Fearless Asynchronous Communications with Timed Multiparty Session Protocols

Ping Hou
University of Oxford, UK

Nicolas Lagaillardie
Imperial College London, UK

Nobuko Yoshida
University of Oxford, UK

Abstract

Session types using *affinity* and *exception handling* mechanisms have been developed to ensure the communication safety of protocols implemented in concurrent and distributed programming languages. Nevertheless, current affine session types are inadequate for specifying real-world asynchronous protocols, as they are usually imposed by *time constraints* which enable *timeout exceptions* to prevent indefinite blocking while awaiting valid messages. This paper proposes the first formal integration of *affinity*, *time constraints*, *timeouts*, and *time-failure handling* based on multiparty session types for supporting reliability in asynchronous distributed systems. With this theory, we statically guarantee that asynchronous timed communication is deadlock-free, communication safe, while being *fearless* – never hindered by timeout errors or abrupt terminations.

To implement our theory, we introduce `MultiCrusty`^{T}, a Rust toolchain designed to facilitate the implementation of safe affine timed protocols. `MultiCrusty`^{T} leverages generic types and the `time` library to handle timed communications, integrated with optional types for affinity. We evaluate `MultiCrusty`^{T} by extending diverse examples from the literature to incorporate time and timeouts. We also showcase the *correctness by construction* of our approach by implementing various real-world use cases, including protocols from the Internet of Remote Things domain and real-time systems.

2012 ACM Subject Classification Software and its engineering → Software usability; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi

Keywords and phrases Session Types, Concurrency, Time Failure Handling, Affinity, Timeout, Rust

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.29

Related Version *Full Version*: https://arxiv.org/abs/2406.19541

Supplementary Material *Software (Source Code)*: https://github.com/NicolasLagaillardie/mpst_rust_github

Funding Work supported by: EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

1 Introduction

Background The growing prevalence of distributed programming has emphasised the significance of prioritising *reliability* in distributed systems. Dedicated research efforts focus on enhancing reliability through the study and modelling of failures. This research enables the design of more resilient distributed systems, capable of effectively handling failures and ensuring reliable operation.

A lightweight, type-based methodology, which ensures basic reliability – *safety* in distributed communication systems, is *session types* [16]. This type discipline is further advanced by *Multiparty Session Types* (MPST) [17, 18], which enable the specification and verification of communication protocols among multiple message-passing processes in concurrent and distributed systems. MPST ensure that protocols are designed to prevent common safety errors, i.e. deadlocks and communication mismatches during interactions among many participants [17, 18, 37]. By adhering to a specified MPST protocol, participants (a.k.a. end-point programs) can communicate reliably and efficiently. From a practical perspective, MPST have been implemented in various programming languages [5, 26, 28, 32, 40, 43], facilitating their applications and providing safety guarantees in real-world programs.

Nevertheless, tackling the challenges of unreliability and failures remains a significant issue for session types. Most session type systems operate under the assumption of flawless and reliable communication without failures. To address this limitation, recent works [31, 14, 15, 28] have developed *affine session types* by incorporating the *affinity* mechanism that explicitly accounts for and handles unreliability and failures within session type systems. Unlike linear types that must be used *exactly* once, affine types can be used *at most* once, enabling the safe dropping of subsequent types and the premature termination of a session in the presence of protocol execution errors.

In most real-life devices and platforms, communications are predominantly asynchronous: inner tasks and message transfers may take time. When dealing with such communications, it becomes crucial to incorporate *time constraints* and implement *timeout failure handling* for each operation. This is necessary to avoid potential blockages where a process might wait indefinitely for a message from a non-failed process. While various works, as explained later, address time conditions and timeouts in session types, it is surprising that none of the mentioned works on affine session types tackles timeout failures during protocol execution.

This Paper introduces a new framework, *affine timed multiparty session types* (ATMP), to address the challenges of timeouts, disconnections and other failures in asynchronous communications: (1) We propose ATMP, an extension of asynchronous MPST that incorporates time specifications, affinity, and mechanisms for handling exceptions, thus facilitating effective management of failures, with a particular focus on timeouts. Additionally, we demonstrate that properties from MPST, i.e. *type safety*, *protocol conformance*, and *deadlock-freedom*, are guaranteed for well-typed processes, even in the presence of timeouts and their corresponding handling mechanism; (2) We present `MultiCrusty`^{T}, our RUST toolchain designed for building asynchronous timed multiparty protocols under ATMP: `MultiCrusty`^{T} enables the implementation of protocols adhering to the properties of ATMP.

The primary focus of ATMP lies in effectively *handling* timeouts during process execution, in contrast to the approaches in [4, 3], which aim to completely *avoid* time failures. Bocchi et al. [4] introduce time conditions in MPST to ensure precise timing in communication protocols, while their subsequent work [3] extends *binary* timed session types to incorporate timeouts, allowing for more robust handling of time constraints. Yet, they adopt strict requirements to *prevent* timeouts. In [4], *feasibility* and *wait-freedom* are required in their protocol design. Feasibility requires precise time specifications for protocol termination, while wait-freedom prohibits overlapping time windows for senders and receivers in a protocol, which is not practical in real-world applications. Similarly, in [3], strong conditions including *progress* of an entire set of processes and *urgent receive* are imposed. The progress property is usually *undecidable*, and the urgent receive condition, which demands immediate message reception upon availability, is infeasible with asynchronous communication.

Recently, [30] proposes the inclusion of timeout as the unique failure notation in MPST, offering flexibility in handling failures. Time also plays a role in *synchronous* communication systems, where [22] develops *rate*-based *binary* session types, ensuring *synchronous* message exchanges at the same *rate*, i.e. within the same time window. However, in both [30] and [22], time constraints are not integrated into types and static type checking, resulting in the specifications lacking the ability to guide time behaviour. Additionally, the model used in [22] assumes that all communications and computations are non-time-consuming, i.e. with zero time cost, making it unfeasible in distributed systems.

By the efficient integration of time and failure handling mechanisms in our framework, none of those impractical requirements outlined in [4, 3] is necessary. In ATMP, when a process encounters a timeout error, a mechanism for handling time failures is triggered, notifying all participants about the timeout, leading to the termination of those participants and ultimately ending the session. Such an approach guarantees that participants consistently reach the *end of the protocol*, as the communication session is entirely dropped upon encountering a timeout error. As a result, every process can terminate successfully, reducing the risk of indefinite blockages, even with timeouts. Additionally, in our system, time constraints over local clocks are incorporated with types to effectively model asynchronous timed communication, addressing the limitations in [30, 22].

Except for [22], the aforementioned works on timed session types focus more on theory, lacking implementations. To bridge this gap on the practical side, we provide `MultiCrusty`^{T}, a Rust implementation of ATMP designed for secure timed communications. `MultiCrusty`^{T} makes use of affine timed meshed channels, a communication data structure that integrates time constraints and clock utilisation. Our toolchain relies on macros and native generic types to ensure that asynchronous protocols are inherently *correct by construction*. In particular, `MultiCrusty`^{T} performs compile-time verification to guarantee that, at any given point in the protocol, each isolated pair of participants comprises one sender and one receiver with corresponding time constraints. Additionally, we employ affine asynchronous primitives and native optional types to effectively handle *runtime* timeouts and errors.

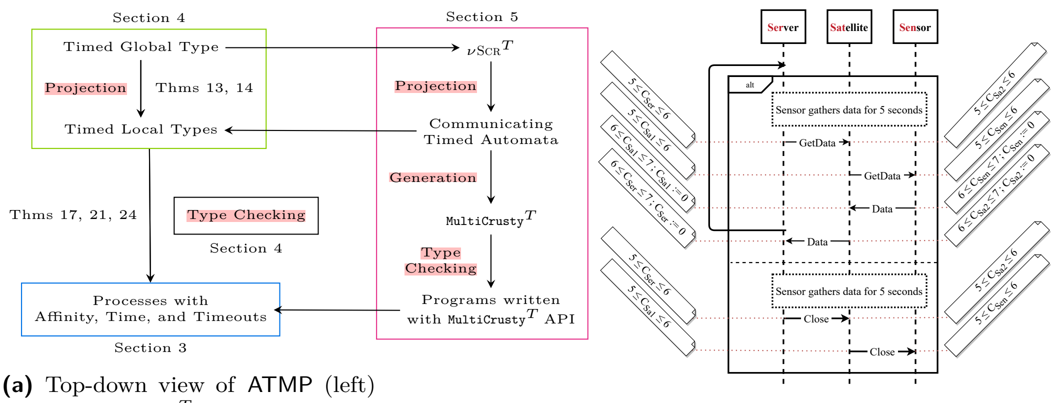
To showcase the capabilities and expressiveness of our toolchain, we evaluate `MultiCrusty`^{T} through examples from the literature, and further case studies including a remote data protocol from an Internet of Remote Things (IoRT) network [7], a servo web protocol from a web engine [38], and protocols from real-time systems such as Android motion sensor [2], PineTime smartwatch [35], and keyless entry [41]. Our comparative analysis with a Rust implementation of affine MPST without time [28] reveals that `MultiCrusty`^{T} exhibits minimal overhead while providing significantly strengthened property checks.

Structure **§ 2** offers a comprehensive overview of our theory and toolchain. **§ 3** provides a session π-calculus for ATMP that incorporates timeout, affinity, asynchrony, and failure handling mechanisms. **§ 4** introduces an extended theory of asynchronous multiparty session types with time annotations. Additionally, we present a typing system for ATMP session π-calculus, and demonstrate the properties of typed processes. **§ 5** delves into the design and usage of `MultiCrusty`^{T}, our Rust implementation of ATMP. **§ 6** showcases the compilation and execution benchmarks of `MultiCrusty`^{T}, based on selected case studies. **§ 7** concludes the paper by discussing related work, and offering conclusions and potential future work. Full proofs, auxiliary material, and more details of `MultiCrusty`^{T} can be found in the full version of the paper [19]. Our toolchain and evaluation examples are available in an artifact.

