\{\delta, \lambda\}.T')}$$

Fig. 8. Proof rules for programs with explicit delegation

C.3 Extension to runtime processes

In this appendix we extend the typing system of programs to type run-time processes in which sessions have been initiated and queues may be non-empty. This extension is necessary for subject reduction.

We first extend timed local types to include run-time queues.

$$\begin{array}{lll} M ::= \langle \mathtt{p}, U\rangle \mid M; M & & \text{Message Types} \\ \mathcal{T} ::= M \mid T \mid M; T & & \text{Generalised Types} \\ \Sigma ::= M \mid (\nu, \mathcal{T}) & & \text{Generalised Specifications} \\ U ::= l\langle S\rangle & & \end{array}$$

The equivalence relation $\equiv$ for generalised specifications is as follows:

$$\begin{array}{l} \langle \mathtt{p}, U\rangle; \langle \mathtt{p}', U'\rangle; M \equiv \langle \mathtt{p}', U'\rangle; \langle \mathtt{p}, U\rangle; M \quad \text{if } \mathtt{p} \neq \mathtt{p}' \\ M \equiv M' \text{ implies } M; \mathcal{T} \equiv M'; \mathcal{T} \\ \mathcal{T} \equiv \mathcal{T}' \text{ implies } (\nu, \mathcal{T}) \equiv (\nu, \mathcal{T}') \end{array}$$

We will use (generalised) session environments mapping $s[\mathtt{p}]$ to generalised specifications. A session environment can be combined with another session environments containing only message types as follows:

$$\Delta; \{s[\mathtt{p}] : M\} = \begin{cases} \Delta', s[\mathtt{p}] : M'; M & \text{if } \Delta = \Delta', s[\mathtt{p}] : M' \\ \Delta, s[\mathtt{p}] : M & \text{otherwise} \end{cases}$$

Figure 9 generalises the proof rules for programs in Figure 8 to processes. The proof rules for queues are on the top of Figure 9. The generalised proof rules rely on the following definitions:

- $\Delta \equiv \Delta'$ if $c : \Sigma \in \Delta$ and $\Sigma \neq (\nu, \mathtt{end})$ imply $c : \Sigma' \in \Delta'$ with $\Sigma \equiv \Sigma'$ and vice-versa.
- $\Delta * \Delta' = \Delta \setminus dom(\Delta') \cup \Delta' \setminus dom(\Delta) \cup \{y : \Sigma * \Sigma' \mid y : \Sigma \in \Delta \wedge y : \Sigma' \in \Delta'\}$
- $\Sigma * \Sigma' = \begin{cases} \Sigma; \Sigma' & \text{if } \Sigma \text{ is a message type} \\ \Sigma'; \Sigma & \text{if } \Sigma' \text{ is a message type} \\ \text{undefined} & \text{otherwise} \end{cases}$

$$\frac{}{\Gamma \vdash s : \emptyset \triangleright \emptyset} \qquad \frac{\Gamma \vdash s : h \triangleright \Delta \qquad \Gamma \vdash v : S}{\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \triangleright \Delta; \{s[\mathtt{p}] : \langle \mathtt{q}, l\langle S \rangle \rangle\}} \quad \lfloor \text{QInit} \rfloor / \lfloor \text{QSend} \rfloor$$

$$\frac{\Gamma \vdash s : h \triangleright \Delta}{\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \triangleright \Delta; s'[\mathtt{p}'] : (\nu, T); \{s[\mathtt{p}] : \langle \mathtt{q}, \langle (T, \delta) \rangle \rangle\}} \quad \lfloor \text{QDel} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Delta \equiv \Delta'}{\Gamma \vdash P \triangleright \Delta'} \qquad \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta * \Delta'} \quad \lfloor \text{Equiv} \rfloor / \lfloor \text{GPar} \rfloor$$

$$\frac{\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a) P \triangleright \Delta} \qquad \frac{\Gamma \vdash P \triangleright \Delta'; \{s[\mathtt{p}_i] : \Sigma_i\}_{i \in I} \quad \{s[\mathtt{p}_i] : \Sigma_i\}_{i \in I} \text{ coherent}}{\Gamma \vdash (\nu s) P \triangleright \Delta'} \quad \lfloor \text{GNRes/GSRes} \rfloor$$

Fig. 9. Proof rules for processes (the rules on the top are for queues)

Note on coherence of Σ The projection of the generalised local timed type $T \downarrow_{\mathtt{q}}$ is a straightforward extension of the one in [6] (where constraints are just projected as they are). The duality relation of local timed types, written $T \bowtie T'$ is also derived from [6] (where the relation holds for any clock constraints and reset predicate). $(\nu, T) \bowtie (\nu', T')$ if $T \bowtie T'$. We say that $\Delta = \{s[\mathtt{p}_i] : \Sigma_i\}_{i \in I}$ is coherent if $s[\mathtt{p}] : (\nu, T) \in \Delta$ and $T \downarrow_{\mathtt{q}} \neq \mathtt{end}$ imply $s[\mathtt{q}] : T' \in \Delta$ and $T \downarrow_{\mathtt{q}} \bowtie T' \downarrow_{\mathtt{p}}$.

Reduction rules for session environments are in Figure 10.

$$\frac{i \in J \quad \delta_i \models \nu \quad \text{if } \lambda_i = \emptyset \text{ then } \nu' = \nu \text{ else } \nu' = 0}{\{s[\mathtt{p}] : (\nu, M; \mathtt{q} \oplus \{l_j : \langle S_j \rangle \{\delta_j, \lambda_j\}.T_j\}_{j \in J})\} \longrightarrow \{s[\mathtt{p}] : (\nu', M; \langle \mathtt{q}, l_i\langle S_i \rangle \rangle; T_i)\}} \quad [\text{RSend}]$$

$$\frac{i \in J \quad \delta_i \models \nu \quad \text{if } \lambda_i = \emptyset \text{ then } \nu' = \nu \text{ else } \nu' = 0}{\begin{array}{l}\{s[\mathtt{p}] : (\nu, \langle \mathtt{q}, l_i\langle S_i \rangle \rangle; \mathcal{T}); s[\mathtt{q}] : (\nu'', \mathtt{p} \& \{l_j : \langle S_j \rangle \{\delta_j, \lambda_i\}.T_j\}_{j \in J})\} \longrightarrow \\ \qquad \{s[\mathtt{p}] : (\nu, \mathcal{T}); s[\mathtt{q}] : (\nu', T_i)\}\end{array}} \quad [\text{RRec}]$$

$$\frac{\forall i \in I, j \in J,\ \nu_{ij} + t \models \mathtt{rdy}(\mathcal{T}_{ij})}{\{\{s_i[\mathtt{p}_j] : (\nu_{ij}, \mathcal{T}_{ij})\}_{j \in J_i}\}_{i \in I} \longrightarrow \{\{s_i[\mathtt{p}_j] : (\nu_{ij} + t, \mathcal{T}_{ij})\}_{j \in J_i}\}_{i \in I}} \quad [\text{RTime}]$$

$$\frac{\Delta \longrightarrow \Delta' \quad \Delta' \ \mathtt{not\ after}\ \Delta}{\Delta \cup \Delta'' \longrightarrow \Delta' \cup \Delta''} \quad [\text{RPar}]$$

$\Delta' \ \mathtt{not\ after}\ \Delta$ if $\forall s[\mathtt{p}] \in \mathit{Dom}(\Delta)$, $\Delta(s[\mathtt{p}]) = (\nu, \Sigma)$ implies
$\Delta'(s[\mathtt{p}])$ is either (ν, Σ') or $(0, \Sigma')$ for some Σ'

Fig. 10. Reduction rules for specifications

D Time-error freedom - definitions and proofs

This appendix includes: (§ D.1) the proof of type preservation under structural equivalence, (§ D.2) auxiliary lemmas for type preservation under reduction (subject reduc-

tion), (§ D.3) the proof of subject reduction, (§ D.4) auxiliary lemmas for time-error freedom and (§ D.5) the proof of time-error freedom.

D.1 Type preservation under equivalence

The proof rules in Figure 8 ensure type preservation under structural equivalence, which we prove below after giving an auxiliary lemma.

Lemma D.1 *If* $\Delta = \Delta_P * \Delta_Q$ *and* $dom(\Delta_P) \cap dom(\Delta_Q) = \emptyset$*, then* $\Delta + t = (\Delta_P + t) * (\Delta_Q + t)$ *and* $dom(\Delta_P + t) \cap dom(\Delta_Q + t) = \emptyset$.

Theorem D.2 (Type preservation under equivalence) *If* $\Gamma \vdash P \triangleright \Delta$ *and* $P \equiv P'$ *then* $\Gamma \vdash P' \triangleright \Delta$.

Proof. We only show the cases for delay processes. The other cases are as in [6].

The case for $\mathtt{delay}(t).0 \equiv 0$ follows from the fact that both processes can be always validated (for all t): the first by one application of rule $\lfloor$VTIME$\rfloor$ and one of $\lfloor$END$\rfloor$, and the second one application of rule $\lfloor$END$\rfloor$.

The case for $\mathtt{delay}(t+t').P \equiv \mathtt{delay}(t).\mathtt{delay}(t').P$ proceeds as follows. If $\Gamma \vdash \mathtt{delay}(t).\mathtt{delay}(t').P \triangleright \Delta$ then by Lemma D.3 (6) we obtain $\Gamma \vdash P \triangleright \Delta + t + t'$ which by associativity of sum of non-negative reals yields

$$\Gamma \vdash P \triangleright \Delta + (t + t') \tag{6}$$

By proof rule $\lfloor$VTIME$\rfloor$ on (6) we obtain $\Gamma \vdash \mathtt{delay}(t+t').P \triangleright \Delta$ as required. If, reversely $\Gamma \vdash \mathtt{delay}(t+t').P \triangleright \Delta$ by applying once Lemma D.3 (6) we obtain $\Gamma \vdash P \triangleright \Delta + (t + t')$ which by associativity of $+$ yields

$$\Gamma \vdash P \triangleright \Delta + t + t' \tag{7}$$

By applying twice rule $\lfloor$VTIME$\rfloor$ on (7) we obtain $\Gamma \vdash \mathtt{delay}(t).\mathtt{delay}(t').P \triangleright \Delta$.

The case for $\mathtt{delay}(t).(\nu a)P \equiv (\nu a)\mathtt{delay}(t).P$ proceeds as follows. We first consider the case $\Gamma \vdash \mathtt{delay}(t).(\nu a)P \triangleright \Delta$. By premise of $\lfloor$VTIME$\rfloor$,

$$\Gamma \vdash (\nu a)P \triangleright \Delta + t \tag{8}$$

and by premise of $\lfloor$GNRES$\rfloor$ on (8)

$$\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta + t \tag{9}$$

We apply (9) as a premise of $\lfloor$VTIME$\rfloor$ obtaining

$$\Gamma, a : \langle G \rangle \vdash \mathtt{delay}(t).P \triangleright \Delta \tag{10}$$

and apply (10) as the premise of $\lfloor$GNRES$\rfloor$ obtaining $\Gamma \vdash (\nu a)\mathtt{delay}(t).P \triangleright \Delta$ as required. The case for $\Gamma \vdash (\nu a)\mathtt{delay}(t).P \triangleright \Delta$ proceeds similarly.

The case for $\mathtt{delay}(t).(P \mid Q) \equiv \mathtt{delay}(t).P \mid \mathtt{delay}(t).Q$ proceeds as follows. If $\Gamma \vdash \mathtt{delay}(t).(P \mid Q) \triangleright \Delta$ then by premise of $\lfloor$VPAR$\rfloor$

$$\Gamma \vdash \mathtt{delay}(t).P \triangleright \Delta_P \qquad \Gamma \vdash \mathtt{delay}(t).Q \triangleright \Delta_Q \tag{11}$$

where $\Delta = \Delta_P * \Delta_Q$ and by the premise of $\lfloor$VTIME$\rfloor$ on each judgement in (11)

$$\Gamma \vdash P \triangleright \Delta_P + t \qquad \Gamma \vdash Q \triangleright \Delta_Q + t \tag{12}$$

We apply both judgements in (12) each as the premise of $\lfloor$VPAR$\rfloor$ and by Lemma (D.1):

$$\Gamma \vdash P \mid Q \triangleright \Delta + t \tag{13}$$

We apply (13) as the premise of $\lfloor$VTIME$\rfloor$ obtaining $\Gamma \vdash \texttt{delay}(t).(P \mid Q) \triangleright \Delta$, as required. The case for $\Gamma \vdash \texttt{delay}(t).(P \mid Q) \triangleright \Delta$ proceeds similarly.

D.2 Auxiliary lemmas (type preservation under reduction)

Lemma D.3 (Inversion Lemma) *The following holds (by induction on derivations):*

1. *If* $\Gamma \vdash P \mid Q \triangleright \Delta$ *then* $\Delta = \Delta_1 * \Delta_2$, *and* $\Gamma \vdash P \triangleright \Delta_1$ *and* $\Gamma \vdash Q \triangleright \Delta_2$.
2. *If* $\Gamma \vdash c[\mathtt{p}] \triangleleft l_i\langle e\rangle; P \triangleright \Delta$ *then*
   (a) $\Gamma \vdash e : S_j$,
   (b) $i \in J$,
   (c) $\Delta = \Delta', c : (\nu, \mathtt{p} \oplus \{l_j : \langle S_j\rangle\{\delta_j, \lambda_j\}.T_j\}_{j\in J})$,
   (d) $\delta_i \models \nu$,
   (e) $\Gamma \vdash P \triangleright \Delta', c : (\nu', T_i)$ *with* $\nu' = \nu$ *if* $\lambda_i = \emptyset$ *and* $\nu' = 0$ *otherwise.*
3. *If* $\Gamma \vdash c[\mathtt{p}] \triangleleft \langle\langle c'\rangle\rangle; P \triangleright \Delta$ *then*
   (a) $\Delta = \Delta', c : (\nu, \mathtt{p} \oplus \langle(T_d, \delta_d)\rangle\{\delta, \lambda\}.T), c' : (\nu_d, T_d)$,
   (b) $\delta \models \nu$,
   (c) $\delta_d \models \nu_d$,
   (d) $\Gamma \vdash P \triangleright \Delta', c : (\nu', T)$ *with* $\nu' = \nu$ *if* $\lambda_i = \emptyset$ *and* $\nu' = 0$ *otherwise.*
4. *If* $\Gamma \vdash c[\mathtt{p}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \triangleright \Delta$ *then*
   (a) $\Delta = \Delta', c : (\nu, \mathtt{p}\&\{l_i : \langle S_i\rangle\{\delta_i, \lambda_i\}.T_i\}_{i\in I})$,
   (b) $\forall i \in I$, $\delta_i \models \nu$,
   (c) $\forall i \in I\ \Gamma \vdash P_i[v/z_i] \triangleright \Delta', c : (\nu', T_i)$ *with* $\nu' = \nu$ *if* $\lambda_i = \emptyset$ *and* $\nu' = 0$ *otherwise.*
5. *If* $\Gamma \vdash c[\mathtt{p}] \triangleright ((y)).P \triangleright \Delta$ *then*
   (a) $\Delta = \Delta', c : (\nu, \mathtt{p}\&\langle(T_d, \delta_d)\rangle\{\delta, \lambda\}.T)$,
   (b) $\delta_i \models \nu$,
   (c) $\delta_d \models \nu_d$,
   (d) $\Gamma \vdash P \triangleright \Delta', c : (\nu', T), y : (\nu_d, T_d)$ *with* $\nu' = \nu$ *if* $\lambda_i = \emptyset$ *and* $\nu' = 0$ *otherwise, and* $\nu_d \models \delta_d$.
6. *If* $\Gamma \vdash \texttt{delay}(t).P \triangleright \Delta$ *then*
   (a) $\Delta = \{c_i : (\nu_i, T_i)\}$,
   (b) $\Gamma \vdash P \triangleright \{c_i : (\nu_i + t, T_i)\}$
7. *If* $\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v\rangle) \triangleright \Delta$ *then*
   (a) $\Delta = \Delta'; \{s[\mathtt{p}] : \langle \mathtt{q}, l\langle S\rangle\rangle\}$,
   (b) $\Gamma \vdash s : h \triangleright \Delta'$
8. *If* $\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \triangleright \Delta$ *then*
   (a) $\Delta = \Delta'; \{s[\mathtt{p}] : \langle \mathtt{q}, \langle(T_d, \delta_d)\rangle\rangle\}, s'[\mathtt{p}'] : (\nu_d, T_d)$,
   (b) $\Gamma \vdash s : h \triangleright \Delta'$

Lemma D.4 (Satisfiability upon validation) *If* $\Gamma \vdash P \triangleright \Delta$ *with* $\Delta = \{c_i : (\nu_i + t, T_i)\}_{i\in I}$ *for some* $t \geq 0$ *then* $\forall i \in I$, $\nu_i + t \models^* \texttt{rdy}(T_i)$.