2 Overview

In this section, we give an overview of affine timed multiparty session types (ATMP) and `MultiCrusty`^{T}, our toolchain for implementing affine timed asynchronous protocols. First, we share a real-world example inspiring our work on affine asynchronous timed communication.

Fig. 1b depicts our running example, *remote data*. This real-world scenario is sourced from



(a) Top-down view of ATMP (left) and MultiCrusty^T (right).

(b) Global protocol for *remote data*.

Figure 1 Overview of affine asynchronous communication with time.

a satellite-enabled Internet of Remote Things network [7], and describes data transmissions among a *Sensor* (Sen), a *Server* (Ser), and a *Satellite* (Sat): Ser aims to periodically retrieve data gathered by Sen via Sat. The protocol revolves around a loop initiated by Ser, which faces a decision: either retrieve data or end the protocol. In the former scenario, Ser requests data retrieval from Sen with a message labelled *GetData* via Sat within the time window of 5 and 6 time units, as indicated by clock constraints (i.e. $5 \le C_{\texttt{Ser}} \le 6$, where $C_{\texttt{Ser}}$ is the clock associated with Ser). Upon receiving this request, Sen responds by sending the data with a message labelled *Data* to Ser through Sat within 6 and 7 time units, followed by clock resets denoted as reset predicates (i.e. $C_{\texttt{Ser}} := 0$, resetting the clock to 0). In the alternative branch, Ser sends a *Close* message to Sat, which is then forwarded to Sen, between 5 and 6 time units.

Our remote data protocol includes internal tasks that consume time, notably Sen requiring 5 time units to gather data before transmitting. In cases where our protocol lacks a specified timing strategy (i.e. no time requirements), and Sen cannot accomplish the data-gathering tasks, it results in indefinite blocking for Sat and Ser as they await the data. This could lead to undesirable outcomes, including partially processed data, data corruption, or incomplete transmission of processed data. Therefore, incorporating time constraints into communication protocols is imperative, as it better reflects real-world scenarios and ensures practical viability.

2.1 ATMP: Theory Overview

Our ATMP theory follows the *top-down* methodology [17, 18], enhancing asynchronous MPST with time features to facilitate timed global and local types. As shown in Fig. 1a (left), we specify multiparty protocols with time as *timed global types*. These timed global types are projected into *timed local types*, which are then used for type-checking processes with affine types, time, timeouts, and failure handling, written in a session calculus. As an example, we consider a simple communication scenario derived from remote data: the Satellite (Sat) communicates with the server (Ser) by sending a *Data* message (Data). Specifically, Sat needs to send the message between 6 and 7 time units and reset its clock afterwards, while Ser is expected to receive the message within the same time window and reset its clock accordingly.

Timed Types and Processes This communication behaviour can be represented by the timed global type G:

$$\texttt{Sat}\to\texttt{Ser}\colon \{\texttt{Data}\{6 \le C_{\texttt{Sat}} \le 7, C_{\texttt{Sat}} := 0, 6 \le C_{\texttt{Ser}} \le 7, C_{\texttt{Ser}} := 0\}.\mathbf{end}\}$$

where $C_{\texttt{Sat}}$ and $C_{\texttt{Ser}}$ denote the clocks of Sat and Ser, respectively. A global type represents a protocol specification involving multiple roles from a global standpoint.

Adhering to the MPST top-down approach, a timed global type is then *projected* onto timed local types, which describe communications from the perspective of individual roles. In our example, G is projected onto two timed local types, one for each role Sat and Ser:

$$T_{\texttt{Sat}} = \texttt{Ser}\oplus\texttt{Data}\{6 \leq C_{\texttt{Sat}} \leq 7, C_{\texttt{Sat}} := 0\}.\textbf{end} \quad T_{\texttt{Ser}} = \texttt{Sat}\&\texttt{Data}\{6 \leq C_{\texttt{Ser}} \leq 7, C_{\texttt{Ser}} := 0\}.\textbf{end}$$

Here $T_{\texttt{Sat}}$ indicates that Sat sends ($\oplus$) the message Data to Ser between 6 and 7 time units and then immediately resets its clock $C_{\texttt{Sat}}$. Dually, $T_{\texttt{Ser}}$ denotes Ser receiving ($\&$) the message from Sat within the same time frame and resetting its clock $C_{\texttt{Ser}}$.

In the final step of the top-down approach, we employ timed local types to conduct type-checking for processes, denoted as P_i, in the ATMP session calculus. Our session calculus extends the framework for *affine multiparty session types* (AMPST) [28] by incorporating processes that model time, timeouts, and asynchrony. In our example, $T_{\texttt{Sat}}$ and $T_{\texttt{Ser}}$ are used for the type-checking of $s[\texttt{Sat}]$ and $s[\texttt{Ser}]$, which respectively represent the channels (a.k.a. session endpoints) played by roles Sat and Ser in a multiparty session s, within the processes:

$$P_{\texttt{Sat}} = \textbf{delay}(C_1 = 6.5) \,.\, s[\texttt{Sat}]^{0.4}[\texttt{Ser}]\oplus\texttt{Data}.\mathbf{0} \quad P_{\texttt{Ser}} = \textbf{delay}(C_2 = 6) \,.\, s[\texttt{Ser}]^{0.3}[\texttt{Sat}]\texttt{Data}.\mathbf{0}$$

The Satellite process $P_{\texttt{Sat}}$ waits for exactly 6.5 time units ($\textbf{delay}(C_1 = 6.5)$), then sends the message Data with a timeout of 0.4 time units ($s[\texttt{Sat}]^{0.4}[\texttt{Ser}]\oplus\texttt{Data}$), and becomes inactive ($\mathbf{0}$). Meanwhile, the Server process $P_{\texttt{Ser}}$ waits for 6 time units ($\textbf{delay}(C_2 = 6)$), then receives the message with a timeout of 0.3 time units ($s[\texttt{Ser}]^{0.3}[\texttt{Sat}]\texttt{Data}$), subsequently becoming inactive.

Solution to Stuck Processes Due to Time Failures It appears that the parallel execution of $P_{\texttt{Sat}}$ and $P_{\texttt{Ser}}$, $P_{\texttt{Sat}} \mid P_{\texttt{Ser}}$, cannot proceed further due to the disparity in timing requirements. Specifically, using the same session s, Sat sends the message Data to Ser between 6.5 and 6.9 time units, while Ser must receive it from Sat between 6 and 6.3 time units. This results in a stuck situation, as Ser cannot meet the required timing condition to receive the message.

Fortunately, in our system, timeout failures are allowed, which can be addressed by leveraging affine session types and their associated failure handling mechanisms. Back to our example, when $s[\texttt{Ser}]$ waits for 6 time units and cannot receive Data within 0.3 time units, a timeout failure is raised ($\texttt{timeout}[s[\texttt{Ser}]^{0.3}[\texttt{Sat}]\texttt{Data}.\mathbf{0}]$). Furthermore, we apply our time-failure handling approach to manage this timeout failure, initiating the termination of the channel $s[\texttt{Ser}]$ and triggering the cancellation process of the session s ($s\lightning$). As a result, the process will successfully terminate by canceling (or killing) all usages of s within it.

Conversely, the system introduced in [4] enforces strict requirements, including *feasibility* and *wait-freedom*, on timed global types to prevent time-related failures in well-typed processes, thus preventing them from becoming blocked due to unsolvable timing constraints. Feasibility ensures the successful termination of each allowed partial execution, while wait-freedom guarantees that receivers do not have to wait if senders follow their time constraints. In our example, we start with a timed global type that is neither feasible nor wait-free, showcasing how our system effectively handles time failures and ensures successful process termination without imposing additional conditions on timed global types. In essence, reliance on feasibility and wait-freedom becomes unnecessary in our system, thanks to the inclusion of affinity and time-failure handling mechanisms.

2.2 MultiCrusty^T: Toolchain Overview

To augment the theory, we introduce the MultiCrusty^T library, a toolchain for implementing communication protocols in Rust. MultiCrusty^T specifies protocols where communication operations must adhere to specific time limits (*timed*), allowing for *asynchronous* message reception and runtime handling of certain failures (*affine*). This library relies on two fundamental types: Send and Recv, representing message sending and receiving, respectively.

```
1  struct Send<T,
2   const CLOCK: char,
3   const START: i128,
4   const INCLUDE_START: bool,
5   const END: i128,
6   const INCLUDE_END: bool,
7   const RESET: char,
8   S>
```

(a) `Send` type

```
1  struct Recv<T,
2   const CLOCK: char,
3   const START: i128,
4   const INCLUDE_START: bool,
5   const END: i128,
6   const INCLUDE_END: bool,
7   const RESET: char,
8   S>
```

(b) `Recv` type

```
1  MeshedChannels<
2   Recv<Data,
3    'a',6,true,7,true,'a',End>,
4   Send<Data,
5    'b',6,true,7,true,'b',End>,
6   RoleSen<RoleSer<End>>,
7   NameSat,
8  >
```

(c) `MeshedChannels` type for `Sat`

Figure 2 Main types of `MultiCrusty`^{T}

Additionally, it incorporates the `End` type, signifying termination to close the connection. Figs. 2a and 2b illustrate the `Send` and `Recv` types respectively, used for sending and receiving messages of any thread-safe type (represented as `T` in Line 1). After sending or receiving a message, the next operation or continuation (`S` in Line 8) is determined, which may entail sending another message, receiving another message, or terminating the connection.

Similar to ATMP, each communication operation in `MultiCrusty`^{T} is constrained by specific time boundaries to avoid infinite waiting. These time bounds are represented by the parameters in Lines 2–7 of Fig. 2a, addressing scenarios where a role may be required to send after a certain time unit or receive between two specific time units. Consider the final communication operation in the first branch of Fig. 1b from `Sat`'s perspective. To remain consistent with § 2.1, the communication is terminated here instead of looping back to the beginning of the protocol. In this operation, `Sat` sends a message labelled `Data` to `Ser` between time units 6 and 7, with respect to its inner clock `'b'`, and then terminates after resetting its clock. This can be implemented as: `Send<Data, 'b', 6, true, 7, true, 'b', End>`.