Proof The proof is my induction on the proof tree of P, with a case analysis of the final rule. We only show the case for selection. The cases for delegation and branching proceed similarly to the case for selection. The remaining cases are straightforward (e.g., for parallel, delay and recursion they hold by inductive hypothesis, for end and recursive call $\texttt{rdy}()$ is trivially satisfiable). If the last rule is $\lfloor$VSEL$\rfloor$, $\Gamma \vdash c[\mathtt{p}] \triangleleft l_k\langle e\rangle; P \triangleright \Delta$. By Lemma D.3 (2))

$$\Delta = \Delta', c : (\nu + t, \mathtt{p} \oplus \{l_j : \langle S_j\rangle\{\delta_j, \lambda_j\}.T_j\}_{j\in J}) \quad k \in J \tag{14}$$

and

$$\delta_k = \nu + t \tag{15}$$

By Definition B.3

$$\texttt{rdy}(\mathtt{p} \oplus \{l_j : \langle S_j\rangle\{\delta_j, \lambda_j\}.T_j\}_{j\in J}) = \{\delta_j\}_{j\in J} \tag{16}$$

By Definition B.3 it is sufficient that one of δ_j is satisfiable for $\{\delta_j\}_{j\in J}$ to be satisfiable. Hence by (15), (16) is satisfiable after t in ν, i.e., $\nu + t \models^* \{\delta_j\}_{j\in J}$. The thesis follows by induction.

D.3 Proof of Theorem 4.3 (type preservation – subject reduction)

Remark D.1. In the statement of Theorem 4.3, P is typed against empty Δ meaning that either (1) P has not started any session yet, or (2) P models a whole system, that is including all participants of all ongoing sessions, technically $P \equiv (\nu\, \vec{s})P'$ for some $\vec{s}'$ and with P' having no free session names. Note in fact that by rule $\lfloor$GSRES$\rfloor$ in Figure 9 a whole system $(\nu s)P$ is typed against $\emptyset$ as long as P is typed against a set of coherent types.

We have stated this special case in the paper for conciseness (as it did not require to introduce the semantics of Δ). Here, however, we proceed as customary and prove Theorem 4.3 via its general case (which subsumes Theorem 4.3).

Theorem D.5 (Type preservation for open systems) *If $\Gamma \vdash P \triangleright \Delta$ and $P \longrightarrow P'$, then there exists Δ' such that $\Delta \longrightarrow \Delta'$ and $\Gamma \vdash P' \triangleright \Delta'$*

Theorem D.5 requires to extend the typing rules for programs to processes (this extension is mechanical and similar to the one in [6], and is reported in Appendix C.3). The proof is by induction on the derivation $P \longrightarrow P'$, with a case analysis on the final rule (using Theorem D.2 for the structural equivalence).

- $\lfloor$LINK$\rfloor$ - This case proceeds as in [6] (Appendix A - Proof of Theorem 4.1).
- $\lfloor$SEL$\rfloor$ - If the reduction is by $\lfloor$SEL$\rfloor$ then $s[\mathtt{p}][\mathtt{q}] \triangleleft l_j\langle e\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v\rangle)$ (with $e \downarrow v$), and by hypothesis

$$\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleleft l_j\langle e\rangle; P \mid s : h \triangleright \Delta$$

By Lemma D.3 (1)

$$\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleleft l_j\langle e\rangle; P \triangleright \Delta_1 \tag{17}$$

and

$$\Gamma \vdash s : h \triangleright \Delta_2 \tag{18}$$

with $\Delta_1 * \Delta_2 = \Delta$. By Lemma D.3 (2) on (17)

$$\Delta_1 = \Delta_1', s[\mathtt{p}] : (\nu, \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I}) \tag{19}$$

with $\Gamma \vdash e : S_j$ which, by subject reduction of expressions, yields

$$\Gamma \vdash v : S_j \tag{20}$$

Furthermore, also by Lemma D.3 (2) on (17):

$$\Gamma \vdash P \triangleright \Delta_1', s[\mathtt{p}] : (\nu', T_j) \quad \text{with } \nu' = \nu \text{ if } \lambda_j = \emptyset \text{ and } \nu' = 0 \text{ otherwise} \tag{21}$$

$$\delta_j \models \nu \tag{22}$$

By $\lfloor$QSEND$\rfloor$ on (18) we derive

$$\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \triangleright \Delta_2, \{s[\mathtt{p}] : \langle \mathtt{q}, l_j\langle S_j \rangle \rangle\} \tag{23}$$

Using $\lfloor$GPAR$\rfloor$ on (21) and (23)

$$\Gamma \vdash P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \triangleright (\Delta_1', s[\mathtt{p}] : (\nu', T_j)) * \Delta_2; \{s[\mathtt{p}] : \langle \mathtt{q}, l_j\langle S_j \rangle \rangle\}$$

Note that by $\lfloor$RPAR$\rfloor$ with premise $\lfloor$RSEND$\rfloor$ – the latter can be applied thanks to (22) – on $\Delta_1 * \Delta_2$ where Δ_1 is as in (19):

$$(\Delta_1', s[\mathtt{p}] : (\nu, \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I}) * \Delta_2 \longrightarrow (\Delta_1', s[\mathtt{p}] : (\nu', T_j)) * (\Delta_2, \{s[\mathtt{p}] : \langle \mathtt{q}, l_j\langle S_j \rangle \rangle\})$$

– $\lfloor$BRA$\rfloor$ - If the reduction is by $\lfloor$BRA$\rfloor$ then $s[\mathtt{p}][\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i \in I} \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \cdot h \longrightarrow P_j[v/z_j] \mid s : h$, and by hypothesis

$$\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i \in I} \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \cdot h \triangleright \Delta$$

By Lemma D.3 (1)

$$\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i \in I} \triangleright \Delta_1 \tag{24}$$

and

$$\Gamma \vdash s : (\mathtt{p}, \mathtt{q}, l_j\langle v \rangle) \cdot h \triangleright \Delta_2 \tag{25}$$

with $\Delta_1 * \Delta_2 = \Delta$. By Lemma D.3 (4) on (24)

$$\Delta_1 = \Delta_1', s[\mathtt{p}] : (\nu, \mathtt{q} \& \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I}) \tag{26}$$

and

$$\forall i \in I \quad \Gamma \vdash P_i[v/z_i] \triangleright \Delta_1', s[\mathtt{p}] : (\nu', T_i) \quad \text{with } \nu' = \nu \text{ if } \lambda_i = \emptyset \text{ and } \nu' = 0 \text{ otherwise} \tag{27}$$

$$\forall i \in I \quad \delta_i \models \nu \tag{28}$$

By Lemma D.3 (7) on (25)

$$\Gamma \vdash s : h \triangleright \Delta_2' \qquad \Delta_2 = \Delta_2', \{s[\mathtt{p}] : \langle \mathtt{q}, l_i\langle S_i\rangle\rangle\} \tag{29}$$

Using $\lfloor$GPar$\rfloor$ on (27) and (29) for all $i \in I$:

$$\Gamma \vdash P[v/z_j] \mid s : h \triangleright (\Delta_1', s[\mathtt{p}] : (\nu', T_j)) * \Delta_2'$$

Note that by $\lfloor$RPar$\rfloor$ with premise $\lfloor$RRec$\rfloor$ – the latter can be applied thanks to (28) – on $\Delta_1 * \Delta_2$:

$$(\Delta_1', s[\mathtt{p}] : (\nu, \mathtt{q}\&\{l_i : \langle S_i\rangle\{\delta_i, \lambda_i\}.T_i\}_{i\in I}) * \Delta_2', \{s[\mathtt{p}] : \langle \mathtt{q}, l_j\langle S_j\rangle\rangle\} \longrightarrow (\Delta_1', s[\mathtt{p}] : (\nu', T_j)) * \Delta_2'$$

– $\lfloor$SRec$\rfloor$ - If the reduction is by $\lfloor$SRec$\rfloor$ then $s[\mathtt{p}][\mathtt{q}] \triangleright ((y)).P \mid s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h \longrightarrow P[s'[\mathtt{p}']/y] \mid s : h$, and by hypothesis

$$\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleright ((y)).P \mid s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h \triangleright \Delta$$

By Lemma D.3 (1)

$$\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleright ((y)).P \triangleright \Delta_1 \tag{30}$$

and

$$\Gamma \vdash s : (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \cdot h \triangleright \Delta_2 \tag{31}$$

with $\Delta_1 * \Delta_2 = \Delta$. By Lemma D.3 (5) on (30)

$$\Delta_1 = \Delta_1', s[\mathtt{p}] : (\nu, \mathtt{q}\&\langle (T_d, \delta_d)\rangle\{\delta, \lambda\}.T) \tag{32}$$

and

$$\Gamma \vdash P[s'[\mathtt{p}']/y] \triangleright \Delta_1', s[\mathtt{p}] : (\nu', T), s'[\mathtt{p}'] : (\nu_d, T_d) \quad \text{with } \nu' = \nu \text{ if } \lambda_i = \emptyset \text{ and } \nu' = 0 \text{ otherwise} \tag{33}$$

and

$$\delta \models \nu \qquad \delta_d \models \nu_d \tag{34}$$

By Lemma D.3 (8) on (31)

$$\Gamma \vdash s : h \triangleright \Delta_2' \qquad \Delta_2 = \Delta_2', \{s[\mathtt{p}] : \langle \mathtt{q}, \langle (T_d, \delta_d)\rangle\rangle\}, s'[\mathtt{p}'] : (\nu_d, T_d) \tag{35}$$

Using $\lfloor$GPar$\rfloor$ on (33) and (35):

$$\Gamma \vdash P[s'[\mathtt{p}']/y] \mid s : h \triangleright \Delta_1', s[\mathtt{p}] : (\nu', T), s'[\mathtt{p}'] : (\nu_d, T_d) * \Delta_2'$$

Note that by $\lfloor$RPar$\rfloor$ with premise $\lfloor$RRec$\rfloor$ – the latter can be applied thanks to (34) – on $\Delta_1 * \Delta_2$:

$$(\Delta_1', s[\mathtt{p}] : (\nu, \mathtt{q}\&\langle (T_d, \delta_d)\rangle\{\delta, \lambda\}.T) * \Delta_2', \{s[\mathtt{p}] : \langle \mathtt{q}, \langle (T_d, \delta_d)\rangle\rangle\}, s'[\mathtt{p}'] : (\nu_d, T_d) \longrightarrow (\Delta_1', s[\mathtt{p}] : (\nu', T)) * \Delta_2'$$

– $\lfloor$DEL$\rfloor$ - If the reduction is by $\lfloor$DEL$\rfloor$ then $s[\mathtt{p}][\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P \mid s : h \longrightarrow P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}'])$, and by hypothesis

$$\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P \mid s : h \triangleright \Delta$$

By Lemma D.3 (1)

$$\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P \triangleright \Delta_1 \tag{36}$$

and

$$\Gamma \vdash s : h \triangleright \Delta_2 \tag{37}$$

with $\Delta_1 * \Delta_2 = \Delta$. By Lemma D.3 (3) on (36)

$$\Delta_1 = \Delta_1', s[\mathtt{p}] : (\nu, \mathtt{q} \oplus \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T), s'[\mathtt{p}'] : (\nu_d, T_d), \tag{38}$$

and

$$\Gamma \vdash P \triangleright \Delta_1', s[\mathtt{p}] : (\nu', T) \quad \text{with } \nu' = \nu \text{ if } \lambda = \emptyset \text{ and } \nu' = 0 \text{ otherwise} \tag{39}$$

$$\delta \models \nu \text{ and } \delta_d \models \nu_d \tag{40}$$

By $\lfloor$QDEL$\rfloor$ on (37) we derive

$$\Gamma \vdash s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \triangleright \Delta_2, \{s[\mathtt{p}] : \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle\} \tag{41}$$

Using $\lfloor$GPAR$\rfloor$ on (39) and (41)

$$\Gamma \vdash P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, \langle (T_d, \delta_d) \rangle) \triangleright (\Delta_1', s[\mathtt{p}] : (\nu', T)) * \Delta_2, \{s[\mathtt{p}] : \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle\}$$

Note that by $\lfloor$RPAR$\rfloor$ with premise $\lfloor$RSEND$\rfloor$ – the latter can be applied thanks to (40) – on $\Delta_1 * \Delta_2$ where Δ_1 is as in (19):

$$(\Delta_1', s[\mathtt{p}] : (\nu, \mathtt{q} \oplus \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T) * \Delta_2 \longrightarrow (\Delta_1', s[\mathtt{p}] : (\nu', \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle; T)) * \Delta_2$$

with

$$(\Delta_1', s[\mathtt{p}] : (\nu', \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle; T)) * \Delta_2 = (\Delta_1', s[\mathtt{p}] : (\nu', T)) * \Delta_2, \{s[\mathtt{p}] : \langle \mathtt{q}, \langle (T_d, \delta_d) \rangle \rangle\}$$

– $\lfloor$DELAY$\rfloor$ - If the reduction is by $\lfloor$DELAY$\rfloor$ then $\mathtt{delay}(t).P_i \mid \prod_{j \in J} s_j : h_j \longrightarrow P \prod_{j \in J} s_j : h_j$, and by hypothesis

$$\Gamma \vdash \mathtt{delay}(t).P \mid \prod_{j \in J} s_j : h_j \triangleright \Delta$$

By Lemma D.3 (1)

$$\Gamma \vdash \mathtt{delay}(t).P_i \triangleright \Delta_1 \qquad \Gamma \vdash \prod_{j \in J} s_j : h_j \triangleright \Delta_2 \qquad (\delta = \delta_1 * \delta_2) \tag{42}$$

By Lemma D.3 (6) on (42 - judgement on the left hand side) $\Delta_1 = \{s_j[\mathtt{p}_j] : (\nu_j, T_j)\}_{j \in J}$ and

$$\Gamma \vdash P \triangleright \Delta_1' \qquad (\Delta_1' = \{s_j[\mathtt{p}_j] : (\nu_j + t, T_j)\}_{j \in J}) \tag{43}$$

Using (43) as a premise of ($\lfloor$VPAR$\rfloor$), observing that $dom(\Delta_1) \cap dom(\Delta_2)\} = \emptyset$ implies $dom(\Delta_1') \cap dom(\Delta_2)\} = \emptyset$

$$\Gamma \vdash P \mid \prod_{j \in J} s_j : h_j \triangleright \Delta_1' * \Delta_2 \tag{44}$$

By Lemma D.4 on (44)(recall $\Delta_1' = \{s_j[\mathtt{p}_j] : (\nu_j + t, T_j)\}_{j \in J}$), T_j is satisfiable after t in ν_j, hence by $\lfloor$RTIME$\rfloor$ $\Delta \longrightarrow \Delta_1' * \Delta_2$.