To enable multiparty communication in `MultiCrusty`^{T}, we use the `MeshedChannels` type, inspired by [28]. This choice is necessary as `Send` and `Recv` types are primarily designed for *binary* (peer-to-peer) communication. Within `MeshedChannels`, each binary channel pairs the owner role with another, establishing a mesh of communication channels that encompasses all participants. Fig. 2c demonstrates an example of using `MeshedChannels` for `Sat` in our running example: `Sat` receives a `Data` message from `Sen` (Line 2) and forwards it to `Ser` (Line 4) before ending all communications, following the order specified by the stack in Line 6.

Creating these types manually in Rust can be challenging and error-prone, especially because they represent the local perspective of each role in the protocol. Therefore, as depicted in Fig. 1a (right), `MultiCrusty`^{T} employs a top-down methodology similar to ATMP to generate local viewpoints from a global protocol, while ensuring the *correctness* of the generated types *by construction*. To achieve this, we extend the syntax of νScr [43], a language for describing multiparty communication protocols, to include time constraints, resulting in νScr^T. A timed global protocol represented in νScr^T is then projected onto local types, which are used for generating Rust types in `MultiCrusty`^{T}.

3 Affine Timed Multiparty Session Calculus

In this section, we formalise an affine timed multiparty session π-calculus, where processes are capable of performing time actions, raising timeouts, and handling failures. We start with the formal definitions of time constraints used in the paper.

Clock Constraint, Valuation, and Reset Our time model is based on the timed automata formalism [1, 27]. Let $\mathbf{C}$ denote a finite set of *clocks*, ranging over $C, C', C_1, \ldots$, that take non-negative real values in $\mathbb{R}_{\geq 0}$. Additionally, let $t, t', t_1, \ldots$ be *time constants* ranging over $\mathbb{R}_{\geq 0}$. A *clock constraint* δ over $\mathbf{C}$ is defined as:

$$\delta \;::=\; \mathtt{true} \;\big|\; C > \mathfrak{b} \;\big|\; C = \mathfrak{b} \;\big|\; \neg\delta \;\big|\; \delta_1 \wedge \delta_2$$

where $C \in \mathbf{C}$ and $\mathfrak{b}$ is a *constant time bound* ranging over non-negative rationals $\mathbb{Q}_{\geq 0}$. We define $\mathtt{false}, <, \geq, \leq$ in the standard way. For simplicity and consistency with our implementation (§ 5), we assume each clock constraint contains a *single* clock. Extending a clock constraint with *multiple* clocks is straightforward.

A *clock valuation* $\mathbb{V} : \mathbf{C} \to \mathbb{R}_{\geq 0}$ assigns time to each clock in $\mathbf{C}$. We define $\mathbb{V} + t$ as the valuation that assigns to each $C \in \mathbf{C}$ the value $\mathbb{V}(C) + t$. The *initial* valuation that maps all clocks to 0 is denoted as $\mathbb{V}^0$, and the valuation that assigns a value of t to all clocks is denoted as $\mathbb{V}^t$. $\mathbb{V} \models \delta$ indicates that the constraint δ is satisfied by the valuation $\mathbb{V}$. Additionally, we use $\sqcup_{i \in I} \mathbb{V}_i$ to represent the overriding union of the valuations $\mathbb{V}_i$ for $i \in I$.

A *reset predicate* λ over $\mathbf{C}$ is a subset of $\mathbf{C}$ that defines the clocks to be reset. If $\lambda = \emptyset$, no reset is performed. Otherwise, the valuation for each clock $C \in \lambda$ is set to 0. For clarity, we represent a reset predicate as $C := 0$ when a single clock C needs to be reset. To denote the clock valuation identical to $\mathbb{V}$ but with the values of clocks in λ to 0, we use $\mathbb{V}[\lambda \mapsto 0]$.

Syntax of Processes Our session π-calculus for affine timed multiparty session types (ATMP) models timed processes interacting via affine meshed multiparty channels. It extends the calculus for affine multiparty session types (AMPST) [28] by incorporating asynchronous communication, time features, timeouts, and failure handling.[1]

▶ **Definition 1** (Syntax). *Let* $\mathtt{p}, \mathtt{q}, \mathtt{r}, \ldots$ *denote* roles *belonging to a (fixed) set* $\mathcal{R}$*;* $s, s', \ldots$ *for* sessions*;* $x, y, \ldots$ *for* variables*;* $\mathtt{m}, \mathtt{m}', \ldots$ *for* message labels*; and* $X, Y, \ldots$ *for* process variables*. The* affine timed multiparty session π-calculus *syntax is defined as follows:*

$$\begin{array}{rcll}
c, d & ::= & x \;\big|\; s[\mathtt{p}] & \textit{(variable, channel with role } \mathtt{p}\textit{)} \\
P, Q & ::= & \mathbf{0} \;\big|\; P \mid Q \;\big|\; (\nu s)\, P & \textit{(inaction, parallel composition, restriction)} \\
 & & c^{\mathtt{n}}[\mathtt{q}] \oplus \mathtt{m}\langle d \rangle . P & \textit{(timed selection towards role } \mathtt{q}\textit{)} \\
 & & c^{\mathtt{n}}[\mathtt{q}] \sum_{i \in I} \mathtt{m}_i(x_i) . P_i & \textit{(timed branching from role } \mathtt{q} \textit{ with } I \neq \emptyset\textit{)} \\
 & & \mathbf{def}\ D\ \mathbf{in}\ P \;\big|\; X\langle \widetilde{c} \rangle & \textit{(process definition, process call)} \\
 & & \mathbf{delay}(\delta) . P \;\big|\; \underline{\mathbf{delay}(t) . P} & \textit{(time-consuming delay, } \underline{\textit{deterministic delay}}\textit{)} \\
 & & \underline{\mathtt{timeout}[P]} \;\big|\; \mathbf{try}\ P\ \mathbf{catch}\ Q & \textit{(}\underline{\textit{timeout failure}}\textit{, try-catch)} \\
 & & \mathbf{cancel}(c) . P \;\big|\; \mathbf{cerr} \;\big|\; \underline{s\lightning} & \textit{(cancel, communication error, } \underline{\textit{kill}}\textit{)} \\
 & & s[\mathtt{p}] \blacktriangleright \sigma & \textit{(output message queue of role } \mathtt{p} \textit{ in session } s\textit{)} \\
D & ::= & X(\widetilde{x}) = P & \textit{(declaration of process variable } X\textit{)} \\
\sigma & ::= & \mathtt{q}!\mathtt{m}\langle s[\mathtt{r}] \rangle \cdot \sigma \;\big|\; \epsilon & \textit{(message queue, non-empty or empty)}
\end{array}$$

Restriction, branching, and process definitions and declarations act as binders; $\mathrm{fc}(P)$ *is the set of* free channels with roles *in* P*,* $\mathrm{fv}(P)$ *is the set of* free variables *in* P*, and* $\Pi_{i \in I} P_i$ *is the parallel composition of processes* P_i*. Extensions w.r.t.* AMPST *calculus are highlighted. Runtime processes, generated dynamically during program execution rather than explicitly written by users, are* underlined*.*

Our calculus comprises:

Channels c, d, being either variables x or channels with roles (a.k.a. session *endpoints*) $s[\mathtt{p}]$.

Standard processes as in [37, 28], including inaction $\mathbf{0}$, parallel composition $P \mid Q$, session scope restriction $(\nu s)\, P$, process definition $\mathbf{def}\ D\ \mathbf{in}\ P$, process call $X\langle \widetilde{c} \rangle$, and communication error $\mathbf{cerr}$.

Time processes that follow the program time behaviour of Fig. 2c:

[1] To simplify, our calculus exclusively emphasises communication. Standard extensions, e.g. integers, booleans, and conditionals, are routine and independent of our formulation.

- ***Timed selection*** (or ***timed internal choice***) $c^{\mathfrak{n}}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P$ indicates that a message $\mathtt{m}$ with payload d is sent to role $\mathtt{q}$ via endpoint c, whereas ***timed branching*** (or ***timed external choice***) $c^{\mathfrak{n}}[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i$ waits to receive a message $\mathtt{m}_i$ from role $\mathtt{q}$ via endpoint c and then proceeds as P_i.
  The parameter $\mathfrak{n}$ in both timed selection and branching is a *timeout* that allows modelling different types of communication primitives: *blocking with a timeout* ($\mathfrak{n}\in\mathbb{R}_{>0}$), *blocking* ($\mathfrak{n}=\infty$), or *non-blocking* ($\mathfrak{n}=0$). When $\mathfrak{n}\in\mathbb{R}_{\geq 0}$, the timed selection (or timed branching) process waits for up to $\mathfrak{n}$ time units to send (or receive) a message. If the message cannot be sent (or received) within this time, the process moves into a *timeout state*, raising a *time failure*. If $\mathfrak{n}$ is set to ∞, the timed selection (or timed branching) process blocks until a message is successfully sent (or received).
  In our system, we allow *send* processes to be time-consuming, enabling processes to wait before sending messages. Consider the remote data example shown in Fig. 1b. This practical scenario illustrates how a process might wait before sending a message, resulting in the possibility of send actions failing due to timeouts. It highlights the importance of timed selection, contrasting with systems like in [3] where send actions are instantaneous.
- $\mathbf{delay}(\delta)\,.\,P$ represents a ***time-consuming delay*** action, such as method invocation or sleep. Here, δ is a clock constraint involving a *single* clock variable C, used to specify the interval for the delay. When executing $\mathbf{delay}(\delta)\,.\,P$, any time value t that satisfies the constraint δ can be consumed. Consequently, the *runtime* ***deterministic delay*** process $\mathbf{delay}(t)\,.\,P$, arising during the execution of $\mathbf{delay}(\delta)\,.\,P$, is introduced. In $\mathbf{delay}(t)\,.\,P$, t is a constant and a solution to δ, and P is executed after a precise delay of t time units.
- $\mathtt{timeout}[P]$ signifies that the process P has violated a time constraint, resulting in a ***timeout failure***.