$P ::=$	$\overline{u}[\mathtt{n}](y).P$	Request			
$\mid$	$u[\mathtt{i}](y).P$	Accept	$\mid X$	Variable	
$\mid$	$c[\mathtt{p}] \triangleleft \{\delta,\lambda\} l\langle e\rangle; P$	Select	$\mid (\nu a)P$	Hide Shared	
$\mid$	$c[\mathtt{p}] \triangleright \{\{\delta_i,\lambda_i\} l_i(z_i).P_i\}_{i\in J}$	Branching	$\mid (\nu s)P$	Hide Session	
$\mid$	$c[\mathtt{p}] \triangleleft \{\delta,\lambda\} \langle\langle c',\nu\rangle\rangle; P$	Delegate	$\mid (c,\nu)$	Clock Assignment	
$\mid$	$c[\mathtt{p}] \triangleright \{\delta,\lambda\}((y)).P$	Session receive	$\mid s:h$	Queue	
$\mid$	$\mathtt{delay}(t).P$	Delay	$h ::= \emptyset \mid h\cdot(\mathtt{p},\mathtt{q},m)$	(queue content)	
$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional	$m ::= l\langle v\rangle$	(messages)	
$\mid$	$P \mid Q$	Parallel	$c ::= s[\mathtt{p}] \mid y$	(session names/variables)	
$\mid$	0	Inaction	$u ::= a \mid z$	(shared names/variables)	
$\mid$	$\mu X.P$	Recursion	$n ::= a \mid s$	(names)	
$\mid$	$\mathtt{error}$	Error			

Fig. 11. Extended syntax of processes for time-error freedom.

D.4 Auxiliary definitions and lemmas (time-error freedom)

Technically, we can prove error freedom through the following steps.

Step 1 - extend processes In our scenario an error state, which we call `error`, is a state reached when a process violates the clock constraints associated to the action prescribed by the corresponding local timed type. To model reduction to error states we extend the syntax of processes with clock constraints, reset predicate and clock assignment (see Figure 11).

Step 2 - extend reductions The extended reduction rules are given in Figure 12, plus rules $\lfloor$IFT$\rfloor$, $\lfloor$IFF$\rfloor$, $\lfloor$CONT$\rfloor$, $\lfloor$STR$\rfloor$, and $\lfloor$COM$\rfloor$, from Figure 2 that remain unchanged. Rule $\lfloor$LINK$\rfloor$ extends session initiation by adding an initial clock assignment $(s[\mathtt{i}],\nu_0)$ for each participant `i` of the new session. The rules for selection and branching, $\lfloor$SEL$\rfloor$ and $\lfloor$BRA$\rfloor$ check clock constraints against clock assignments and appropriately reset the clock assignments. Rules $\lfloor$DEL$\rfloor$ and $\lfloor$SREC$\rfloor$ for session delegation update the set of clock processes. Time actions by $\lfloor$DELAY$\rfloor$ increment the clock assignments. We introduce error reductions to be triggered when a process tries to perform an action in a time that does not satisfy the prescribed constraint: $\lfloor$EBRA$\rfloor$, $\lfloor$ESEL$\rfloor$, $\lfloor$EDEL$\rfloor$ and $\lfloor$ESREC$\rfloor$ are error reductions for send, branching, delegation, session receive. Note that rules $\lfloor$EBRA$\rfloor$ and $\lfloor$ESREC$\rfloor$ can be applied both when p tries to read too early or too late w.r.t. the prescribed time.

Step 4 - Extended Validation Rules Figure 13 gives the extended proof rules for processes with errors and clocks. We show the rules that are different from Figure 8 and omit the rules that are unchanged.

Lemma D.6 *If* $\Gamma \vdash P \triangleright \Delta$, *then* $P \not\equiv \mathtt{error}$.

The proof of Lemma D.6 is mechanical by induction on G proceeding by case analysis and inspecting the extended proof rules.

D.5 Proof of Theorem 4.4 (time-error freedom)

Let $\Gamma \vdash P_0 \triangleright \Delta$ and $P_0 \longrightarrow^* P \longrightarrow P'$. We proceed by induction on the length of the reduction: we assume by induction that $P \not\equiv \mathtt{error}$ and proceed by case analysis on the last reduction $P \longrightarrow P'$.

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

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

$$\frac{j\in J \qquad \nu' = [\lambda_j \mapsto 0]\nu \qquad \delta_j \models \nu}{s[\mathtt{p}][\mathtt{q}] \triangleright \{\{\delta_i,\lambda_i\} l_i(z_i).P_i\}_{i\in J} \mid s:(\mathtt{p},\mathtt{q},l_j\langle v\rangle)\cdot h \mid (s[\mathtt{p}],\nu) \longrightarrow P_j[v/z_j] \mid s:h \mid (s[\mathtt{p}],\nu')} \lfloor \textsc{Bra} \rfloor$$

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

$$\frac{\nu' = [\lambda \mapsto 0]\nu \qquad \delta \models \nu}{s[\mathtt{p}][\mathtt{q}] \triangleright \{\delta,\lambda\}((y)).P \mid s:(\mathtt{p},\mathtt{q},(s'[\mathtt{p}'],\nu''))\cdot h \mid (s[\mathtt{p}],\nu) \longrightarrow P[s'[\mathtt{p}']/y] \mid s:h \mid (s[\mathtt{p}],\nu') \mid (s'[\mathtt{p}'],\nu'')} \lfloor \textsc{SRec} \rfloor$$

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

$$\frac{j\in J \qquad \neg\delta_j \models \nu}{s[\mathtt{p}][\mathtt{q}] \triangleright \{\{\delta_i,\lambda_i\} l_i(z_i).P_i\}_{i\in J} \mid s:h \mid (s[\mathtt{p}],\nu) \longrightarrow \mathtt{error} \mid s:h \mid (s[\mathtt{p}],\nu)} \lfloor \textsc{EBra} \rfloor$$

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

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

$$\frac{\neg\delta \models \nu}{s[\mathtt{p}][\mathtt{q}] \triangleright \{\delta,\lambda\}((y)).P \mid s:h \mid (s[\mathtt{p}],\nu) \longrightarrow \mathtt{error} \mid s:h \mid (s[\mathtt{p}],\nu)} \lfloor \textsc{ESRec} \rfloor$$

Fig. 12. Extended reduction (showing modified/added rules, omitting rules that are as in Figure 2).

In the case analysis, we use again induction on the (depth of the tree of the) reduction rule applied. The inductive cases, $\lfloor \textsc{Cont} \rfloor$, $\lfloor \textsc{Str} \rfloor$ and $\lfloor \textsc{Com} \rfloor$, and are straightforward. We consider now the base cases. If the last reduction is by rules $\lfloor \textsc{Link} \rfloor$, $\lfloor \textsc{Delay} \rfloor$, $\lfloor \textsc{Sel} \rfloor$, $\lfloor \textsc{Bra} \rfloor$, $\lfloor \textsc{Del} \rfloor$, $\lfloor \textsc{SRec} \rfloor$, $\lfloor \textsc{IfT} \rfloor$, or $\lfloor \textsc{IfF} \rfloor$, then $P' \neq \mathtt{error}$ since (1) the rules do not introduce error processes, and (2) error processes are a run-time process which cannot appear as a continuation of P. Furthermore, we can proceed similarly to Theorem 4.3 and show that $\Gamma' \vdash P' \triangleright \Delta'$ for some Γ' and Δ' (obtained by some reduction from Γ and Δ). This yields the result for these cases.

The only reductions that introduce error processes are $\lfloor \textsc{ESel} \rfloor$, $\lfloor \textsc{EBra} \rfloor$, $\lfloor \textsc{EDel} \rfloor$, $\lfloor \textsc{ESRec} \rfloor$. We consider $\lfloor \textsc{ESel} \rfloor$ (the others proceed similarly). Assume $\lfloor \textsc{ESel} \rfloor$ can be applied to P which is therefore of the following form: $s[\mathtt{p}][\mathtt{q}] \triangleleft \{\delta,\lambda\} l\langle e\rangle; P'$. Then by $\lfloor \textsc{ESel} \rfloor$

$$\neg\nu \models \delta \tag{45}$$

By hypothesis P is well typed. By inspecting the validation rules, the derivation of $s[\mathtt{p}][\mathtt{q}] \triangleleft \{\delta,\lambda\} l\langle e\rangle; P'$ is done by applying the rule $\lfloor \textsc{vSel} \rfloor$

$$\frac{\Gamma \vdash e : S_j \quad \Gamma \vdash P' \mid (s[\mathtt{p}],\nu') \triangleright \Delta, s[\mathtt{p}] : (\nu', T') \quad \nu' = [\lambda_j \mapsto 0]\nu \qquad \nu \models \delta}{\Gamma \vdash s[\mathtt{p}][\mathtt{q}] \triangleleft \{\delta,\lambda\} l\langle e\rangle; P' \mid (s[\mathtt{p}],\nu) \triangleright \Delta, s[\mathtt{p}] : (\nu, \mathtt{q} \oplus \{l_i : \langle S_i\rangle\{\delta_i,\lambda_i\}.T'_i\}_{i\in I})}$$

with the condition $\nu \models \delta$, which contradicts (45) and the fact that $\lfloor \textsc{ESel} \rfloor$ can be applied. The fact that none of the reductions that introduce errors can be applied (together with the fact that $P \not\equiv \mathtt{error}$) yields the thesis.

$$\frac{\Gamma, u : G \vdash P \mid (y[1], \nu_0) \triangleright \Delta, y[1] : (\nu_0, G\downarrow_1) \quad dom(\nu_0) = \{x_1\} \quad \nu_0(x_1) = 0}{\Gamma, u : G \vdash \overline{u}[\mathtt{n}](y).P \triangleright \Delta} \lfloor \text{VREQ} \rfloor$$

$$\frac{\Gamma, u : G \vdash P \mid (y[\mathtt{i}], \nu_0) \triangleright \Delta, y[\mathtt{i}] : (\nu_0, G\downarrow_{\mathtt{i}}) \quad dom(\nu_0) = \{x_{\mathtt{i}}\} \quad \nu_0(x_{\mathtt{i}}) = 0 \quad \mathtt{i} \neq 1}{\Gamma, u : G \vdash u[\mathtt{i}](y).P \triangleright \Delta} \lfloor \text{VACC} \rfloor$$

$$\frac{j \in I \quad \Gamma \vdash e : S_j \quad \Gamma \vdash P \mid (c, \nu') \mid \prod (c_i, \nu_i) \triangleright \Delta, c : (\nu', T_j) \quad \nu' = [\lambda_j \mapsto 0]\nu \qquad \delta_j \models \nu}{\Gamma \vdash c[\mathtt{p}] \triangleleft \{\delta, \lambda\} l_j \langle e \rangle; P \mid (c, \nu) \mid \prod (c_i, \nu_i) \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I})} \lfloor \text{VSEL} \rfloor$$

$$\frac{\Gamma \vdash P \mid (c, \nu') \mid \prod (c_i, \nu_i) \triangleright \Delta, c : (\nu', T') \qquad \nu' = [\lambda \mapsto 0]\nu \qquad \nu \models \delta \qquad \nu_d \models \delta_d}{\Gamma \vdash c[\mathtt{p}] \triangleleft \{\delta, \lambda\} \langle\langle c', \nu_d \rangle\rangle; P \mid (c, \nu) \mid (c', \nu_d) \mid \prod (c_i, \nu_i) \triangleright \Delta, c : (\nu, \mathtt{p} \oplus \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T'), c' : (\nu_d, T_d)} \lfloor \text{VDEL} \rfloor$$

$$\frac{\forall i \in I \quad \Gamma, z_i : S_i \vdash P_i \mid (c, \nu_i') \mid \prod (c_i, \nu_i) \triangleright \Delta, c : (\nu_i, T_i) \quad \nu_i' = [\lambda_i \mapsto 0]\nu \qquad \nu \models \delta_i}{\Gamma \vdash \ c[\mathtt{p}] \triangleright \{\{\delta_i, \lambda_i\} l_i(z_i).P_i\}_{i \in J} \mid (c, \nu) \mid \prod (c_i, \nu_i) \triangleright \Delta, c : (\nu, \mathtt{p} \& \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I})} \lfloor \text{VBRA} \rfloor$$

$$\frac{\Gamma \vdash P \mid (c, \nu') \mid \prod (c_i, \nu_i) \triangleright \Delta, c : (\nu', T'), y : (\nu_d, T_d) \quad \nu' = [\lambda \mapsto 0]\nu \qquad \nu \models \delta \quad \nu_d \models \delta_d}{\Gamma \vdash \ c[\mathtt{q}] \triangleright \{\delta, \lambda\}((y)).P \mid (c, \nu) \mid (c', \nu_d) \mid \prod (c_i, \nu_i) \triangleright \Delta, c : (\nu, \mathtt{p} \& \langle (T_d, \delta_d) \rangle \{\delta, \lambda\}.T')} \lfloor \text{VSREC} \rfloor$$

$$\frac{\Gamma \vdash P \mid (s[\mathtt{p}], \nu + t) \mid \prod (c_i, \nu_i + t) \triangleright \{c_i : (\nu_i + t, T_i)\}_{i \in I}}{\Gamma \vdash \mathtt{delay}(t).P \mid (s[\mathtt{p}], \nu) \mid \prod (c_i, \nu_i) \triangleright \{c_i : (\nu_i, T_i)\}_{i \in I}} \lfloor \text{VTIME} \rfloor$$

$$\frac{\begin{array}{c} dom(\Delta_1) \cap dom(\Delta_2) = \emptyset \\ \Gamma \vdash P_i \triangleright \Delta_i \quad P_i \equiv P_i' \mid (c, \nu) \Rightarrow c \notin dom(\Delta_j) \quad i \neq j \in \{1, 2\} \end{array}}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1, \Delta_2} \lfloor \text{VPAR} \rfloor$$

$$\frac{\forall c \in dom(\Delta) \quad \Delta(c) = (\nu, \mathtt{end})}{\Gamma \vdash 0 \mid \prod_i (c_i, \nu_i) \triangleright \Delta} \lfloor \text{VEND} \rfloor$$

Fig. 13. Extended reduction rules (with error processes).

E Feasibility and wait-freedom - definitions and proofs

This appendix includes: (§ E.1) definition of time graphs used by the checker for feasibility and way-freedom, (§ E.2) definition of virtual state, (§ E.3) the algorithms for checking feasibility and wait-freedom, auxiliary lemmas on (§ E.4) general properties of clock constraints in recursions, (§ E.5) feasibility, and (§ E.6) wait-freedom, and (§ E.7) the proof of Proposition 5.1. We will use the assumptions and definition discussed below.

Remark E.1. In § 3, following [3, Definition 3.6], the syntax of δ does not allow *clocks comparison*. For instance, clocks of the form $x < x'$ cannot be expressed. Constraints can, however, be defined on more than one clock (e.g., $x_1 < 10 \wedge x_2 > 3$).

Remark E.2. In the proofs we assume that each participant owns at most one clock. This assumption is without loss of generality because of the compositionality of the checker in § E.3 w.r.t. the different clocks of a participant. Definitions E.7 and E.10 (checking satisfiability/wait-freedom in a state) can be decomposed in independent checks on each clock owned by the participant performing an action in that state. In fact, there are no interdependencies on the values of clocks in a given state (clock comparisons are not allowed - see Remark E.1). The extension of the proofs to multiple participants is mechanical (and verbose) hence omitted.

Remark E.3. We assume fairness in the choice of branches: (1) if a branching is met infinitely often each branch is eventually taken, assuming that its constraint is satisfiable, and (2) if a communication step can occur it will not be deferred infinitely often.

By Remark E.1, for every δ built with the syntax in § 3 we can build a logically equivalent (by associativity and commutativity of logical conjunction) constraint δ' of the form $\delta_0 \wedge (\wedge_{x \in fn(\delta)} \delta_x)$ where $fn(\delta_0) = \emptyset$, and $fn(\delta_x) = x$ for all $x \in fn(\delta)$. We call δ' a canonical form of δ. Note that δ may have more than one canonical forms, which are logically equivalent, and that $\delta_\emptyset$ is logically equivalent to either `true` or `false`.

Definition E.1 (Components of δ) *Let δ be a clock constraint (defined with the syntax in § 3) and $\delta_0 \wedge (\wedge_{x \in fn(\delta)} \delta_x)$ one of its canonical forms. The set of components of δ is $\{\delta_0\} \cup \{\delta_x\}_{x \in fn(\delta)}$.*

E.1 Time graphs

To make dependencies between constraints explicit when checking feasibility and wait-freedom, we create a representation of global timed types as *time graphs*.

Let G be a timed global type, a subterm G' of G, written $G' \in G$, is a timed global type that appears in G. To build the time graph of G we first annotate each $G' \in G$ with a distinguished name, ranged over by $\overline{\mathtt{n}}$. The nodes $\mathtt{n}$ of a timed graph are of the form $(\overline{\mathtt{n}}, i, !)$ for output action in branch i of an interaction type, $(\overline{\mathtt{n}}, i, ?)$ for input action in branch i, $(\overline{\mathtt{n}}, \mu\mathbf{t})$ for recursive type, $(\overline{\mathtt{n}}, \mathbf{t})$ for type variable and the special node `end`. We build the set of nodes from an annotated timed global type as follows:

$$
\begin{array}{ll}
\mathtt{nodes}(\overline{\mathtt{n}} : \mathtt{p} \to \mathtt{q} : \{l_i\langle S_I\rangle\{A_i\}.G_i\}_{i\in I}) & = \{(\overline{\mathtt{n}}, i, \#) \mid i \in I,\ \# \in \{!, ?\}\} \cup \\
& \quad \bigcup_{i\in I} \mathtt{nodes}(G_i) \\
\mathtt{nodes}(\overline{\mathtt{n}} : \mu\mathbf{t}.G') & = \{(\overline{\mathtt{n}}, \mu\mathbf{t})\} \cup \mathtt{nodes}(G') \\
\mathtt{nodes}(\mathbf{t}) & = \{(\overline{\mathtt{n}}, \mathbf{t})\} \\
\mathtt{nodes}(\overline{\mathtt{n}} : \mathtt{end}) & = \{\mathtt{end}\}
\end{array}
$$

The set of nodes deriving from G but not from the subterms of G, written $\mathtt{t\_nodes}(G)$, is defined as $\mathtt{nodes}(G) \setminus \{\mathtt{n} \mid \mathtt{n} \in \mathtt{nodes}(G') \text{ and } G' \in G\}$. Vice-versa, we use the function $\mathtt{term}_G(\mathtt{n})$ to return the subterm of annotated type G corresponding to node $\mathtt{n}$. We will use the following auxiliary functions that can be defined straightforwardly from $\mathtt{term}_G(\mathtt{n})$. Let $\mathtt{n} \in \mathtt{nodes}(G)$ and

$$\mathtt{term}_G(\mathtt{n}) = \mathtt{p} \to \mathtt{q} : \{l_i\langle S_i\rangle\{\delta_{\mathtt{0}i}, \lambda_{\mathtt{0}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}.G_i\}_{i\in I}$$

then:

- *subj*$(\mathtt{n})$ returns the participant that is performing the action associated to $\mathtt{n}$. More precisely: let $\mathtt{n} = (\overline{\mathtt{n}}, i, \#)$ then *subj*$(\mathtt{n}) = \mathtt{p}$ if $\# = !$ and *subj*$(\mathtt{n}) = \mathtt{q}$ otherwise.
- $\mathtt{const}(\mathtt{n})$ returns: $\delta_{\mathtt{0}i}$ if $\mathtt{n} = (\overline{\mathtt{n}}, i, !)$, and $\delta_{\mathtt{I}i}$ if $\mathtt{n} = (\overline{\mathtt{n}}, i, ?)$.
- $\mathtt{resInfo}(\mathtt{n})$ returns: $\lambda_{\mathtt{0}i}$ if $\mathtt{n} = (\overline{\mathtt{n}}, i, !)$, and $\lambda_{\mathtt{I}i}$ if $\mathtt{n} = (\overline{\mathtt{n}}, i, ?)$.

Function $\mathtt{const}(\mathtt{n})$ returns `true` and all other functions above return $\emptyset$ if $\mathtt{term}_G(\mathtt{n})$ is not an interaction type.

Definition E.2 (Time graph of a *TG*) *The time graph of G is a pair (N,E) where the set of nodes $N = \mathtt{nodes}(G)$ and E is defined as follows:*

- *(I/O dependency) for each $(\overline{\mathtt{n}},i,!)$ and $(\overline{\mathtt{n}},i,?)$ in N, $((\overline{\mathtt{n}},i,!),(\overline{\mathtt{n}},i,?)) \in E$,*
- *(single participant dependency) for each pair of nodes $\mathtt{n}_1 = (\overline{\mathtt{n}},i,\#)$ and $\mathtt{n}_2 = (\overline{\mathtt{n}},i',\#')$ in N, if $\mathtt{term}_G(\mathtt{n}_2) \in \mathtt{term}_G(\mathtt{n}_1)$, $subj(\mathtt{n}_1) = subj(\mathtt{n}_2)$ and $\overline{\mathtt{n}} \neq \overline{\mathtt{n}}'$ then $(\mathtt{n}_1,\mathtt{n}_2) \in E$.*
- *(syntactic dependency and recursion) for each node $\mathtt{n} \in N$:*
  1. *if $\mathtt{term}_G(\mathtt{n}) = \mu \boldsymbol{t}.G'$ then $(\mathtt{n},\mathtt{t\_nodes}(G')) \in E$, and*
  2. *if $\mathtt{n} = (\overline{\mathtt{n}},i,\#)$ and $\mathtt{term}_G(\mathtt{n}) = \mathtt{p} \to \mathtt{q} : \{l_j\langle S_j\rangle\{\delta_{\mathtt{0}j},\lambda_{\mathtt{0}j},\delta_{\mathtt{I}j},\lambda_{\mathtt{I}j}\}.G_j\}_{j\in J}$ then for all j such that $G_j \in \{\boldsymbol{t},\mathtt{end}\}$, $(\mathtt{n},\mathtt{t\_nodes}(G_j)) \in E$.*

The edges of the time graph of G define an immediate successor relation (i.e., $(\mathtt{n},\mathtt{n}') \in N$) between nodes written $\mathtt{n} <_G \mathtt{n}'$. We refer to path in the time graph as $\mathtt{n}_0 <_G \ldots <_G \mathtt{n}_m$ or more concisely $\mathtt{n}_0,\ldots,\mathtt{n}_m$. We say that a path $\mathtt{n}_0,\ldots,\mathtt{n}_m$ has postfix $\mathtt{n}_i,\ldots,\mathtt{n}_m$ for $0 < i < m$. A path from $\mathtt{n}$ to $\mathtt{n}'$ is a path of the form $\mathtt{n},\ldots,\mathtt{n}'$. We write $\mathtt{n},p$ for the path obtained by appending path p to $\mathtt{n}$. Similarly, we write $\mathtt{p},\mathtt{n}$ for the path obtained appending node $\mathtt{n}$ to p. We write $p \setminus p'$ for the path obtained by removing from p all the nodes in p' (assuming node names unique within a path).

E.2 Virtual time

To reason on the properties of a G the checker will use extended constraints that allow to compare clocks. Clock comparison is used to compose different constraints occurring in G. The aim is to determine the possible times in which an action in G can happen on the basis of the constraints that occur earlier of that action in G.

Definition E.3 (Extended clock constraints) *The set of extended clock constraints δ on X is:*

$$\begin{array}{l}\delta ::= \mathtt{true} \mid x > e \mid x = e \mid \neg\delta \mid \delta_1 \wedge \delta_2 \mid \\ e ::= x \mid c \mid e + e\end{array}$$

The extension of $fc(\delta)$ to constraints on multiple clocks is straightforward. Hereafter we implicitly assume constraints are extended constraints.

To check satisfiability of a constraint we model, given the time graph of a G, the time scenario in which node $\mathtt{n}$ can be reached in an execution of G. We call this scenario *virtual time* of $\mathtt{n}$, written $\overline{\delta}_{\mathtt{n}}$. This virtual time is calculated by considering that the current instant in time must not precede the the time of past actions. The time of past actions is determined considering the past constraints (as reset may have occurred, the virtual time must be appropriately shifted forward w.r.t. to constraints referring to reset clocks). We will use the following notation: assume participant $\mathtt{p}$ owns j clocks ($j \in \mathbb{N}$), we denote with $x_{\mathtt{p}j}$ the $j-th$ clocks owned by $\mathtt{p}$, with $x^{\mathtt{n}}_{\mathtt{p}j}$ the state of the $j-th$ clocks owned by $\mathtt{p}$ in node $\mathtt{n}$, with $\vec{x_{\mathtt{p}}}$ the vector of all clocks owned by $\mathtt{p}$, and with $\vec{x^{\mathtt{n}}_{\mathtt{p}}}$ the vector of all clock states of clocks owned by $\mathtt{p}$ in $\mathtt{n}$.

Definition E.4 *The sum of* δ *with a clock x, written* $\delta + x$ *is defined as follows:*

$$\begin{array}{l}
\texttt{true} + x = \texttt{true} \\
(x' \ \texttt{bop}\ e) + x = \begin{cases} x' \ \texttt{bop}\ (e+x) & (f(x,x') = \texttt{true}) \\ x' \ \texttt{bop}\ e & (f(x,x') = \texttt{false}) \end{cases} \quad \textit{with } (\texttt{bop} \in \{\wedge, \vee, =\}) \\
(\neg\delta) + x = \neg(\delta + x) \\
\textit{with} \quad f(x^{\mathtt{n}}_{\mathtt{p}j}, x^{\mathtt{n}'}_{\mathtt{p}'j'}) = \begin{cases} \texttt{true} & (\mathtt{p} = \mathtt{p}',\ j = j') \\ \texttt{false} & (\textit{otherwise}) \end{cases}
\end{array}$$

The sum of δ *with a vector of clocks is* $\delta + \vec{x} = (\delta + x_0) + \vec{x}'$ *with* $\vec{x} = x_0 + \vec{x}'$.

To calculate the virtual time in $\mathtt{n}$, $\overline{\delta}_{\mathtt{n}}$, we use a function R that takes a node and a participant and returns the sum of clock states in which this participant has reset the clock. Let us set default initial values to recursively calculate the time scenario of a node: $\overline{\delta}_0 = \texttt{true}$ and $\overline{R}(\mathtt{n}_0, \mathtt{p}, j) = 0$ for all $\mathtt{p} \in \mathcal{P}(G)$ with $j \in X(\mathtt{p}, G)$ (recall from § 3 that $X(\mathtt{p}, G)$ is the set of clocks owned by $\mathtt{p}$ in G). Let $\mathtt{n}$ be a node in the time graph of the all-unfolding of G, $\texttt{const}(\mathtt{n}) = \delta$, and $M = \{\mathtt{n}' \mid \mathtt{n}' \in N \text{ and } \mathtt{n}' <_G \mathtt{n}\}$. We denote with $\{\delta_\emptyset\} \cup \{\delta_j\}_{j \in J}$ the components of δ (see Definition E.1) where, abusing the notation, we let $\mathit{fn}(\delta_j) = \{x_{\mathtt{p}j}\}$ for each $j \in J$ (if $\mathit{subj}(\mathtt{n}) = \emptyset$ then $\delta = \delta_\emptyset$).

Definition E.5 (Virtual time)

$$\overline{\delta}_{\mathtt{n}} = \wedge_{\mathtt{n}' \in M} \overline{\delta}_{\mathtt{n}'}(\vec{x}) \wedge \delta_\emptyset \wedge \textstyle\bigwedge_{j \in J} ((\delta_j[x^{\mathtt{n}}_{\mathtt{p}j} / x_{\mathtt{p}j}] + \overline{R}(\mathtt{n}', \mathtt{p}, j)) \wedge (\vec{x} \leq x^{\mathtt{n}}_{\mathtt{p}j}))$$

$$\overline{R}(\mathtt{n}, \mathtt{p}, j) = \begin{cases} \sum_{\mathtt{n}' \in M} \overline{R}(\mathtt{n}', \mathtt{p}, j) + x^{\mathtt{n}}_{\mathtt{p}j} & \textit{if } \texttt{resInfo}(\mathtt{n}) = \{x_{\mathtt{p}j}\} \\ \sum_{\mathtt{n}' \in M} \overline{R}(\mathtt{n}', \mathtt{p}, j) & \textit{otherwise} \end{cases}$$

E.3 Algorithms for checking feasibility and wait-freedom

Feasibility. Feasibility of G requires the satisfiability of each constraint in G, in every possible scenario that satisfies the previously occurred constraints. We also need to take care that each constraint occurring in the body of a recursive global timed type can be satisfied when the body is executed infinitely many times.

For recursion we must consider that the action associated to each node $\mathtt{n}$ in a recursion body can be executed in two different scenarios: (1) in the first iteration, (2) in successive iterations. In (2) we must consider that the time scenario in successive iterations is affected by time constraints occurring syntactically after $\mathtt{n}$ in the recursion body. To account for this, we check feasibility on the one-time unfolding of *all* the recursions in a *TG*. We will show the conditions by which one unfolding is sufficient to guarantee that $\mathtt{n}$ will be also satisfiable in successive unfoldings (hence iteration instances).

Step 1 : all-unfolding. The first step for checking feasibility of G is to make the one-time unfolding of all recursions in G.

Step 2 : the time graph. A time graph is generated on the all-unfolding of G.

Step 3 : the dependency constraints We define for each node of the time graph of G a dependency constraint that models the possible time scenarios in which the action associated to that node is executed (i.e., the virtual time $\overline{\delta}_{\mathtt{n}}$ in Definition E.5).

Definition E.6 (Dependency constraint and reset) *Let* $\mathtt{n}$ *be a node in* G *and* $M = \{\mathtt{n}' \mid \mathtt{n}' \in N$ *and* $\mathtt{n}' <_G \mathtt{n}\}$. *The dependency constraint of* $\mathtt{n}$ *is defined as follows:*

$$\delta_{\mathtt{n}} = \wedge_{\mathtt{n}' \in M} \overline{\delta}_{\mathtt{n}'}$$

The dependency reset of $\mathtt{n}$ *is defined as follows:*

$$R(\mathtt{n}, \mathtt{p}, j) = \sum_{\mathtt{n}' \in M} \overline{R}(\mathtt{n}', \mathtt{p}, j)$$

Step 4 : checking satisfiability of time graph We define time graph satisfiability (Definition E.8) in terms of node satisfiability (Definition E.7).