Failure-handling processes that adopt the AMPST approach [28]:

- **try** P **catch** Q consists of a **try** process P that is prepared to communicate with a parallel composed process, and a **catch** process Q, which becomes active in the event of a cancellation or timeout. For clarity, **try** $\mathbf{0}$ **catch** Q is not allowed within our calculus.
- $\mathbf{cancel}(c)\,.\,P$ performs the ***cancellation*** of other processes with channel c.
- $s\lightning$ ***kills*** (terminates) all processes with session s, and is dynamically generated only at *runtime* from timeout failure or cancel processes.

Message queues: $s[\mathtt{p}]\blacktriangleright\sigma$ represents the ***output message queue*** of role $\mathtt{p}$ in session s. It contains all the messages previously sent by $\mathtt{p}$. The queue σ can be a sequence of messages of the form $\mathtt{q}!\mathtt{m}\langle s[\mathtt{r}]\rangle$, where $\mathtt{q}$ is the receiver, or ϵ, indicating an *empty* message queue. The set of *receivers in* σ, denoted as receivers(σ), is defined in a standard way as:

$$\text{receivers}(\mathtt{q}!\mathtt{m}\langle s[\mathtt{r}]\rangle\cdot\sigma') = \{\mathtt{q}\}\cup\text{receivers}(\sigma') \qquad \text{receivers}(\epsilon)=\emptyset$$

Operational Semantics We present the operational semantics of our session π-calculus for modelling the behaviour of affine timed processes, including asynchronous communication, time progression, timeout activation, and failure handling.

▶ **Definition 2** (Semantics). *A* **try-catch** context $\mathbb{E}$ *is defined as* $\mathbb{E} ::= \mathbf{try}\ \mathbb{E}\ \mathbf{catch}\ P \mid [\,]$, *and a* reduction context $\mathbb{C}$ *is defined as* $\mathbb{C} ::= \mathbb{C}\mid P \mid (\nu s)\mathbb{C} \mid \mathbf{def}\ D\ \mathbf{in}\ \mathbb{C} \mid [\,]$. *The* reductions $\to$, $\rightarrowtail$, and $\rightharpoonup$ *are inductively defined in Fig. 3 (top), with respect to a* structural congruence $\equiv$ *depicted in Fig. 3 (bottom). We write* $\to^*$, $\rightarrowtail^*$, *and* $\rightharpoonup^*$ *for their reflexive and transitive closures, respectively.* $P\not\to$ *(or* $P\not\rightarrowtail$, $P\not\rightharpoonup$*) means* $\nexists P'$ *such that* $P\to P'$ *(or* $P\rightarrowtail P'$, $P\rightharpoonup P'$*) is derivable. We say* P has a communication error *iff* $\exists\mathbb{C}$ *with* $P=\mathbb{C}[\mathbf{cerr}]$.

We decompose the reduction rules in Fig. 3 into three relations: $\rightarrowtail$ represents *instantaneous* reductions without time consumption, $\rightharpoonup$ handles time-consuming steps, and $\to$ is a

$$
\begin{array}{rll}
\text{[R-OUT]} & \mathbb{E}\left[s[\mathtt{q}]^{\mathtt{n}}[\mathtt{p}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.Q\right] \mid s[\mathtt{q}]\blacktriangleright\sigma \;\rightarrowtail\; Q \mid s[\mathtt{q}]\blacktriangleright\sigma\cdot\mathtt{p}!\mathtt{m}\langle s'[\mathtt{r}]\rangle\cdot\epsilon & \\
\text{[R-IN]} & \mathbb{E}\left[s[\mathtt{p}]^{\mathtt{n}}[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i\right] \mid s[\mathtt{q}]\blacktriangleright\mathtt{p}!\mathtt{m}_k\langle s'[\mathtt{r}]\rangle\cdot\sigma \;\rightarrowtail\; P_k\{^{s'[\mathtt{r}]}/_{x_k}\} \mid s[\mathtt{q}]\blacktriangleright\sigma & (k\in I) \\
\text{[R-ERR]} & \mathbb{E}\left[s[\mathtt{p}]^{\mathtt{n}}[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i\right] \mid s[\mathtt{q}]\blacktriangleright\mathtt{p}!\mathtt{m}\langle s'[\mathtt{r}]\rangle\cdot\sigma \;\rightarrowtail\; \textbf{cerr} & (\forall i\in I:\mathtt{m}_i\neq\mathtt{m}) \\
\text{[R-DET]} & \models\delta[t/C] \text{ implies } \mathbb{E}[\textbf{delay}(\delta).P] \;\rightarrowtail\; \textbf{delay}(t).P & \\
\text{[R-TIME]} & P \;\rightharpoonup\; \Psi_t(P) & \\
\text{[R-FAIL]} & \mathtt{timeout}[P] \;\rightarrowtail\; s\lightning & (\exists\mathtt{r}.\,\mathrm{subjP}(P)=\{s[\mathtt{r}]\}) \\
\text{[R-CAN]} & \mathbb{E}[\textbf{cancel}(s[\mathtt{p}]).Q] \;\rightarrowtail\; s\lightning \mid Q & \\
\text{[R-FAILCAT]} & \textbf{try } \mathtt{timeout}[P] \textbf{ catch } Q \;\rightarrowtail\; s\lightning \mid Q & (\exists\mathtt{r}.\,\mathrm{subjP}(P)=\{s[\mathtt{r}]\}) \\
\text{[C-CAT]} & \textbf{try } P \textbf{ catch } Q \mid s\lightning \;\rightarrowtail\; Q \mid s\lightning & (\exists\mathtt{r}.\,\mathrm{subjP}(P)=\{s[\mathtt{r}]\}) \\
\text{[C-IN]} & s[\mathtt{p}]^{\mathtt{n}}[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s[\mathtt{q}]\blacktriangleright\sigma \mid s\lightning & \\
 & \quad\rightarrowtail\; (\nu s')\,(P_k\{^{s'[\mathtt{r}]}/_{x_k}\} \mid s'\lightning) \mid s[\mathtt{q}]\blacktriangleright\sigma \mid s\lightning & (\mathtt{p}\notin\mathrm{receivers}(\sigma),\, k\in I,\, s'\notin\mathrm{fc}(P_k)) \\
\text{[C-QUEUE]} & s[\mathtt{p}]\blacktriangleright\mathtt{q}!\mathtt{m}\langle s'[\mathtt{r}]\rangle\cdot\sigma \mid s\lightning \;\rightarrowtail\; s[\mathtt{p}]\blacktriangleright\sigma \mid s\lightning \mid s'\lightning & \\
\text{[R-}X\text{]} & \textbf{def } X(x_1,\dots,x_n)=P \textbf{ in } (X\langle s_1[\mathtt{p}_1],\dots,s_n[\mathtt{p}_n]\rangle \mid Q) & \\
 & \quad\rightarrowtail\; \textbf{def } X(x_1,\dots,x_n)=P \textbf{ in } (P\{^{s_1[\mathtt{p}_1]}/_{x_1}\}\cdots\{^{s_n[\mathtt{p}_n]}/_{x_n}\} \mid Q) & \\
\text{[R-CTX]} & P \rightarrowtail P' \text{ implies } \mathbb{C}[P] \rightarrowtail \mathbb{C}\left[P'\right] & \\
\text{[R-}\equiv\text{]} & P'\equiv P \rightarrowtail Q\equiv Q' \text{ implies } P' \rightarrowtail Q' \qquad \text{[R-}\equiv\text{T]} \; P'\equiv P \rightharpoonup Q\equiv Q' \text{ implies } P' \rightharpoonup Q' & \\
\text{[R-INS]} & P \rightarrowtail P' \text{ implies } P \to P' \qquad \text{[R-TC]} \; P \rightharpoonup P' \text{ implies } P \to P' &
\end{array}
$$

- -

$$
\begin{gathered}
P\mid Q\equiv Q\mid P \quad (P\mid Q)\mid R\equiv P\mid(Q\mid R) \quad P\mid\mathbf{0}\equiv P \quad (\nu s)\,\mathbf{0}\equiv\mathbf{0} \quad (\nu s)\left(\nu s'\right)P\equiv\left(\nu s'\right)(\nu s)\,P \quad s\lightning\mid s\lightning\equiv s\lightning \\
(\nu s)\,(P\mid Q)\equiv P\mid(\nu s)\,Q \text{ if } s\notin\mathrm{fc}(P) \quad \textbf{def } D \textbf{ in } \mathbf{0}\equiv\mathbf{0} \quad \textbf{def } D \textbf{ in } (\nu s)\,P\equiv(\nu s)\,(\textbf{def } D \textbf{ in } P) \text{ if } s\notin\mathrm{fc}(D) \\
\textbf{delay}(0).P\equiv P \quad \textbf{def } D \textbf{ in } (P\mid Q)\equiv(\textbf{def } D \textbf{ in } P)\mid Q \text{ if } \mathrm{dpv}(D)\cap\mathrm{fpv}(Q)=\emptyset \\
(\nu s)\,(s[\mathtt{p}_1]\blacktriangleright\epsilon\mid\cdots\mid s[\mathtt{p}_n]\blacktriangleright\epsilon)\equiv\mathbf{0} \quad \textbf{def } D \textbf{ in } (\textbf{def } D' \textbf{ in } P)\equiv\textbf{def } D' \textbf{ in } (\textbf{def } D \textbf{ in } P) \\
\text{if } (\mathrm{dpv}(D)\cup\mathrm{fpv}(D))\cap\mathrm{dpv}\left(D'\right)=\left(\mathrm{dpv}\left(D'\right)\cup\mathrm{fpv}\left(D'\right)\right)\cap\mathrm{dpv}(D)=\emptyset \\
s[\mathtt{p}]\blacktriangleright\sigma\cdot\mathtt{q}_1!\mathtt{m}_1\langle s_1[\mathtt{r}_1]\rangle\cdot\mathtt{q}_2!\mathtt{m}_2\langle s_2[\mathtt{r}_2]\rangle\cdot\sigma'\equiv s[\mathtt{p}]\blacktriangleright\sigma\cdot\mathtt{q}_2!\mathtt{m}_2\langle s_2[\mathtt{r}_2]\rangle\cdot\mathtt{q}_1!\mathtt{m}_1\langle s_1[\mathtt{r}_1]\rangle\cdot\sigma' \quad \text{if } \mathtt{q}_1\neq\mathtt{q}_2
\end{gathered}
$$

Figure 3 Top: reduction rules for ATMP session π-calculus. Bottom: structural congruence rules for the ATMP π-calculus, where $\mathrm{fpv}(D)$ is the set of *free process variables* in D, and $\mathrm{dpv}(D)$ is the set of *declared process variables* in D. New rules are highlighted.

general relation that can arise either from $\rightarrowtail$ by [R-INS] or $\rightharpoonup$ by [R-TC]. Now let us explain the operational semantics rules for our session π-calculus.

Communication: Rules [R-OUT] and [R-IN] model asynchronous communication by queuing and dequeuing *pending* messages, respectively. Rule [R-ERR] is triggered by a message label mismatch, resulting in a fatal **c**ommunication **err**or.

Time: Rule [R-DET] specifies a deterministic delay of a specific duration t, where t is a solution to the clock constraint δ. Rule [R-TIME] incorporates a time-passing function $\Psi_t(P)$, depicted in Fig. 4, to represent time delays within a process. This *partial* function simulates a delay of time t that may occur at different parts of the process. It is *undefined* only if P is a time-consuming delay, i.e. $P = \textbf{delay}(\delta).P'$, or if the specified delay time t exceeds the duration of a runtime deterministic delay, i.e. $P = \textbf{delay}(t').P'$ with $t > t'$. The latter case arises because deterministic delays must always adhere to their specified durations, e.g. if a program is instructed to sleep for 5 time units, it must strictly follow this duration.

Notably, $\Psi_t(P)$ acts as the *only mechanism* for triggering a timeout failure $\mathtt{timeout}[P]$, resulting from a timed selection or branching. Such a timeout failure occurs when $\Psi_t(P)$ is defined, and the specified delay t exceeds a *deadline* set within P.

Cancellation: Rules [C-IN] and [C-QUEUE] model the process cancellations. [C-IN] is triggered only when there are no messages in the queue that can be received from $\mathtt{q}$ via the endpoint $s[\mathtt{p}]$. Cancellation of a timed selection is expected to eventually occur via [C-QUEUE]; therefore, there is no specific rule dedicated to it. Similarly, in our implementation, the timed selection is not directly cancelled either.

$$\Psi_t(\mathbf{0}) = \mathbf{0} \quad \Psi_t(P_1 \mid P_2) = \Psi_t(P_1) \mid \Psi_t(P_2) \quad \Psi_t((\nu s)\, P) = (\nu s)\, \Psi_t(P) \quad \Psi_t(\mathtt{timeout}[P]) = \mathtt{timeout}[P]$$

$$\Psi_t(\mathbf{cerr}) = \mathbf{cerr} \quad \Psi_t(\mathbf{def}\ D\ \mathbf{in}\ P) = \mathbf{def}\ D\ \mathbf{in}\ \Psi_t(P) \quad \Psi_t(\mathbf{try}\ \ P\ \ \mathbf{catch}\ \ Q) = \mathbf{try}\ \ \Psi_t(P)\ \ \mathbf{catch}\ \ \Psi_t(Q)$$

$$\Psi_t(s[\mathtt{p}] \blacktriangleright \sigma) = s[\mathtt{p}] \blacktriangleright \sigma \quad \Psi_t(\mathbf{delay}(\delta)\,.\,P) = \mathtt{undefined} \quad \Psi_t(\mathbf{cancel}(c)\,.\,Q) = \mathbf{cancel}(c)\,.\,\Psi_t(Q)$$

$$\Psi_t(\mathbf{delay}(t')\,.\,P) = \begin{cases} \mathbf{delay}(t'-t)\,.\,P & \text{if } t' \geq t \\ \mathtt{undefined} & \text{otherwise} \end{cases} \quad \Psi_t(c^{\infty}[\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i) = c^{\infty}[\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i$$

$$\Psi_t(c^{t'}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P) = \begin{cases} c^{t'-t}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P & \text{if } t' \geq t \\ \mathtt{timeout}[c^{t'}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P] & \text{otherwise} \end{cases} \quad \Psi_t(c^{\infty}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P) = c^{\infty}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P$$

$$\Psi_t(s\lightning) = s\lightning \quad \Psi_t(c^{t'}[\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i) = \begin{cases} c^{t'-t}[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i & \text{if } t' \geq t \\ \mathtt{timeout}[c^{t'}[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i] & \text{otherwise} \end{cases}$$

Figure 4 Time-passing function $\Psi_t(P)$.

Rules [R-Can] and [C-Cat], adapted from [28], state cancellations from other parties. [R-Can] facilitates cancellation and generates a kill process, while [C-Cat] transitions to the **catch** process Q due to the termination of session s, where the **try** process P is communicating on s. Therefore, the set of *subjects* of process P, denoted as $\mathrm{subjP}(P)$, is included in the side condition of [C-Cat] to ensure that P has a prefix at s, as defined below:

$$\mathrm{subjP}(\mathbf{0}) = \mathrm{subjP}(\mathbf{cerr}) = \emptyset \quad \mathrm{subjP}(P \mid Q) = \mathrm{subjP}(P) \cup \mathrm{subjP}(Q) \quad \mathrm{subjP}(s[\mathtt{p}] \blacktriangleright \sigma) = \left\{ s[\mathtt{p}]^{\Omega} \right\}$$
$$\mathrm{subjP}((\nu s)\, P) = \mathrm{subjP}(P) \setminus (\{s[\mathtt{p}_i]\}_{i\in I} \cup \left\{ s[\mathtt{p}_i]^{\Omega} \right\}_{i\in I})$$
$$\mathrm{subjP}(\mathbf{def}\ X(\widetilde{x}) = P\ \mathbf{in}\ Q) = \mathrm{subjP}(Q) \cup \mathrm{subjP}(P) \setminus \left\{ \widetilde{x} \right\} \text{ with } \mathrm{subjP}(X\langle\widetilde{c}\rangle) = \mathrm{subjP}(P\left\{\widetilde{c}/\widetilde{x}\right\})$$
$$\mathrm{subjP}(c^{\mathtt{n}}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P) = \mathrm{subjP}(c^{\mathtt{n}}[\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i) = \mathrm{subjP}(\mathbf{cancel}(c)\,.\,P) = \{c\}$$
$$\mathrm{subjP}(\mathbf{delay}(\delta)\,.\,P) = \mathrm{subjP}(\mathbf{delay}(t)\,.\,P) = \mathrm{subjP}(\mathbf{try}\ \ P\ \ \mathbf{catch}\ \ Q) = \mathrm{subjP}(\mathtt{timeout}[P]) = \mathrm{subjP}(P)$$

Subjects of processes determine sessions that may need cancellation, a crucial aspect for handling failed or cancelled processes properly. In our definition, subjects not only denote the endpoints via which processes start interacting but also indicate whether they are used for message queue processes. Specifically, an endpoint $s[\mathtt{p}]$ annotated with Ω signifies its use in a queue process. This additional annotation, and thus the distinction it implies, is pivotal in formulating the typing rule for the **try-catch** process, as discussed later in § 4.4, where we rely on subjects to exclude queue processes within any **try** construct.

Timeout Handling: Rules [R-Fail] and [R-FailCat] address time failures. In the event of a timeout, a killing process is generated. Moreover, in [R-FailCat], the **catch** process Q is triggered. To identify the session requiring termination, the set of subjects of the failure process $\mathtt{timeout}[P]$ is considered in both rules as a side condition. Note that a timeout arises exclusively from timed selection or branching. Therefore, the subject set of $\mathtt{timeout}[P]$ must contain a *single* endpoint devoid of Ω, indicating the generation of only one killing process.

Standard: Rules [R-X], [R-Ctx], and [R-$\equiv$] are standard [37, 28]. [R-X] expands process definitions when invoked; [R-Ctx] and [R-$\equiv$] allow processes to reduce under reduction contexts and through structural congruence, respectively. Rule [R-$\equiv$T] introduces a timed variant of [R-$\equiv$], enabling time-consuming reductions via structural congruence.

Congruence: As shown in Fig. 3 (bottom), we introduce additional congruence rules related to queues, delays, and process killings, alongside standard rules from [37]. Specifically, two rules are proposed for queues: the first addresses the *garbage* collection of queues that are not referenced by any process, while the second rearranges messages with different receivers. The rule for delays states that adding a delay of zero time units has no effect on the process execution. The rule regarding process killings eliminates duplicate kills.

▶ **Example 3.** Consider the processes: $P_1 = s[\mathtt{Sat}]^{0.4}[\mathtt{Ser}]\oplus\mathtt{Data}.\mathbf{0}$, $P_2 = s[\mathtt{Ser}]^{0.3}[\mathtt{Sat}]\mathtt{Data}.\mathbf{0}$, and $P_3 = s[\mathtt{Sat}] \blacktriangleright \epsilon$. Rule [C-Cat] can be applied to **try** P_1 **catch** $Q \mid s\lightning$, as $\mathrm{subjP}(P_1) = \{s[\mathtt{Sat}]\}$ satisfies its side condition. However, neither $\mathtt{timeout}[P_1 \mid P_2]$ nor $\mathtt{timeout}[P_3]$ can generate the killing process $s\lightning$, as $\mathrm{subjP}(P_1 \mid P_2) = \{s[\mathtt{Sat}], s[\mathtt{Ser}]\}$, whereas $\mathrm{subjP}(P_3) = \left\{ s[\mathtt{Sat}]^{\Omega} \right\}$.