Definition E.7 (Satisfiable node) *Let* $\mathtt{n}$ *be a node of the (all-unfolding) time graph of* G, *and* $\mathtt{const}(\mathtt{n}) = \delta$. *If* $subj(\mathtt{n}) = \mathtt{p}$ *and* $fn(\delta) \neq \emptyset$ *we denote with* $\{\delta_0\} \cup \{\delta_j\}_{j \in J}$ *the components of* δ *(see Definition E.1) where, abusing the notation, we let* $fn(\delta_j) = \{x_{\mathtt{p}j}\}$ *for each* $j \in J$ *(if* $subj(\mathtt{n}) = \emptyset$ *then* $\delta = \delta_\emptyset$*). Node* $\mathtt{n}$ *is* satisfiable *if* $\delta_\emptyset$ *is not logically equivalent to* `false` *and*

$$\delta_n(\vec{x}) \supset \bigwedge_{j \in J} (\exists x_{\mathtt{p}j}.(\delta_j(x_{\mathtt{p}j}) + R(\mathtt{n}, \mathtt{p}, j)) \wedge (\vec{x} \leq x_{\mathtt{p}j}))$$

Definition E.8 (Satisfiable time graph) *The time graph of* G *is satisfiable if*

1. *all nodes* $\mathtt{n}$ *that are not of the form* $(\overline{\mathtt{n}}, i, !)$ *are satisfiable, and*
2. *for all* $(\overline{\mathtt{n}}, i, !)$ *(with* $\mathtt{term}_G(\overline{\mathtt{n}}, j, !) = \mathtt{p} \to \mathtt{q} : \{l_i\langle S_I\rangle\{A_i\}.G_i\}_{i \in I}$*) there exists* $j \in I$ *and node* $(\overline{\mathtt{n}}, j, !)$ *in the graph tree of* G *that is satisfiable and such that* $\mathtt{end} \in G_j$.

Condition (2) ensures that, in case of branching, at least one branch is satisfiable, and that there is always at least one satisfiable path to `end`. Recall that in § 3 we assume there is always a path to `end`, here we state that this path is allowed by the time constraints. Note that j can be equal to i. Also note that if a branch i cannot be taken then the node associated with the receive/branching $(\overline{\mathtt{n}}, i, ?)$ is trivially satisfiable as its node constraint is false.

Step 5 : checking for infinite satisfiability of nodes in recursion bodies Although in the paper we assume infinite satisfiability, we give a hints on how to algorithmically check infinite satisfiability (see Definition B.7) on a tree graph.

We consider all paths $p = \mathtt{nodes}(G'), \ldots, \mathbf{t}$ such that $\mu\mathbf{t}.G' \in G$. Let $p = \mathtt{n}, p'$ be one of such paths and $\mathtt{term}_G(\mathtt{n}) = G'$, then either of the following conditions must hold:

1. for all $\mathtt{n}' \in p$, $\mathtt{resInfo}(\mathtt{n}') = \emptyset$ and $\neg\mathcal{B}(\mathtt{const}(\mathtt{n}'))$,
2. for all $\mathtt{p} \in \mathcal{P}(G')$ there exists $\mathtt{n}' \in p$ such that $subj(\mathtt{n}') = \mathtt{p}$ and $\mathtt{resInfo}(\mathtt{n}') \neq \emptyset$

Recall $\mathcal{B}(\delta)$ is from Definition B.6.

Summary Definition E.9 summarises the steps performed by the checker.

Definition E.9 (Feasibility checker) *To check for feasibility of a G: (1-2) generates the time graph for the all-unfolding of G, (3) annotates the nodes of the graph with their dependency constraints and resets, (4) checks that the time graph is satisfiable, and (5) checks that all recursions in G are infinitely satisfiable. We write* $\vdash_F G$ *if G satisfies the checker.*

Wait-freedom. Wait-freedom can be checked following a similar approach as for Feasibility (steps 1,2,3,4,5) but substituting Definition E.7 (satisfiable node) with Definition E.10 (wait-free node).

Definition E.10 (Wait-free node) *A node* $\mathtt{n}$ *such that* $subj(\mathtt{n}) = \emptyset$ *or* $fn(\mathtt{const}(\mathtt{n})) = \emptyset$ *is always wait-free. Let* $\mathtt{n}_! = (\overline{\mathtt{n}}, i, !)$ *and* $\mathtt{n}_? = (\overline{\mathtt{n}}, i, ?)$ *be nodes of the (all-unfolding) time graph of G with* $subj(\mathtt{n}_?) = \mathtt{p}$, $\mathtt{const}(\mathtt{n}_?) = \delta$. *If* $fn(\delta) \neq \emptyset$ *then we let* $\{\delta_\emptyset\} \cup \{\delta_j\}_{j \in J}$ *be the components of* δ. *We say that* $\mathtt{n}_?$ *is* wait-free

$$\delta_{\mathtt{n}_?}(\vec{x}) \bigwedge_{j \in J} ((\delta_j(x_{\mathtt{p}j}) + R(\mathtt{n}_?, \mathtt{p}, j)) \supset (\vec{x} \leq x_{\mathtt{p}j}))$$

E.4 Auxiliary lemmas (properties of clock constraints in recursive *TG*s)

Let $G = \mu \mathbf{t}.G'$ be a recursive *TG*. We denote with G_i the $i-$th all-unfolding of G. The main lemma of this section is Lemma E.14, showing that if the constraint in all nodes in G_i are satisfiable then also those of G_{i+1} are, assuming G infinitely satisfiable. Lemma E.14 relies on two auxiliary results: (1) Lemma E.11 states that satisfiability of a node is preserved when a recursion has no reset and no bad constraints, (2) Lemma E.12 will be used for loops where each participant resets at each cycle.

Let $\vec{x} = x_1, \ldots, x_n$ and $\vec{x}' = x'_1, \ldots, x'_n$ be vectors of clocks having the same number n of clocks. We denote $(x_1 \leq x'_1) \wedge \ldots \wedge (x_n \leq x'_n)$ by $\vec{x} \leq \vec{x}'$. When relating a vector with a single clock x we denote $(x_1 \leq x) \wedge \ldots \wedge (x_n \leq x)$ by $\vec{x} \leq x$.

Lemma E.11 *Assume* $\neg\mathcal{B}(\delta_\mathtt{n})$ *and* $\neg\mathcal{B}(\delta)$ *(Definition B.6). Then*

$$\delta_\mathtt{n}(\vec{x}) \supset \exists x.\delta(x) \wedge \vec{x} \leq x \tag{46}$$

implies

$$\delta_\mathtt{n} \wedge \delta_\mathtt{n}[\vec{x}'/\vec{x}] \wedge \vec{x} \leq \vec{x}' \supset \exists x.\delta(x) \wedge \vec{x}\vec{x}' \leq x \tag{47}$$

Proof. Since $\vec{x}$ are universally quantified in (46) then time can be shifted forward in (47) while preserving the satisfiability of δ. The fact that time can be shifted forward preserving satisfiability is straightforward by induction on the structure of clock constraints under the assumption that $\neg\mathcal{B}(\delta_\mathtt{n})$ and $\neg\mathcal{B}(\delta)$.

Lemma E.12 *Assume* $\neg\mathcal{B}(\delta_\mathtt{n})$ *and* $\neg\mathcal{B}(\delta)$. *Then*

$$\delta_\mathtt{n}(\vec{x}) \supset \exists x.\delta(x) \wedge \vec{x} \leq x$$

implies

$$\delta_\mathtt{n}(\vec{x}) \supset \exists x.\delta(x) + R \wedge \vec{x} \leq x$$

Proof. Intuitively, the solution of $\delta + R$ is still greater to or equal than $\vec{x}$ since time is shifted forward. Again we can proceed mechanically by induction on the structure of clock constraints.

Lemma E.13 *If* $\mathtt{n} <_G \mathtt{n}'$ *then for all* $\mathtt{p} \in \mathcal{P}(G)$, $R(\mathtt{n},\mathtt{p}) \leq R(\mathtt{n},\mathtt{p})$.

Lemma E.14 *Let* $G = \mu t.G'$ *be a TG, and assume all the nodes of G are infinitely satisfiable. For all* $i \in \mathbb{N}$, *if* $\vdash^F G^i$ *then* $\vdash^F G^{i+1}$.

Proof By Remark E.2 we assume, without loss of generality, that p owns at most one clock. Namely we simplify

$$\delta_n(\vec{x}) \supset \delta_0 \wedge \bigwedge_{j \in J} (\exists x_{\mathtt{p}j}.(\delta_j(x_{\mathtt{p}j}) + R(\mathtt{n},\mathtt{p},j)) \wedge (\vec{x} \leq x_{\mathtt{p}j}))$$

with

$$\delta_n(\vec{x}) \supset \exists x.(\delta(x) + R(\mathtt{n},\mathtt{p},1)) \wedge (\vec{x} \leq x))$$

Let $\mathtt{n}_i$ be a node in the i-th all-unfolding of G, $\delta_{\mathtt{n}_i}(\vec{x})$ be its dependency constrains and $\delta = \mathtt{const}(\mathtt{n})$. If $fn(\delta) = \emptyset$ the thesis follows immediately by the fact that the validity of δ is not influenced by the state or previous constraints. We therefore assume $fn(\delta) = \{x\}$. Let $\mathtt{n}_{i+1}$ be the node that corresponds to $\mathtt{n}_i$ in the $i+1$-th all-unfolding of G, and observe that $\mathtt{const}(\mathtt{n}_{i+1}) = \mathtt{const}(\mathtt{n}) = \delta$. We distinguish two main cases: (1) there are no resets in the loop and there are no bad constraints, and (2) all participants reset in G'. If (1) then $\vdash^F G$

$$\delta_{\mathtt{n}_i}(\vec{x}) \supset \exists x.\delta(x) \wedge \vec{x} \leq x \tag{48}$$

By Lemma E.11 on (48) we obtain

$$\delta_{\mathtt{n}_i} \wedge \delta_{\mathtt{n}_i}[\vec{x}'/\vec{x}] \wedge \vec{x} \leq \vec{x}' \supset \exists x.\delta(x) \wedge \vec{x}\vec{x}' \leq x \tag{49}$$

By (49) and observing that $\delta_{\mathtt{n}_{i+1}} = \delta_{\mathtt{n}_i} \wedge \delta_{\mathtt{n}_i}[\vec{x}'/\vec{x}] \wedge \vec{x} \leq \vec{x}'$ we obtain that δ is satisfiable in the corresponding node of the $i+1$-th all-unfolding.

If (2) since there are resets then for every node $\mathtt{n}^i$ in the i-th occurrence of the recursion by unfolding, and $\mathtt{const}(\mathtt{n}^i)$ is satisfiable then, as the same scenario will repeat identically in loop $i+1$, $\mathtt{const}(\mathtt{n}^{i+1})$ is also satisfiable. We first observe $\delta_{\mathtt{n}_{i+1}} = \delta_{\mathtt{n}_i} \wedge \delta'$ (δ is the logical conjunction of all the single constraints $\delta_{\mathtt{n}_i}$ with, added, the corresponding resets). Also observe that (as standard for logical conjunction)

$$\delta_{\mathtt{n}_i}(\vec{x}) \wedge \delta'(\vec{x}') \wedge \vec{x} \leq \vec{x}' \quad \supset \quad \delta_{\mathtt{n}_i} \tag{50}$$

Let $\mathtt{p} = subj(\mathtt{n}_i)$, by satisfiability of node $\mathtt{n}_i$

$$\delta_{\mathtt{n}_i}(\vec{x}) \supset \exists \delta(x) + R(\mathtt{n}_i,\mathtt{p}) \wedge \vec{x} \leq x \tag{51}$$

By Lemma E.12 on (51)

$$\delta_{\mathtt{n}_i}(\vec{x}) \supset \exists \delta(x) + R(\mathtt{n}_i,\mathtt{p}) + R' \wedge \vec{x} \leq x \tag{52}$$

We set $R' = R(\mathtt{n}_{i+1},\mathtt{p}) - R(\mathtt{n}_i,\mathtt{p})$ (note that $R' \geq 0$ as R is non decreasing by Lemma (E.13). The result follows by (50), (52) and transitivity of the logical implication observing that $\delta_{\mathtt{n}_{i+1}} = \delta_{\mathtt{n}_i}(\vec{x}) \wedge \delta'(\vec{x}') \wedge \vec{x} \leq \vec{x}'$.

E.5 Auxiliary lemmas (feasibility)

Definition E.15 will be convenient, in the proofs, to make it clear what is the global time elapsed since the beginning of the protocol.

Definition E.15 (Annotated executions) *The annotated execution from G_0 to G_n is defined as $(\nu_0, G_0) \xrightarrow{\ell_1,t_1} (\nu_1, G_1) \xrightarrow{\ell_2,t_2} \dots \xrightarrow{\ell_n,t_n} (\nu_n, G_n)$ such that:*

1. $\nu_0(x) = 0\ \forall x \in dom(\nu_0)$,
2. $(\nu_i, G_i) \xrightarrow{\ell_i,t_i} (\nu_{i+1}, G_{i+1})$ *if* $(\nu_i, G_i) \xrightarrow{\ell_i} (\nu_{i+1}, G_{i+1})$ *by the transitions in Figure 1,*
3. $t_0 = 0$, $t_i = t_{i-1} + t$ *if* $\ell_i = t$ *for some* $t \in \mathbb{R}^{\geq 0}$, *and* $t_i = t_{i-1}$ *otherwise.*

Lemma E.16 *Let G_0 be a TG such that $\vdash^F G_0$, $(\nu_0, G_0) \longrightarrow^* (\nu, G)$ and $G \neq \mathtt{end}$. If $\nu \models^* \mathtt{rdy}(G)$ then (using the annotated executions from Definition E.15)*

1. $(\nu, G) \longrightarrow^* \xrightarrow{(\ell,\ t_i)} (\nu', G')$ *and*
2. $\nu' \models^* \mathtt{rdy}(G')$.

Proof. By Remark E.2, without loss of generality we assume that $\mathtt{p}$ owns at most one clock.

Let $\mathtt{n}_i$ be a node in the i-th all-unfolding of G, $\delta_{\mathtt{n}_i}(\vec{x})$ be its dependency constrains and $\delta = \mathtt{const}(\mathtt{n})$. If $fn(\delta) = \emptyset$ the thesis follows immediately by the fact that the validity of δ is not influenced by the state or previous constraints. We therefore assume $fn(\delta) = \{x\}$. Let $\mathtt{n}_{i+1}$ be the node that corresponds to $\mathtt{n}_i$ in the $i+1$-th all-unfolding of G, and observe that $\mathtt{const}(\mathtt{n}_{i+1}) = \mathtt{const}(\mathtt{n}) = \delta$.

We proceed by case analysis on the structure of G.

Case $G = \mathtt{p} \to \mathtt{q} : \{l_j\langle S_j\rangle\{A_j\}.G_j\}_{j\in J}$ with $A_j = \{\delta_{\mathtt{0}j}, \lambda_{\mathtt{0}j}, \delta_{\mathtt{I}j}, \lambda_{\mathtt{I}j}\}$. Since $\nu \models^* \mathtt{rdy}(G)$ then there exist $t \geq 0$ and $k \in J$ such that $\nu + t \models \delta_{\mathtt{0}k}$ hence

$$(\nu, G) \xrightarrow{(t,\ t_i - t)} \xrightarrow{(\mathtt{pq}!l_k\langle S_k\rangle,\ t_i)} (\nu', G')$$

with $G' = \mathtt{p} \rightsquigarrow \mathtt{q} : l_k\langle S_k\rangle\{A_k\}.G_k$ and $\nu' = [\lambda_{\mathtt{0}k} \mapsto 0]\nu + t$. Observe that

$$\mathtt{rdy}(\mathtt{p} \rightsquigarrow \mathtt{q} : l_k\langle S_k\rangle\{A_k\}.G_k) = \mathtt{rdy}(G) \cup \delta_{\mathtt{I}k} \cup \{\delta_i\}_{i\in I} \setminus \{\delta_{\mathtt{0}j}\}_{j\in J}$$

where $\{\delta_i\}_{i\in I}$ is the set of clock constraints associated to the next action (if any) of $\mathtt{p}$. The constraints in $\mathtt{rdy}(G)$ remain satisfiable in ν' by definition of $\nu \models^* \mathtt{rdy}(G)$. We have to prove, instead, that $\delta_{\mathtt{I}k}$ and $\{\delta_i\}_{i\in I}$ are satisfiable after some non-negative t' in ν'. As to $\{\delta_i\}_{i\in I}$, if $\mathtt{p} \notin \mathcal{P}(G')$, namely there is no edge $(\mathtt{nodes}(G'), \mathtt{n})$ in the time graph of G_0 such that $subj(\mathtt{n}) = \mathtt{p}$, then there is no next ready action for $\mathtt{p}$ and no δ_i is added to $\mathtt{rdy}(G')$ (i.e., $I = \emptyset$) hence done. If $\mathtt{p} \in \mathcal{P}(G')$ then there exists a next node $\mathtt{n}$ such that $subj(\mathtt{n}) = \mathtt{p}$ and $\mathtt{const}(\mathtt{n}) = \delta(x_{\mathtt{p}})$. By $\vdash^F G_0$ we have

$$\delta_{\mathtt{n}}(\vec{x}) \supset \exists x_{\mathtt{p}}.(\delta(x_{\mathtt{p}}) + R(\mathtt{n}, \mathtt{p})) \wedge (\vec{x} \leq x_{\mathtt{p}}) \tag{53}$$

with $R(\mathtt{n}, \mathtt{p})$ being clocks of the predicates in $\delta_{\mathtt{n}}(\vec{x})$ associated to actions in which $\mathtt{p}$ has reset.

By (53) we know that all steps caused by actions that causally/temporally precede the last action to (ν', G') precede some of the solutions of (53) for $x_{\mathtt{p}}$. Therefore by (53) there exists a solution $t_{\mathtt{p}}$ for (53). We have two cases:

1. $t_{\mathtt{p}} - t_i < 0$. In this case the delays introduced by all causally/temporally preceding actions (i.e., those actions whose states are nodes in the path to G' in the time graph of G_0) have been included in $\delta_{\mathtt{n}}$. Hence if time elapsed after the solutions of (53) (hence $t_{\mathtt{p}} - t_i < 0$) it is because of some steps that are not causally/temporally related with the last action to (ν', G'). In this case timed has elapsed by $\lfloor \textsc{time} \rfloor$, by the premise of rule $\lfloor \textsc{time} \rfloor$ (i.e., $\delta(x_{\mathtt{p}})$ is satisfiable after that time step) we obtain the thesis.
2. $t_{\mathtt{p}} - t_i \geq 0$. In this case observe that by (53) $t_{\mathtt{p}} - R(\mathtt{n},\mathtt{p})$ is a solution of $\delta(x_{\mathtt{p}})$. Since $R(\mathtt{n},\mathtt{p})$ is the lower bound of the resets occurred for $\mathtt{p}$ then

$$\nu'(x_{\mathtt{p}}) \leq t_i - R(\mathtt{n},\mathtt{p}) \tag{54}$$

   By hypothesis for this case $t_{\mathtt{p}} \geq t_i$ which applied to (54) yields

$$\nu'(x_{\mathtt{p}}) \leq t_{\mathtt{p}} - R(\mathtt{n},\mathtt{p})$$

   that is, the solution $t_{\mathtt{p}} - R(\mathtt{n},\mathtt{p})$ of δ does not precede the current clock assignment $\nu'(x_{\mathtt{p}})$ for $\mathtt{p}$. Hence δ is satisfiable after some non-negative delay in ν'.

The case for $\delta_{\mathtt{I}k}(x_{\mathtt{q}})$ and those for branching process are similar.
The case for $G = \mu \mathbf{t}.G'$ follows by Lemma E.14 under assumption $\vdash^F G_0$.

Lemma E.17 *Let G_0 be infinitely satisfiable and $\vdash^F G_0$ then $(\nu_0, G_0) \longrightarrow^* (\nu, G)$ implies $(\nu, G) \longrightarrow (\nu', \mathtt{end})$.*

Proof We reason by induction on the complexity of G that can be calculated with the following function:

$$\begin{array}{l}
\mathtt{complexity}(\mathtt{p} \to \mathtt{q} : \{l_j : \{A_j\}.G_j\}_{j \in J}) = 1 + max(\mathtt{complexity}(G_j)) \\
\mathtt{complexity}(\mathtt{p} \rightsquigarrow \mathtt{q} : \{l_j : \langle A_j \rangle \{G_j\}.j \in J\}_{)} = 0.5 + max(\mathtt{complexity}(G_j)) \\
\mathtt{complexity}(\mu \mathbf{t}.G') = \mathtt{complexity}(G') \\
\mathtt{complexity}(\mathbf{t}) = \mathtt{complexity}(\mathtt{end}) = 0
\end{array}$$

We observe that (1) the only steps that do not decrease the complexity are $\lfloor \textsc{time} \rfloor$, $\lfloor \textsc{end} \rfloor$ and $\lfloor \textsc{rec} \rfloor$, (2) by Lemma E.16 after some $\lfloor \textsc{time} \rfloor$ steps a visible step is always possible, which decrease the complexity. Since the complexity is always non negative, some step $\lfloor \textsc{end} \rfloor$ or $\lfloor \textsc{rec} \rfloor$ will eventually occur. If $\lfloor \textsc{end} \rfloor$ occurs then we are done. If $\lfloor \textsc{rec} \rfloor$ occurs then fix one of the annotated executions of G_0 and consider the following set of pairs of states of that execution:

$$\begin{array}{rl}
R_{set} = \{(\nu_i, G_i), (\nu_{i+1}, G_{i+1}) \mid (\nu_i, G_i) \xrightarrow{(\ell,t)} (\nu_{i+1}, G_{i+1}) \text{ and} \\
\qquad\qquad \mathtt{complexity}(G_i) < \mathtt{complexity}(G_{i+1})\}
\end{array}$$

By assumption on recursive timed global types in § 3 there is at least one $\mathtt{end}$ in each recursion. By infinite satisfiability of G (straightforward by infinite satisfiability of G_0) and by condition (2) of Definition E.8 in each iteration there is a reachable path to $\mathtt{end}$ satisfying the constraints. By Remark E.3 this path to $\mathtt{end}$ will eventually be taken, hence R_{set} is finite. Let $(\nu_i, G_i), (\nu_{i+1}, G_{i+1})$ be the pair of states that occur later than all the other couples of states in R_{set}. From (ν_i, G_i) and (ν_{i+1}, G_{i+1}), since the complexity is decreasing and $\lfloor \textsc{rec} \rfloor$ will no longer occur, we have that $\lfloor \textsc{end} \rfloor$ will eventually occur.

E.6 Auxiliary lemmas (wait-freedom)

In this section we rely on the annotated traces in Definition E.15.

Lemma E.18 *If* $(\nu_0, G_0) \longrightarrow^* \xrightarrow{(\ell,\, t)} (\nu, G)$ *then* $\delta_{\mathtt{n}}(\vec{x}) \supset \vec{x} \leq t$.

The proof of Lemma E.18 is mechanical by induction on the transitions. It states that the set of solutions of a node constraint includes the solutions of all possible executions until the state corresponding to that node, hence it includes also the one with larger delays (if any) which is certainly greater than or equal to the delay t of the execution we are taking into account.

Lemma E.19 *Let G_0 be a TG such that $\vdash^W G_0$ and, using annotated executions in Definition E.15, $(\nu_0, G_0) \longrightarrow^* (\nu', G') \xrightarrow{(\mathtt{pq}!\ell\langle S\rangle,\, t)} (\nu, G)$ with*

- $\mathtt{p} \to \mathtt{q} : \{l_j\langle S_j\rangle\{\delta_{\mathtt{0}j}, \lambda_{\mathtt{0}j}, \delta_{\mathtt{I}j}, \lambda_{\mathtt{I}j}\}.G_j\}_{j\in I} \in G'$ *and*
- $\mathtt{p} \rightsquigarrow \mathtt{q} : l_i\langle S_i\rangle\{\delta_{\mathtt{0}i}, \lambda_{\mathtt{0}i}, \delta_{\mathtt{I}i}, \lambda_{\mathtt{I}i}\}.G'_i \in G \qquad (i \in I)$.

Then $\delta_{\mathtt{I}i}(\vec{x}) \supset \nu(\vec{x}) \leq \vec{x}$.

Proof. By Remark E.2, without loss of generality we assume that $\delta_{\mathtt{I}i}$ has at most one clock. The statement $\delta_{\mathtt{I}i}(\vec{x}) \supset \nu(\vec{x}) \leq \vec{x}$ becomes:

$$\delta_{\mathtt{I}i}(x) \supset \nu(x) \leq x$$

If $fn(\delta_{\mathtt{I}}) = \emptyset$ the thesis follows immediately. We proceed assuming $fn(\delta_{\mathtt{I}i}) = \{x\}$. The constraint $\delta_{\mathtt{0}i}$ is satisfied by the clock assignment ν' since an output step $\mathtt{pq}!\ell\langle S\rangle$ is made from (ν', G'). Let $\mathtt{n}'$ is the node in the time graph of G_0 corresponding to the receive of branch i in G, by $\vdash^W G_0$ we have

$$\delta_{\mathtt{n}'}(\vec{x}) \wedge (\delta_{\mathtt{I}i}(x) + R(\mathtt{n}', \mathtt{q})) \supset (\vec{x} \leq x) \tag{55}$$

By Lemma E.18 on (55) we have

$$(\delta_{\mathtt{I}i}(x) + R) \supset (t \leq x) \tag{56}$$

for $R = t - \nu(x)$ (the reset is difference between the absolute time from the beginning t and the current clock assignment). By Definition E.4 (56) is equivalent to

$$\delta_{\mathtt{I}i}(x) \supset \nu(x) \leq x$$

as required.

E.7 Proof of Proposition 5.1

Proposition 5.1 follows immediately from Lemma E.17 and Lemma E.19.

F Time Progress - definitions and proofs

This appendix includes: (§ F.1) auxiliary definitions and lemmas for time progress, and (§ F.2) the proof of Theorem 5.4 (time progress).

F.1 Auxiliary definitions and lemmas

Definition F.1 (Time erasure) *The erasure* $\texttt{erase}(P)$ *of time from a process P is defined inductively as follows:*

$$
\begin{array}{ll}
\texttt{erase}(\overline{u}[\texttt{n}](y).P) & = \overline{u}[\texttt{n}](y).\texttt{erase}(P) \\
\texttt{erase}(u[\texttt{i}](y).P) & = u[\texttt{i}](c).\texttt{erase}(P) \\
\texttt{erase}(c[\texttt{p}] \triangleleft l\langle e\rangle; P) & = c[\texttt{p}] \triangleleft l\langle e\rangle; \texttt{erase}(P) \\
\texttt{erase}(c[\texttt{p}] \triangleright \{l_i(z_i).P_i\}_{i\in I}) & = c[\texttt{p}] \triangleright \{l_i(z_i).\texttt{erase}(P_i)\}_{i\in I} \\
\texttt{erase}(c[\texttt{p}] \triangleleft \langle\langle c'\rangle\rangle; P) & = c[\texttt{p}] \triangleleft \langle\langle c'\rangle\rangle; \texttt{erase}(P) \\
\texttt{erase}(c[\texttt{p}] \triangleright ((y)).P) & = c[\texttt{p}] \triangleright ((y)).\texttt{erase}(P) \\
\texttt{erase}(\texttt{delay}(t).P) & = \texttt{erase}(P) \\
\texttt{erase}(\texttt{if } e \texttt{ then } P \texttt{ else } Q) & = \texttt{if } e \texttt{ then } \texttt{erase}(P) \texttt{ else } \texttt{erase}(Q) \\
\texttt{erase}(P \mid Q) & = \texttt{erase}(P) \mid \texttt{erase}(Q) \\
\texttt{erase}(\mathbf{0}) & = \mathbf{0} \\
\texttt{erase}((\nu n)P) & = (\nu n)\texttt{erase}(P) \\
\texttt{erase}(\mu X.P) & = \mu X.\texttt{erase}(P) \\
\texttt{erase}(X) & = X \\
\texttt{erase}(s:h) & = s:h
\end{array}
$$

Definition F.2 (Deferrability) *A process P is* deferrable *iff one of the following holds:*

$$
\begin{array}{l}
P = \texttt{delay}(t).P' \text{ and } t > 0 \\
P = \texttt{delay}(t).P',\ t = 0 \text{ and } P' \text{ is deferrable} \\
P = Q_1 \mid Q_2 \text{ and both } Q_1 \text{ and } Q_2 \text{ are deferrable} \\
P \in \{\mathbf{0}, \boldsymbol{t}\} \\
P = s:h \\
P = (\nu n)P' \text{ and } P' \text{ is deferrable} \\
P = \mu X.P' \text{ and } P' \text{ is deferrable}
\end{array}
$$

Intuitively, a process is *non deferrable* if no action ready to execute in any of its threads is a time action. We say that P is *non deferrable* if P is not deferrable.

Lemma F.3 *If* $\texttt{erase}(P) \longrightarrow$ *then* $\texttt{erase}(P)$ *is of one of the following forms:*

1. $\overline{a}[\texttt{n}](y).P_1 \mid \prod_{i\in\{2,\ldots,n\}} a[\texttt{i}](y).P_i$
2. $s[\texttt{p}][\texttt{q}] \triangleleft l\langle e\rangle; P \mid s:h$
3. $s[\texttt{p}][\texttt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \mid s:(\texttt{p},\texttt{q},l_j\langle S_j\rangle)\cdot h \quad (j\in I)$
4. $s[\texttt{p}][\texttt{q}] \triangleleft \langle\langle s'[\texttt{p}']\rangle\rangle; P \mid s:h$
5. $s[\texttt{p}][\texttt{q}] \triangleright ((z)).P \mid s:(\texttt{p},\texttt{q},s'[\texttt{p}'])\cdot h$
6. $\texttt{if } e \texttt{ then } P \texttt{ else } Q$
7. $P \mid Q$ *with* P *and* Q *non deferrable*
8. $(\nu n)P$

Lemma F.4 1. *if* $\texttt{erase}(P) = \overline{a}[\texttt{n}](y).P_1 \mid \prod_{i\in\{2,\ldots,n\}} a[\texttt{i}](y).P_i$ *then* $\exists P'_1,\ldots,P'_n, t_1 \geq 0,\ldots,t_n \geq 0$ *s.t.* $P \equiv \texttt{delay}(t_1).\overline{a}[\texttt{n}](y).P'_1 \mid \prod_{i\in\{2,\ldots,n\}} \texttt{delay}(t_i).a[\texttt{i}](y).P'_i$ *and* $P_i = \texttt{erase}(P'_i)$ *for all* $i \in 1,\ldots,n$.
2. *if* $\texttt{erase}(P) = s[\texttt{p}][\texttt{q}] \triangleleft l\langle e\rangle; P_1 \mid s:h$ *then* $\exists P'_1, t \geq 0$ *s.t.* $P \equiv \texttt{delay}(t).s[\texttt{p}][\texttt{q}] \triangleleft l\langle e\rangle; P'_1 \mid s:h$ *and* $P_1 = \texttt{erase}(P'_1)$.