▶ **Example 4.** Processes $Q_{\texttt{Sen}}$, $Q_{\texttt{Sat}}$, and $Q_{\texttt{Ser}}$ interact on a session s:

$$Q_{\texttt{Sen}} = \textbf{delay}(C_{\texttt{Sen}} = 6.5) \,.\, Q'_{\texttt{Sen}} \mid s[\texttt{Sen}] \blacktriangleright \epsilon \quad \text{where} \quad Q'_{\texttt{Sen}} = \textbf{try} \;\; s[\texttt{Sen}]^{0.3}[\texttt{Sat}] \oplus \texttt{Data} \;\; \textbf{catch} \;\; \textbf{cancel}(s[\texttt{Sen}])$$

$$Q_{\texttt{Sat}} = \textbf{delay}(C_{\texttt{Sat}} = 6) \,.\, Q'_{\texttt{Sat}} \mid s[\texttt{Sat}] \blacktriangleright \epsilon \quad \text{where} \quad Q'_{\texttt{Sat}} = s[\texttt{Sat}]^{0.2}[\texttt{Sen}] \sum \begin{Bmatrix} \texttt{Data}.s[\texttt{Sat}]^{0.3}[\texttt{Ser}] \oplus \texttt{Data} \\ \texttt{fail}.s[\texttt{Sat}]^{0.4}[\texttt{Ser}] \oplus \texttt{fatal} \end{Bmatrix}$$

$$Q_{\texttt{Ser}} = \textbf{delay}(C_{\texttt{Ser}} = 6) \,.\, Q'_{\texttt{Ser}} \mid s[\texttt{Ser}] \blacktriangleright \epsilon \quad \text{where} \quad Q'_{\texttt{Ser}} = s[\texttt{Ser}]^{0.8}[\texttt{Sat}] \sum \{\texttt{Data}, \texttt{fatal}\}$$

Process $Q_{\texttt{Sen}}$ delays for exactly 6.5 time units before executing process $Q'_{\texttt{Sen}}$. Here, $Q'_{\texttt{Sen}}$ attempts to use $s[\texttt{Sen}]$ to send `Data` to `Sat` within 0.3 time units. If the attempt fails, the cancellation of $s[\texttt{Sen}]$ is triggered. Process $Q_{\texttt{Sat}}$ waits for precisely 6 time units before using $s[\texttt{Sat}]$ to receive either `Data` or `fail` from `Sen` within 0.2 time units; subsequently, in the first case, it uses $s[\texttt{Sat}]$ to send `Data` to `Ser` within 0.3 time units, while in the latter, it uses $s[\texttt{Sat}]$ to send `fail` to `Ser` within 0.4 time units. Similarly, process $Q_{\texttt{Ser}}$ waits 6 time units before using $s[\texttt{Ser}]$ to receive either `Data` or `fatal` from `Sat` within 0.8 time units.

In $Q_{\texttt{Sen}}$, $s[\texttt{Sen}]$ can only start sending `Data` to `Sat` after 6.5 time units, whereas in $Q_{\texttt{Sat}}$, $s[\texttt{Sat}]$ must receive the message from `Sen` within 0.2 time units after a 6-time unit delay. Consequently, $s[\texttt{Sat}]$ fails to receive the message from `Sen` within the specified interval, resulting in a timeout failure, i.e.

$$\begin{aligned} Q_{\texttt{Sen}} \mid Q_{\texttt{Sat}} \mid Q_{\texttt{Ser}} &\rightarrowtail \textbf{delay}(6.5) \,.\, Q'_{\texttt{Sen}} \mid s[\texttt{Sen}] \blacktriangleright \epsilon \mid \textbf{delay}(6) \,.\, Q'_{\texttt{Sat}} \mid s[\texttt{Sat}] \blacktriangleright \epsilon \mid \textbf{delay}(6) \,.\, Q'_{\texttt{Ser}} \mid s[\texttt{Ser}] \blacktriangleright \epsilon \\ &\rightharpoonup \Psi_{6.5}(\textbf{delay}(6.5) \,.\, Q'_{\texttt{Sen}} \mid s[\texttt{Sen}] \blacktriangleright \epsilon \mid \textbf{delay}(6) \,.\, Q'_{\texttt{Sat}} \mid s[\texttt{Sat}] \blacktriangleright \epsilon \mid \textbf{delay}(6) \,.\, Q'_{\texttt{Ser}} \mid s[\texttt{Ser}] \blacktriangleright \epsilon) \\ &\equiv Q'_{\texttt{Sen}} \mid s[\texttt{Sen}] \blacktriangleright \epsilon \mid \texttt{timeout}[Q'_{\texttt{Sat}}] \mid s[\texttt{Sat}] \blacktriangleright \epsilon \mid \Psi_{0.5}(Q'_{\texttt{Ser}}) \mid s[\texttt{Ser}] \blacktriangleright \epsilon \end{aligned}$$

Therefore, the kill process $s\lightning$ is generated from $\texttt{timeout}[Q'_{\texttt{Sat}}]$, successfully terminating the process $Q_{\texttt{Sen}} \mid Q_{\texttt{Sat}} \mid Q_{\texttt{Ser}}$ by the following reductions:

$$\begin{aligned} & Q'_{\texttt{Sen}} \mid s[\texttt{Sen}] \blacktriangleright \epsilon \mid \texttt{timeout}[Q'_{\texttt{Sat}}] \mid s[\texttt{Sat}] \blacktriangleright \epsilon \mid \Psi_{0.5}(Q'_{\texttt{Ser}}) \mid s[\texttt{Ser}] \blacktriangleright \epsilon \\ \rightarrowtail \;\; & Q'_{\texttt{Sen}} \mid s[\texttt{Sen}] \blacktriangleright \epsilon \mid s\lightning \mid s[\texttt{Sat}] \blacktriangleright \epsilon \mid \Psi_{0.5}(Q'_{\texttt{Ser}}) \mid s[\texttt{Ser}] \blacktriangleright \epsilon \\ \rightarrowtail \;\; & \textbf{cancel}(s[\texttt{Sen}]) \mid s[\texttt{Sen}] \blacktriangleright \epsilon \mid s\lightning \mid s[\texttt{Sat}] \blacktriangleright \epsilon \mid \mathbf{0} \mid s[\texttt{Ser}] \blacktriangleright \epsilon \\ \rightarrowtail \;\; & s\lightning \mid \mathbf{0} \mid s[\texttt{Sen}] \blacktriangleright \epsilon \mid s\lightning \mid s[\texttt{Sat}] \blacktriangleright \epsilon \mid \mathbf{0} \mid s[\texttt{Ser}] \blacktriangleright \epsilon \;\equiv\; \mathbf{0} \mid s\lightning \end{aligned}$$

4 Affine Timed Multiparty Session Type System

In this section, we introduce our affine timed multiparty session type system. We begin by exploring the types used in ATMP, as well as subtyping and projection, in § 4.1. We furnish a Labelled Transition System (LTS) semantics for typing environments (collections of timed local types and queue types) in § 4.2, and timed global types in § 4.3, illustrating their relationship with Thms. 13 and 14. Furthermore, we present a type system for our ATMP session π-calculus in § 4.4. Finally, we show the main properties of the type system: *subject reduction* (Thm. 17), *session fidelity* (Thm. 21), and *deadlock-freedom* (Thm. 24), in § 4.5.

4.1 Timed Multiparty Session Types

Affine session frameworks keep the original system's type-level syntax intact, requiring no changes. To introduce affine timed asynchronous multiparty session types, we simply need to augment global and local types with clock constraints and resets introduced in § 3 to derive *timed global and local types*. The syntax of types used in this paper is presented in Fig. 5. As usual, all types are required to be closed and have guarded recursion variables.

Sorts are ranged over $S, S', S_i, \ldots$, and facilitate the delegation of the remaining behaviour T to the receiver, who can execute it under any clock assignment satisfying δ.

Timed Global Types are ranged over $G, G', G_i, \ldots$, and describe an *overview* of the behaviour for all roles ($\texttt{p}, \texttt{q}, \texttt{s}, \texttt{t}, \ldots$) belonging to a (fixed) set $\mathcal{R}$. The set of roles in a timed global type G is denoted as $\text{roles}(G)$, while the set of its free variables as $\text{fv}(G)$.

$$
\begin{array}{rcll}
S & ::= & (\delta, T) & \text{sort} \\
G & ::= & \mathtt{p}\to\mathtt{q}\colon \{\mathtt{m_i}(S_i)\{\delta_{\mathrm{O}i}, \lambda_{\mathrm{O}i}, \delta_{\mathrm{I}i}, \lambda_{\mathrm{I}i}\}.G_i\}_{i\in I} & \text{transmission} \\
 & \mid & \mathtt{p}\rightsquigarrow\mathtt{q}\colon j\ \{\mathtt{m_i}(S_i)\{\delta_{\mathrm{O}i}, \lambda_{\mathrm{O}i}, \delta_{\mathrm{I}i}, \lambda_{\mathrm{I}i}\}.G_i\}_{i\in I}\ (j\in I) & \text{transmission en route} \\
 & \mid & \mu\mathbf{t}.G \quad\mid\quad \mathbf{t} \quad\mid\quad \mathbf{end} & \text{recursion, type variable, termination} \\
T & ::= & \mathtt{p}\&\{\mathtt{m_i}(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i\in I} \quad\mid\quad \mathtt{p}\oplus\{\mathtt{m_i}(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i\in I} & \text{external choice, internal choice} \\
 & \mid & \mu\mathbf{t}.T \quad\mid\quad \mathbf{t} \quad\mid\quad \mathbf{end} & \text{recursion, type variable, termination} \\
\mathcal{M} & ::= & \mathtt{p}!\mathtt{m}(S)\cdot\mathcal{M} \quad\mid\quad \oslash & \text{queue types}
\end{array}
$$

Figure 5 Syntax of timed global types, timed local types, and queue types.