3. *if* $\texttt{erase}(P) = s[\mathtt{p}][\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \mid s : (\mathtt{p},\mathtt{q},l_j\langle v\rangle)\cdot h \quad (j \in I)$ *then* $\exists P'_i, t \geq 0$ *for* $i \in I$ *s.t.* $P \equiv \texttt{delay}(t).s[\mathtt{p}][\mathtt{q}] \triangleright \{l_i(z_i).P'_i\}_{i\in I} \mid s : (\mathtt{p},\mathtt{q},l_j\langle v\rangle)$ *and* $P_i = \texttt{erase}(P'_i)$ *for all* $i \in I$.
4. *if* $\texttt{erase}(P) = s[\mathtt{p}][\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P_1 \mid s : h$ *then* $\exists P'_1, t \geq 0$ *s.t.* $P \equiv \texttt{delay}(t).s[\mathtt{p}][\mathtt{q}] \triangleleft \langle\langle s'[\mathtt{p}']\rangle\rangle; P'_1 \mid s : h$ *and* $P_1 = \texttt{erase}(P'_1)$.
5. *if* $\texttt{erase}(P) = s[\mathtt{p}][\mathtt{q}] \triangleright ((z)).P_1 \mid s : (\mathtt{p},\mathtt{q},s'[\mathtt{p}'])\cdot h$ *then* $\exists P'_1, t \geq 0$ *s.t.* $P \equiv \texttt{delay}(t).s[\mathtt{p}][\mathtt{q}] \triangleright ((z)).P'_1 \mid s : (\mathtt{p},\mathtt{q},s'[\mathtt{p}'])\cdot h$ *and* $P_1 = \texttt{erase}(P'_1)$.
6. *if* $\texttt{erase}(P) = \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2$ *then* $\exists P'_1, P'_2, t \geq 0$ *s.t.* $P \equiv \texttt{delay}(t).\texttt{if } e \texttt{ then } P'_1 \texttt{ else } P'_2$ *and* $P_i = \texttt{erase}(P'_i)$ *for* $i \in \{1,2\}$.
7. *if* $\texttt{erase}(P) = P_1 \mid P_2$ *then* $\exists P'_1, P'_2, t_1 \geq 0, t_2 \geq 0$ *s.t.* $P \equiv \texttt{delay}(t_1).P'_1 \mid \texttt{delay}(t_2).P'_2$ *with* $P_i \equiv \texttt{erase}(P'_i)$ *and* P'_i *non deferrable for* $i \in \{1,2\}$.
8. *if* $\texttt{erase}(P) = (\nu n)P'$ *then* $\exists P'', t_{\geq} 0$ *s.t.* $P \equiv \texttt{delay}(t).(\nu)P''$ *and* $\texttt{erase}(P'') = P'$.

In case (7) of Lemma F.4 observe that we can always find non deferrable (Definition F.2) continuations P'_1 and P'_2 as we can collect the delays in front of the process by structural equivalence rule $\texttt{delay}(t).\texttt{delay}(t').P_i \equiv \texttt{delay}(t+t').P_i$.

Lemma F.5 *If P is non deferrable and* $\texttt{erase}(P) \longrightarrow P'$ *then* $P \longrightarrow P''$ *and* $\texttt{erase}(P'') = P'$

Proof. Mechanical, by induction on the derivation of $\texttt{erase}(P)$, proceeding by case analysis using Lemma F.3 and Lemma F.4.

Lemma F.6 (Time progress - parallelism and interleaved sessions) *Let Γ be a* feasible *and* wait free *mapping, $\Gamma \vdash P_0 \triangleright \emptyset$, and $P_0 \longrightarrow^+ P$ where P is session delay and*

$$P \equiv \texttt{delay}(t).P \mid R \qquad \textit{for some } t > 0 \textit{ and non deferrable } R$$

If $\texttt{erase}(P)$ *is not a deadlock process and* $\texttt{erase}(P) \longrightarrow$ *then* $R \longrightarrow$.

Proof. Without loss of generality we assume that for some L

$$R = R' \mid \prod_{l\in L} s_l : h_l$$

We proceed by induction on the complexity of R', proceeding by case analysis on the possible form of process R'.

The complex cases are those for receive and session receive, (i.e., R' is stuck waiting for messages). In all other cases one can immediately show that R' reduces. In fact,

- R' cannot be a of the form $\overline{u}[\mathtt{n}](y).R''$ or $u[\mathtt{i}](y).R''$ since (1) $\texttt{erase}(P)$ is deadlock free hence all the other corresponding session requests or accepts to start session y are in P, and (2) P is session delay hence none of the corresponding session accepts/delay can be in P_i.
- R' cannot be a delay process as P it is non deferrable (Definition F.2) by hypothesis,
- if R' is of the form $s[\mathtt{p}][\mathtt{q}] \triangleleft l\langle e\rangle; R''$ or $s[\mathtt{p}][\mathtt{q}] \triangleleft \langle\langle c'\rangle\rangle; R''$ then one can immediately show that R can reduce by $\lfloor$SEL$\rfloor$ or $\lfloor$SREC$\rfloor$.
- in the inductive cases (e.g., for hiding and parallel processes) the result follows by induction.

We will show the case for R' being an input process. The case for session receive is similar hence omitted.

Case $R' \equiv s[\mathtt{p}][\mathtt{q}] \triangleright \{l_k(z_k).R_k\}_{k\in I}$. In this case:

$$R \equiv s[\mathtt{p}][\mathtt{q}] \triangleright \{l_k(z_k).R_k\}_{k\in I} \mid \prod_{l\in L} s_l : h_l$$

We proceed by contradiction assuming Γ feasible and wait-free, $\mathtt{erase}(P)$ deadlock free, $\mathtt{erase}(P) \longrightarrow$ and $R \not\longrightarrow$. If $R \not\longrightarrow$ we have the following cases depending on where the corresponding output of $s[\mathtt{p}][\mathtt{q}] \triangleright \{l_k(z_k).R_k\}_{k\in I}$ is:

1. the corresponding output appears (immediately or after some other input) in some continuation R_k,
2. the corresponding output appears in P_i for some $i \in I$
3. the corresponding output does not appear.

If (1) or (3) then we contradict that $\mathtt{erase}(P)$ is deadlock free. If (2) then we contradict the fact P_0 was validated against a feasible and wait-free Γ in fact: feasibility ensures that there exists at least one execution of the types of sessions of s_i where a selection/delegate happens before or at the same time of the corresponding receive/session receive, and by wait-freedom since in one execution the output does not precede the input then in all correct implementations the output does not precede the input, hence the output for a non deferrable (Definition F.2) input does not occur after $t_i > 0$ time which contradicts fact that the output appears in P_i.

F.2 Proof of Theorem 5.4 (time progress)

We prove Theorem 5.4 by induction on the proof of P, proceeding by case analysis on the possible forms of P according to Lemma F.3.

Case Lemma F.3 (1): Since we assume that P is session delay then this case can only occur if $P = \mathtt{erase}(P)$, hence the thesis follows immediately by the hypothesis.

Cases Lemma F.3 (2), (3), (4), and (5) proceed similarly. We show the case for Lemma F.3 (3): $\mathtt{erase}(P) = s[\mathtt{p}][\mathtt{q}] \triangleright \{l_i(z_i).P_i\}_{i\in I} \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle S_j\rangle) \cdot h \quad (j \in I)$. By Lemma F.4 (3)

$$P \equiv \mathtt{delay}(t).\mathtt{erase}(P)[P_i'/P_i] \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle S_j\rangle) \cdot h \quad (j \in I) \tag{57}$$

By two reductions by rule $\lfloor$ STR $\rfloor$ with premises by rules $\lfloor$ TIME $\rfloor$ and then $\lfloor$ BRA $\rfloor$ on (57)

$$P \longrightarrow \mathtt{erase}(P)[P_i'/P_i] \mid s : (\mathtt{p}, \mathtt{q}, l_j\langle S_j\rangle) \cdot h \longrightarrow P_j \mid s : h \tag{58}$$

and by reduction rule $\lfloor$ BRA $\rfloor$ on $\mathtt{erase}(P)$

$$\mathtt{erase}(P) \longrightarrow \mathtt{erase}(P_j) \mid s : h \tag{59}$$

Finally, by (58) and (59), since

$$\mathtt{erase}(P_j) \mid s : h = \mathtt{erase}(P_j \mid s : h)$$

we have the thesis for this case.

Case Lemma F.3 (6): immediate considering that $e \downarrow \mathtt{true}$ in both P and $\mathtt{erase}(P)$ (the erasure does not affect the conditions of conditional statements as such conditions do not involve delays).

Case Lemma F.3 (7): $\texttt{erase}(P) = P_1 \mid P_2$ and $\texttt{erase}(P) \longrightarrow P_1' \mid P_2$. By reduction rule $\lfloor$Com$\rfloor$

$$P_1 \longrightarrow P_1' \tag{60}$$

By Lemma F.4 (7)

$$P \equiv \texttt{delay}(t_1).Q_1 \mid \texttt{delay}(t_2).Q_2$$
$$\texttt{erase}(Q_i) = P_i,\ i \in \{1,2\} \tag{61}$$

and

$$Q_i \textit{ non deferrable},\ i \in \{1,2\} \tag{62}$$

If $t_1 = t_2$ then $\texttt{delay}(t_1).Q_1 \mid \texttt{delay}(t_2).Q_2 \longrightarrow Q_1 \mid Q_2$ by a $\lfloor$Time$\rfloor$ reduction. By (61), (62) and (60) $Q_1 \longrightarrow$ hence by $\lfloor$Com$\rfloor$

$$Q_1 \mid Q_2 \longrightarrow Q_1' \mid Q_2$$

hence done.

If $t_1 \neq t_2$, assume without loss of generality that $t_1 > t_2$. After a $\lfloor$Time$\rfloor$ reduction

$$\texttt{delay}(t_1).Q_1 \mid \texttt{delay}(t_2).Q_2 \longrightarrow \texttt{delay}(t_1 - t_2).Q_1 \mid Q_2$$

with Q_2 *non deferrable*. This case holds by Lemma F.6

Case Lemma F.3 (8): immediate by induction.

G CTAs - encoding, characterisation and properties

This appendix includes: (§ G.1) the LTS for CTAs, (§ G.2) the encoding from timed (global/local) types to CTAs, (§ G.3) the formal definition of multiparty compatibility for CTAs, (§ G.4) auxiliary definitions and lemmas on the encoding, and (§ G.6) the proof of Theorem 5.7 (progress).

G.1 Labelled transitions for CTAs

For the reader's convenience we report the definition of LTS for CTAs from [14] (we ignore the ε actions in [14] as they are not present in CTA encoded from *TGs*).

The labelled transitions for CTA*s* is defined on labels $\mathcal{L}$ (i.e., the labels used by timed types). The semantics of $(\mathcal{A}_1, \ldots, \mathcal{A}_n, \vec{w})$, following [14], is a LTS with states of the form $((q_1, \nu_1), \ldots, (q_n, \nu_n), \vec{w})$. The initial state is $((q_0^1, \nu_0^1), \ldots, (q_0^n, \nu_0^n), \vec{\varepsilon})$ where q_0^i is the initial state of $\mathcal{A}_i$.

Let $s = ((q_1, \nu_1), \ldots, (q_n, \nu_n), \vec{w})$ and $s' = ((q_1', \nu_1'), \ldots, (q_n', \nu_n'), \vec{w}')$, the transitions are defined as follows:

- $s \xrightarrow{\texttt{pq}!l\langle S\rangle} s'$ if $(q_\texttt{p}, q_\texttt{p}', \texttt{pq}!l\langle S\rangle, \lambda, \delta) \in E$, and (i) $w_\texttt{pq} = l\langle S\rangle \cdot w_\texttt{pq}'$, $\delta[\nu_\texttt{p}] \downarrow \texttt{true}$, $\nu_\texttt{p}' = \nu_\texttt{p}[\lambda \mapsto 0]$, (ii) $q_\texttt{r} = q_\texttt{r}'$, $\nu_\texttt{r} = \nu_\texttt{r}'$ for all $\texttt{r} \neq \texttt{p} \in \{1,..,n\}$, and $w_\texttt{rk} = w_\texttt{rk}'$ for all $\texttt{rk} \neq \texttt{pq}$.
- $s \xrightarrow{\texttt{pq}?l\langle S\rangle} s'$ if $(q_\texttt{q}, q_\texttt{q}', \texttt{pq}?l\langle S\rangle, \lambda, \delta) \in E$, and (i) $w_\texttt{qp} \cdot l\langle S\rangle = w_\texttt{pq}'$, $\delta[\nu_\texttt{q}] \downarrow \texttt{true}$, $\nu_\texttt{q}' = \nu_\texttt{q}[\lambda \mapsto 0]$, (ii) $q_\texttt{r} = q_\texttt{r}'$, $\nu_\texttt{r} = \nu_\texttt{r}'$ for all $\texttt{r} \neq \texttt{q} \in \{1,..,n\}$, and $w_\texttt{rk} = w_\texttt{rk}'$ for all $\texttt{rk} \neq \texttt{pq}$.
- $s \xrightarrow{t} s'$ if $\nu_\texttt{p}' = \nu_\texttt{p} + t$, $q_\texttt{p} = q_\texttt{p}'$ and $w_\texttt{pq} = w_\texttt{pq}'$ for all $\texttt{p},\texttt{q} \neq \texttt{p} \in \{1,..,n\}$.

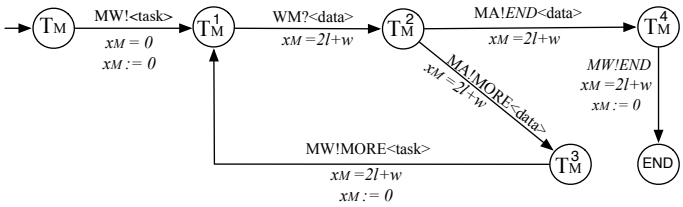
G.2 The encoding

We first define an encoding from single projections of *TG* into timed automata. The encoding of T_0 into a timed automaton, written $\mathcal{A}(T_0)$, is $(Q, T_0, I \times \mathcal{S}, \{x_{\mathtt{p}}\}, E, F)$ where: $Q = \{T' \mid T' \in T_0, T' \neq \mathbf{t}, T' \neq \mu\mathbf{t}.T\}$; $q_0 = T_0$ with $T_0 = \mu\mathbf{t}.T'$ and $T' \in Q$; E is defined as follows, for all $T \in Q$:

$$\text{If } T = \mathtt{q} \oplus \{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I} \text{ then}$$
$$\begin{cases} (T, T_i, (\mathtt{pq}!l_i\langle S_i\rangle), \lambda_i, \delta_i) \in E \;\; T_i \neq \mathbf{t} \\ (T, T', (\mathtt{pq}!l_i\langle S_i\rangle), \lambda_i, \delta_i) \in E \;\; T_i = \mathbf{t},\ \mu\mathbf{t}\,\vec{\mathbf{t}}.T' \in T_0, T' \in Q \end{cases}$$
$$\text{If } T = \mathtt{q}\&\{l_i : \langle S_i \rangle \{\delta_i, \lambda_i\}.T_i\}_{i \in I} Q \text{ then}$$
$$\begin{cases} (T, T_i, (\mathtt{pq}?l_i\langle S_i\rangle), \lambda_i, \delta_i) \in E \;\; T_i \neq \mathbf{t} \\ (T, T', (\mathtt{pq}?l_i\langle S_i\rangle), \lambda_i, \delta_i) \in E \;\; T_i = \mathbf{t},\ \mu\mathbf{t}\,\vec{\mathbf{t}}.T' \in T_0, T' \in Q \end{cases}$$

and the set of final states is $F = \{\mathtt{end}\}$. The definition says that the set of states Q are the subterms of branching, selection or end in T_0; the initial state q_0 is the occurrence of (the recursion body of) T_0; the channels correspond to those in T_0; and the transition is defined from the state T to continuation T_i with the action $\mathtt{pq}!l_i\langle S_i\rangle$ for the output and $\mathtt{pq}?l_i\langle S_i\rangle$ for the input and with the clock constraint and reset predicate of that branch. If T_i is a recursive type variable $\mathbf{t}$, it points the state of the body of the corresponding recursive type.

The figure below illustrates timed automaton $\mathcal{A}_{\mathtt{M}}$ that is the encoding of the master role of the three party protocol for distributed computation where the set of final states is $F = \{\mathtt{end}\}$.



The *encoding of a set of local timed types* $\{T_i\}_{i \in I}$ into a network of CTAs, written $\mathcal{A}(\{T_i\}_{i \in I})$, is the tuple $(\mathcal{A}(T_1), \ldots, \mathcal{A}(T_n), \vec{\varepsilon})$.

G.3 Multiparty compatibile CTAs

We extend the definition of multiparty compatibility for communicating automata given in [11, Definition 4.2] to CTAs.

We say that a state s is *n-bounded reachable* if it is reachable from the initial state s_0 and if in each intermediary state from s_0 to s the size of each buffer contains no more than n messages.

We let transitions in E to range over $e, e', \ldots$, and given a transition $e = (q, q', \ell, \lambda, \delta)$ we denote label ℓ by $act(e)$. An alternation ϕ is a (possibly empty) sequence of transitions such that their labels are an alternation of sending and corresponding receive actions (i.e., actions of the form $\mathtt{pq}!l\langle S\rangle$ immediately followed by $\mathtt{pq}?l\langle S\rangle$).

Definition G.1 (Multiparty Compatibility) *A CTA $\mathcal{C} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, \vec{\varepsilon})$ ($n \geq 2$) is multiparty compatible if for any 1-bounded reachable state s of $\mathcal{C}$:*

1. *for any output action $ij!l\langle S\rangle$ from s in $\mathcal{A}_i$ ($i \in \{1,..,n\}$) there exists a sequence of transitions $\phi \cdot e$ from s in $\mathcal{C}$ such that ϕ is an alternation, $i \notin act(\phi)$, and $act(e) = ij?l\langle S\rangle$; and*
2. *for any input action $ji?l\langle S\rangle$ from s in $\mathcal{A}_i$ ($i \in \{1,..,n\}$) there exists an input action $ji?l'\langle S'\rangle$ and a sequence of transitions $\phi \cdot e$ from s in $\mathcal{C}$ such that ϕ is an alternation, $i \notin act(\phi)$, and $act(e) = ij!l\langle S\rangle$*

Remark G.1. Multiparty compatibility allows scenarios with unbounded channels e.g., in $G = \mu \mathbf{t}.\mathtt{p} \to \mathtt{q} : l\langle S\rangle\{A\}.\mathbf{t}$, $\mathtt{p}$ can send $\mathtt{q}$ an unlimited number of messages before $\mathtt{q}$ receives, hence the channel of $\mathtt{q}$ is unbounded. By multiple application of $\lfloor$Rec$\rfloor$ and $\lfloor$Select$\rfloor$ (and $\lfloor$Async1$\rfloor$ after the first action) we obtain the following execution:

$$\mu \mathbf{t}.\mathtt{p} \to \mathtt{q} : l\langle S\rangle\{A\}.\mathbf{t} \xrightarrow{\mathtt{pq}!l\langle S\rangle} \mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S\rangle\{A\}.G \xrightarrow{\mathtt{pq}!l\langle S\rangle} \mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S\rangle\{A\}.\mathtt{p} \rightsquigarrow \mathtt{q} : l\langle S\rangle\{A\}.G \xrightarrow{\mathtt{pq}!l\langle S\rangle} \ldots$$

Remark G.2. Considering 1-bounded executions in Definition G.1 (which leads to a simpler theory) preserves generality due to a property called *stability* in [11] and directly applicable to our scenario. By stability, if $\mathcal{C}$ is basic and multiparty compatible, then for all its reachable states s there exists an execution $s \longrightarrow^* s'$ from s to a stable state s', and there exists a 1-bounded execution $s_0 \longrightarrow^* s'$ from the initial state s_0 of $\mathcal{C}$ to s'. Namely, after an appropriate execution *any* reachable state can be reached by a 1-bounded execution.

G.4 Auxiliary definitions and lemmas (sound and complete encoding)

A Communicating Automaton (CA) [8] (adapting the syntax of [8] to our scenario for convenience) is defined as $M = (Q, q_0, \mathcal{A}ct, E, F)$ where Q, q_0, $\mathcal{A}ct$, E and F are as for timed automata but there are no clocks and reset predicates. A system of CAs is a tuple $S = (M_1, \ldots, M_n, \vec{w})$ with $\vec{w}$ unbound unidirectional channels. We denote by $\mathtt{erase}(\mathcal{A})$ the CFSM obtained from $\mathcal{A}$ by removing clocks and reset predicates. We can denote by $\mathtt{erase}(\mathcal{C})$ the system of CFSMs obtained by pointwise erasure of the timed automata in $\mathcal{C}$ (i.e., $(\mathtt{erase}(\mathcal{A}_1), \ldots, \mathtt{erase}(\mathcal{A}_n), \vec{w})$). The basic property and multiparty compatibility for CAs is defined exactly as for CTAs; the fact that these are properties on the the structure of the CA and CTA (and do not involve temporal aspects) directly yields Lemma G.2.

Lemma G.2 *$\mathcal{C}$ is basic and multiparty compatible if and only if $\mathtt{erase}(\mathcal{C})$ is basic and multiparty compatible.*

Following a similar argument, we can state the following:

Lemma G.3 *G is projectable if and only if $\mathtt{erase}(G)$ is projectable.*

Let $\mathcal{A}(\_)^{-time}$ denote the encoding from (untimed) local types and systems of CAs from [**?**] (Definition 3.5 in [**?**]). The definition of the encoding in § G, based on the encoding in [**?**] except that it adds to the resulting automata the corresponding annotations with clock constraints and reset predicates, yields Lemma G.4.

Lemma G.4 $\texttt{erase}(\mathcal{A}(G)) = \mathcal{A}(\texttt{erase}(G))^{-time}$.

Lemma G.5 *Let G be a (projectable) TG then $\mathcal{A}(G)$ is basic and multiparty compatible.*

Proof. The projectability of G (by hypothesis) together with Lemma G.3 yields that $\texttt{erase}(G)$ is projectable, hence we can encode it into a system of CAs $\mathcal{A}(\texttt{erase}(G))^{-time}$. $\mathcal{A}(\texttt{erase}(G))^{-time}$ is basic and multiparty compatible by soundness of the encoding $\mathcal{A}(\_)^{-time}$ in [**?**]. By Lemma G.4 $\mathcal{A}(\texttt{erase}(G))^{-time} = \texttt{erase}(\mathcal{A}(G))$, which yields that $\texttt{erase}(\mathcal{A}(G))$ is basic and multiparty compatible. The results follows by Lemma G.2 using the fact that $\texttt{erase}(\mathcal{A}(G))$ is basic multiparty compatible.

Lemma G.6 *Let T be a TL, then $\mathcal{A}(T) \approx T$ (with specified semantics).*

Proof. The proof by induction on the encoding. We first consider $T \precsim \mathcal{A}(T)$.

If $T = \texttt{end}$ then both T and $\mathcal{A}(T)$ produce an empty set of traces.

If $T = \mathtt{p} \oplus \{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I}$ then we either have a send action or a time action. In case of send $(\nu, T) \xrightarrow{\mathtt{pq}!l_k\langle S_k \rangle} (\nu', T')$ by $\lfloor \text{LSEL} \rfloor$. Hence $T = \mathtt{p} \oplus \{l_i : \langle S_i \rangle \{B_i\}.T_i\}_{i \in I}$ and $k \in I$, $T' = T_k$. Let $B_k = \{\delta, \lambda\}$; by the definition of encoding

$$(T, T_k, (\mathtt{pq}!l_k\langle S_k \rangle), \lambda, \delta) \in E \tag{63}$$

with E being the set of edges of $\mathcal{A}(T)$. We observe that by $\lfloor \text{LSEL} \rfloor$ $\nu \models \delta$ hence (63) can take place and the clock is updated exactly as ν' (i.e., $\nu' = [\lambda \mapsto 0]\nu$). The thesis follows by induction (observing that the next state may be a step back to the beginning of the a recursion, which is mimicked by the automaton). In case of a time action,

$$(\nu, T) \xrightarrow{t} (\nu + t, T)$$

and the encoding can make a corresponding action as the clock constraint correspond precisely to the constraint of the ready action of T.

If T is a receive type we proceed as in the case for send.

If T is of the form $\mu \mathbf{t}.T'$ the thesis directly follows by induction. If T is $\mathbf{t}$ then no step can be made. Note that $\mathbf{t}$ is never reached even if in the starting *TL* it is enclosed in recursion definition $\mu \mathbf{t}.T'$. In this case, in order to make a step, an unfolding must occur which bring us back to the case $\mu \mathbf{t}.T'$ and the corresponding state in the timed automaton.

We next consider $\mathcal{A}(T) \precsim T$, which is delicate as timed automata can always make time actions whereas timed local types only make time actions that preserve satisfiability of the ready clock constraints (hence our assumption of a specified semantics for timed automata). We write $q \sim T$ if q is the initial state of the timed automaton generated from T (i.e., $\mathcal{A}(T)$) or from some unfolding (i.e., $\mathcal{A}(T[\mu \mathbf{t}.T/\mathbf{t}])$ for some $\mathbf{t}$).

We proceed by case analysis on the label ℓ. We show the case for $\ell = \mathtt{pq}?l_k\langle S_k\rangle$ and the case for $\ell = t'$ (the case $\ell = \mathtt{pq}!l_k\langle S_k\rangle$ proceeds similarly and is omitted).

If $\ell = \mathtt{pq}?l_k\langle S_k\rangle$ then

$$(q, q', \mathtt{pq}?l_k\langle S_k\rangle, \lambda_k, \delta_k) \in E \tag{64}$$

with E set of transitions of $\mathcal{A}(T)$.

Assume $(q,\nu) \xrightarrow{\mathtt{pq}?l_k\langle S_k\rangle} (q',\nu')$ for some ν, ν'. By LTS

$$\nu \models \delta_k \tag{65}$$

$$\nu' = [\lambda_k \mapsto 0]\nu \tag{66}$$

By definition of encoding

$$T = \mu\vec{\mathbf{t}}.\mathrm{p}\&\{l_i : \langle S_i\rangle\{\delta_i,\lambda_i\}.T_i\}_{i\in I}$$

with $k \in I$. We proceed by induction on the step rule, showing the base case (the inductive case by rule $\lfloor$LRec$\rfloor$ is immediate). In the base case, $\vec{\mathbf{t}}$ is an empty vector and the step is for rule $\lfloor$LBra$\rfloor$. By (65) and rule $\lfloor$LBra$\rfloor$ (LTS for *TLs*)

$$(\nu, T) \xrightarrow{l_k\langle S_k\rangle} (\nu', T_k) \tag{67}$$

If $T_k \neq \mathbf{t}$ then immediately by definition of encoding $q' \sim T_k$. Note that T_k cannot be $\mathbf{t}$ because of the unfolding made by rule $\lfloor$LRec$\rfloor$. More precisely, in this case T_k is equal to $T[\mu\mathbf{t}.T/\mathbf{t}]$ for some T, hence $q' \sim T_k$.

If $\ell = t$ then since we assume a specified semantics, then:

$$\nu + t \models \mathtt{rdy}(T)$$

hence

$$(\nu, T) \xrightarrow{t} (\nu + t, T)$$

where all $q_i \sim T'$ since $q_i = q_i'$ and $T_i = T_i'$.

Lemma G.7 *If $\mathcal{C}$ is a session CTA then there exists G such that $\mathcal{C} \approx G$.*

Proof. By Proposition 3.2 in [**?**], if M is a basic and multiparty compatible CA then there exists an untimed global type G such that $M \approx G$. This results is proven by giving a terminating algorithm that builds a synthesis that returns, given basic and multiparty compatible CAs a global type. We use the synthesis algorithm to create a skeleton of *TG*. We observe that there is a one-to-one correspondence between the states of $\mathtt{erase}(\mathcal{C})$ and those of the synthesised untimed global type, hence we can annotate each transition with the corresponding clock constraint and reset predicate. Note that by determinism, there is always one transition from a given a state and label. The proof proceeds similarly to Theorem 3.3 where by relying on the structural correspondence between G and $\mathcal{C}$ from [**?**] it is immediate to show that clocks evolve in a similar way and clock constraints allow the same timings.

G.5 Proof of Theorem 5.6 (sound and complete encoding)

Let $\{T_i\}_{i\in\mathcal{P}(G)}$ be the projections of G. By definition (§ 5)

$$\mathcal{A}(G) = (\{\mathcal{A}(T_i)\}_{i\in\mathcal{P}(G)}, \vec{\varepsilon}) \tag{68}$$

By Lemma G.6 for all $i \in \mathcal{P}(G)$

$$\mathcal{A}(T_i) \approx T_i \tag{69}$$

By (68) and (69) we obtain:

$$\mathcal{A}(G) \approx (\{T_i\}_{i\in\mathcal{P}(G)}, \vec{\varepsilon}) \tag{70}$$

Theorem 3.3 and (70) give

$$\mathcal{A}(G) \approx G \tag{71}$$

Clause (1) follows by Lemma G.5 ($\mathcal{A}(G)$ is basic and multiparty compatible) and (71). Clause (2) follows by Lemma G.7.

G.6 Proof of Theorem 5.7 (progress)

By feasibility of G if $(\nu, G) \to^* (\nu', G')$ implies $(\nu', G') \to^* (\nu'', \mathtt{end})$. By Theorem 5.6 $\mathcal{A}(G) \approx G$ hence since G are deterministic (i.e., $(\nu, G) \xrightarrow{\ell} (\nu', G')$ and $(\nu, G) \xrightarrow{\ell} (\nu'', G'')$ imply $(\nu', G') = (\nu'', G'')$) then $\mathcal{A}(G)$ always reaches a final state (liveness). Since it is always possible to proceed to a final state it never occurs that $\mathcal{C}$ remains stuck in a state because of an input action (with no output counter-part) and it is always possible to let time diverge (in final states `end` time can diverge), which yields progress.