A transmission $\mathtt{p}\to\mathtt{q}\colon \{\mathtt{m_i}(S_i)\{\delta_{\mathrm{O}i}, \lambda_{\mathrm{O}i}, \delta_{\mathrm{I}i}, \lambda_{\mathrm{I}i}\}.G_i\}_{i\in I}$ represents a message sent from role $\mathtt{p}$ to role $\mathtt{q}$, with labels $\mathtt{m}_i$, payload types S_i (which are sorts), and continuations G_i, where i is taken from an index set I, and $\mathtt{m}_i$ taken from a fixed set of all labels $\mathcal{M}$. Each branch is associated with a *time assertion* consisting of four components: $\delta_{\mathrm{O}i}$ and $\lambda_{\mathrm{O}i}$ for the output (sending) action, and $\delta_{\mathrm{I}i}$ and $\lambda_{\mathrm{I}i}$ for the input (receiving) action. These components specify the clock constraint and reset predicate for the respective actions. A message can be sent (or received) at any time satisfying the guard $\delta_{\mathrm{O}i}$ (or $\delta_{\mathrm{I}i}$), and the clocks in $\lambda_{\mathrm{O}i}$ (or $\lambda_{\mathrm{I}i}$) are reset upon sending (or receiving). In addition to the standard requirements for global types as in [11], we impose a condition from [4], stating that sets of clocks "owned" by different roles, i.e. those that can be read and reset, must be pairwise disjoint. Furthermore, the clock constraint and reset predicate of an output or input action performed by a role are defined only over the clocks owned by that role.

A transmission en route $\mathtt{p}\rightsquigarrow\mathtt{q}\colon j\ \{\mathtt{m_i}(S_i)\{\delta_{\mathrm{O}i}, \lambda_{\mathrm{O}i}, \delta_{\mathrm{I}i}, \lambda_{\mathrm{I}i}\}.G_i\}_{i\in I}$ $(j \in I)$ is a *runtime* construct to represent a message $\mathtt{m}_j$ sent by $\mathtt{p}$, and yet to be received by $\mathtt{q}$. Recursion $\mu\mathbf{t}.G$ and termination $\mathbf{end}$ (omitted where unambiguous) are standard [11]. Note that contractive requirements [34, §21.8], i.e. ensuring that each recursion variable $\mathbf{t}$ is bound within a $\mu\mathbf{t}.\ldots$ and is guarded, are applied in recursive types.

Timed Local Types (timed session types) are ranged over $T, U, T', U', T_i, U_i, \ldots$, and describe the behaviour of a *single* role. An *internal choice* (*selection*) $\mathtt{p}\oplus\{\mathtt{m_i}(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i\in I}$ (or *external choice* (*branching*) $\mathtt{p}\&\{\mathtt{m_i}(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i\in I}$) states that the *current* role is to *send* to (or *receive* from) the role $\mathtt{p}$ when δ_i is satisfied, followed by resetting the clocks in λ_i. *Recursive* and *termination* types are defined similarly to timed global types. The requirements for the index set, labels, clock constraints, and reset predicates in timed local types mirror those in timed global types.

Queue Types are ranged over $\mathcal{M}, \mathcal{M}', \mathcal{M}_i, \ldots$, and represent (possibly empty) sequences of *message types* $\mathtt{p}!\mathtt{m}(S)$ having receiver $\mathtt{p}$, label $\mathtt{m}$, and payload type S (omitted when $S = (\delta, \mathbf{end})$). As interactions in our formalisation are asynchronous, queue types are used to capture the states in which messages are in transit. We adopt the notation receivers$(\cdot)$ from §3 to denote the set of *receivers* in $\mathcal{M}$ as receivers$(\mathcal{M})$ as well, with a similar definition.

Subtyping We introduce a subtyping relation $\leqslant$ on timed local types in Def. 5, based on the standard behaviour-preserving subtyping [37]. This relation indicates that a smaller type entails fewer external choices but more internal choices.

▶ **Definition 5** (Subtyping). *The* subtyping relation $\leqslant$ *is coinductively defined:*

$$
\frac{\forall i \in I \quad S'_i \leqslant S_i \quad \delta_i = \delta'_i \quad \lambda_i = \lambda'_i \quad T_i \leqslant T'_i}{\mathtt{p}\oplus\{\mathtt{m_i}(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i\in I\cup J} \leqslant \mathtt{p}\oplus\{\mathtt{m_i}(S'_i)\{\delta'_i, \lambda'_i\}.T'_i\}_{i\in I}}\ [\textsc{Sub-}\oplus]
$$

$$
\frac{\forall i \in I \quad S_i \leqslant S'_i \quad \delta_i = \delta'_i \quad \lambda_i = \lambda'_i \quad T_i \leqslant T'_i}{\mathtt{p}\&\{\mathtt{m_i}(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i\in I} \leqslant \mathtt{p}\&\{\mathtt{m_i}(S'_i)\{\delta'_i, \lambda'_i\}.T'_i\}_{i\in I\cup J}}\ [\textsc{Sub-}\&] \qquad \frac{}{\mathbf{end} \leqslant \mathbf{end}}\ [\textsc{Sub-}\mathbf{end}]
$$

$$
\frac{T \leqslant T'}{(\delta, T) \leqslant (\delta, T')}\ [\textsc{Sub-S}] \qquad \frac{T\{\mu\mathbf{t}.T/\mathbf{t}\} \leqslant T'}{\mu\mathbf{t}.T \leqslant T'}\ [\textsc{Sub-}\mu\textsc{L}] \qquad \frac{T \leqslant T'\left\{\mu\mathbf{t}.T'/\mathbf{t}\right\}}{T \leqslant \mu\mathbf{t}.T'}\ [\textsc{Sub-}\mu\textsc{R}]
$$

Projection of a timed global type G onto a role $\mathtt{p}$ yields a timed local type. Our definition of projection in Def. 6 is mostly standard [37], with the addition of projecting time assertions onto the sender and receiver, respectively.

▶ **Definition 6** (Projection). *The* projection of a timed global type G onto a role $\mathtt{p}$, *written as* $G{\upharpoonright}\mathtt{p}$, *is:*

$$(\mathtt{q}{\to}\mathtt{r}{:}\,\{\mathtt{m_i}(S_i)\{\delta_{Oi},\lambda_{Oi},\delta_{Ii},\lambda_{Ii}\}.G_i\}_{i\in I}){\upharpoonright}\mathtt{p} = \begin{cases} \mathtt{r}{\oplus}\{\mathtt{m_i}(S_i)\{\delta_{Oi},\lambda_{Oi}\}.(G_i{\upharpoonright}\mathtt{p})\}_{i\in I} & \textit{if } \mathtt{p}=\mathtt{q} \\ \mathtt{q}\&\{\mathtt{m_i}(S_i)\{\delta_{Ii},\lambda_{Ii}\}.(G_i{\upharpoonright}\mathtt{p})\}_{i\in I} & \textit{if } \mathtt{p}=\mathtt{r} \\ \bigsqcap_{i\in I} G_i{\upharpoonright}\mathtt{p} & \textit{otherwise} \end{cases}$$

$$(\mathtt{q}{\rightsquigarrow}\mathtt{r}{:}j\,\{\mathtt{m_i}(S_i)\{\delta_{Oi},\lambda_{Oi},\delta_{Ii},\lambda_{Ii}\}.G_i\}_{i\in I}){\upharpoonright}\mathtt{p} = \begin{cases} G_j{\upharpoonright}\mathtt{p} & \textit{if } \mathtt{p}=\mathtt{q} \\ \mathtt{q}\&\{\mathtt{m_i}(S_i)\{\delta_{Ii},\lambda_{Ii}\}.(G_i{\upharpoonright}\mathtt{p})\}_{i\in I} & \textit{if } \mathtt{p}=\mathtt{r} \\ \bigsqcap_{i\in I} G_i{\upharpoonright}\mathtt{p} & \textit{otherwise} \end{cases}$$

$$(\mu\mathbf{t}.G){\upharpoonright}\mathtt{p} \;=\; \begin{cases} \mu\mathbf{t}.(G{\upharpoonright}\mathtt{p}) & \textit{if } \mathtt{p}\in\mathrm{roles}(G) \textit{ or } \mathrm{fv}(\mu\mathbf{t}.G)\neq\emptyset \\ \mathbf{end} & \textit{otherwise} \end{cases} \qquad \begin{array}{c} \mathbf{t}{\upharpoonright}\mathtt{p}=\mathbf{t} \\ \mathbf{end}{\upharpoonright}\mathtt{p}=\mathbf{end} \end{array}$$

where $\bigsqcap$ *is the* merge operator for timed session types*:*

$$\mathtt{p}\&\{\mathtt{m_i}(S_i)\{\delta_i,\lambda_i\}.T_i\}_{i\in I} \sqcap \mathtt{p}\&\left\{\mathtt{m_j}(S'_j)\{\delta'_j,\lambda'_j\}.T'_j\right\}_{j\in J} =$$
$$\mathtt{p}\&\{\mathtt{m_k}(S_k)\{\delta_k,\lambda_k\}.(T_k{\sqcap}T'_k)\}_{k\in I\cap J} \,\&\, \mathtt{p}\&\{\mathtt{m_i}(S_i)\{\delta_i,\lambda_i\}.T_i\}_{i\in I\setminus J} \,\&\, \mathtt{p}\&\left\{\mathtt{m_j}(S'_j)\{\delta'_j,\lambda'_j\}.T'_j\right\}_{j\in J\setminus I}$$
$$\mathtt{p}{\oplus}\{\mathtt{m_i}(S_i)\{\delta_i,\lambda_i\}.T_i\}_{i\in I} \sqcap \mathtt{p}{\oplus}\{\mathtt{m_i}(S_i)\{\delta_i,\lambda_i\}.T'_i\}_{i\in I} \;=\; \mathtt{p}{\oplus}\{\mathtt{m_i}(S_i)\{\delta_i,\lambda_i\}.(T_i\sqcap T'_i)\}_{i\in I}$$
$$\mu\mathbf{t}.T \sqcap \mu\mathbf{t}.U \;=\; \mu\mathbf{t}.(T\sqcap U) \qquad \mathbf{t}\sqcap\mathbf{t} \;=\; \mathbf{t} \qquad \mathbf{end}\sqcap\mathbf{end} \;=\; \mathbf{end}$$

▶ **Example 7.** Take the timed global type G, and timed local types $T_{\mathtt{Sat}}$ and $T_{\mathtt{Ser}}$ from § 2.1. Consider a timed global type G_{data}, derived from remote data (Fig. 1b) as well, representing data transmission from $\mathtt{Sen}$ to $\mathtt{Ser}$ via $\mathtt{Sat}$:

$$G_{\mathrm{data}} = \mathtt{Sen}{\to}\mathtt{Sat}{:}\,\{\mathtt{Data}\{6\le C_{\mathtt{Sen}}\le 7, C_{\mathtt{Sen}}:=0, 6\le C_{\mathtt{Sat}}\le 7,\emptyset\}.G\}$$

which can be projected onto roles $\mathtt{Sen}$, $\mathtt{Sat}$, and $\mathtt{Ser}$, respectively, as:

$$G_{\mathrm{data}}{\upharpoonright}\mathtt{Sen} = \mathtt{Sat}{\oplus}\mathtt{Data}\{6\le C_{\mathtt{Sen}}\le 7, C_{\mathtt{Sen}}:=0\}.\mathbf{end} \qquad G_{\mathrm{data}}{\upharpoonright}\mathtt{Ser} = G{\upharpoonright}\mathtt{Ser} = T_{\mathtt{Ser}}$$
$$G_{\mathrm{data}}{\upharpoonright}\mathtt{Sat} = \mathtt{Sen}\&\mathtt{Data}\{6\le C_{\mathtt{Sat}}\le 7,\emptyset\}.G{\upharpoonright}\mathtt{Sat} = \mathtt{Sen}\&\mathtt{Data}\{6\le C_{\mathtt{Sat}}\le 7,\emptyset\}.T_{\mathtt{Sat}}$$

4.2 Typing Environments

To reflect the behaviour of timed global types (§ 4.3), present a typing system for our session π-calculus (§4.4), and introduce type-level properties (§4.5), we formalise *typing environments* in Def. 8, followed by their Labelled Transition System (LTS) semantics in Def. 9.

▶ **Definition 8** (Typing Environments). *The typing environments* Θ *and* Γ *are defined as:*

$$\Theta \;::=\; \emptyset \;\big|\; \Theta,\, X{:}(\mathbb{V}_1,T_1),\ldots,(\mathbb{V}_n,T_n) \qquad \Gamma \;::=\; \emptyset \;\big|\; \Gamma, x{:}(\mathbb{V},T) \;\big|\; \Gamma, s[\mathtt{p}]{:}\tau$$

where τ *is a* timed-session/queue *type:* $\tau ::= (\mathbb{V},T) \mid \mathcal{M} \mid \mathcal{M};(\mathbb{V},T)$*, i.e. either a timed session type, a queue type, or a combination.*

The environment composition Γ_1,Γ_2 *is defined iff* $\forall c \in \mathrm{dom}(\Gamma_1)\cap\mathrm{dom}(\Gamma_2) : \Gamma_i(c) = \mathcal{M}$ *and* $\Gamma_j(c) = (\mathbb{V},T)$ *with* $i,j\in\{1,2\}$*, and for all such* c*, we posit* $(\Gamma_1,\Gamma_2)(c) = \mathcal{M};(\mathbb{V},T)$*.*

We write $\mathrm{dom}(\Gamma) = \{s\}$ *iff for any* $c\in\mathrm{dom}(\Gamma)$*, there is* $\mathtt{p}$ *such that* $c = s[\mathtt{p}]$ *(i.e.* Γ *only contains session* s*). We write* $s\notin\Gamma$ *iff* $\forall\mathtt{p} : s[\mathtt{p}]\notin\mathrm{dom}(\Gamma)$ *(i.e. session* s *does not occur in* Γ*). We write* Γ_s *iff* $\mathrm{dom}(\Gamma_s) = \{s\}$*,* $\mathrm{dom}(\Gamma_s)\subseteq\mathrm{dom}(\Gamma)$*, and* $\forall s[\mathtt{p}]\in\mathrm{dom}(\Gamma) : \Gamma(s[\mathtt{p}]) = \Gamma_s(s[\mathtt{p}])$ *(i.e. restriction of* Γ *to session* s*). We denote updates as* $\Gamma[c\mapsto\tau]$*:* $\Gamma[c\mapsto\tau](c) = \tau$ *and* $\Gamma[c\mapsto\tau](c') = \Gamma(c')$ *(where* $c\neq c'$*).*

Congruence and subtyping are imposed on typing environments: $\Gamma\equiv\Gamma'$ *(resp.* $\Gamma\leqslant\Gamma'$*) iff* $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ *and* $\forall c\in\mathrm{dom}(\Gamma) : \Gamma(c)\equiv\Gamma'(c)$ *(resp.* $\Gamma(c)\leqslant\Gamma'(c)$*), incorporating additional congruence and subtyping rules for time-session/queue types, as depicted in Fig. 6.*

$$(\mathbb{V}, T) \equiv (\mathbb{V}, T) \qquad \frac{\mathtt{p} \neq \mathtt{q}}{\mathtt{p}!\mathtt{m}_1(S_1)\cdot\mathtt{q}!\mathtt{m}_2(S_2)\cdot\mathcal{M} \;\equiv\; \mathtt{q}!\mathtt{m}_2(S_2)\cdot\mathtt{p}!\mathtt{m}_1(S_1)\cdot\mathcal{M}}$$

$$\oslash\cdot\oslash \equiv \oslash \qquad \mathtt{p}!\mathtt{m}(S)\cdot\oslash\cdot\mathcal{M} \equiv \oslash\cdot\mathtt{p}!\mathtt{m}(S)\cdot\mathcal{M} \qquad \frac{\mathcal{M} \equiv \mathcal{M}' \quad (\mathbb{V}, T) \equiv (\mathbb{V}, T')}{\mathcal{M}; (\mathbb{V}, T) \equiv \mathcal{M}'; (\mathbb{V}, T')}$$

$$\frac{}{\oslash \leqslant \oslash} \qquad \frac{S' \leqslant S \quad \mathcal{M} \leqslant \mathcal{M}'}{\mathtt{q}!\mathtt{m}(S)\cdot\mathcal{M} \leqslant \mathtt{q}!\mathtt{m}(S')\cdot\mathcal{M}'} \qquad \frac{T \leqslant T'}{(\mathbb{V}, T) \leqslant (\mathbb{V}, T')} \qquad \frac{\mathcal{M} \leqslant \mathcal{M}' \quad (\mathbb{V}, T) \leqslant (\mathbb{V}, T')}{\mathcal{M}; (\mathbb{V}, T) \leqslant \mathcal{M}'; (\mathbb{V}, T')}$$

Figure 6 Congruence (top) and subtyping (bottom) rules for timed-session/queue types.

In Def. 8, the typing environment Θ maps process variables to n-tuples of timed session types, while Γ maps variables to timed session types, and channels with roles to timed-session/queue types. Note that in our typing environments, timed session types are annotated with clock valuations, denoted as $(\mathbb{V}, T)$. This enables us to capture timing information within the type system, facilitating the tracking of the (virtual) time at which the next action can be validated during the execution of a process.

The congruence relation $\equiv$ for timed-session/queue types is inductively defined as in Fig. 6 (top), reordering queued messages with different receivers. Subtyping for timed-session/queue types extends Def. 5 with rules in Fig. 6 (bottom): particularly, rule [Sub-$\mathcal{M}$] states that a sequence of queued message types is a subtype of another if messages in the same position have identical receivers and labels, and their payload sorts are related by subtyping.

▶ **Definition 9** (Typing Environment Reduction). *Let α be a transition label of the form $s{:}\mathtt{p}!\mathtt{q}{:}\mathtt{m}$, $s{:}\mathtt{p}{,}\mathtt{q}{:}\mathtt{m}$, or t. The* typing environment transition $\xrightarrow{\alpha}$ *is inductively defined by the rules in Fig. 7 (top). We write $\Gamma \xrightarrow{\alpha}$ iff $\Gamma \xrightarrow{\alpha} \Gamma'$ for some Γ'. We define two* reductions $\Gamma \to_s \Gamma'$ *(where s is a session) and $\Gamma \to \Gamma'$ as follows:*

- *$\Gamma \to_s \Gamma'$ holds iff $\Gamma \xrightarrow{\alpha} \Gamma'$ with $\alpha \in \{s{:}\mathtt{p}!\mathtt{q}{:}\mathtt{m}, s{:}\mathtt{p}{,}\mathtt{q}{:}\mathtt{m}, t \mid \mathtt{p}, \mathtt{q} \in \mathcal{R}\}$ (where $\mathcal{R}$ is the set of all roles). We write $\Gamma \to_s$ iff $\Gamma \to_s \Gamma'$ for some Γ', and $\to_s^*$ as the reflexive and transitive closure of $\to_s$;*
- *$\Gamma \to \Gamma'$ holds iff $\Gamma \to_s \Gamma'$ for some s. We write $\Gamma \to$ iff $\Gamma \to \Gamma'$ for some Γ', and $\to^*$ as the reflexive and transitive closure of $\to$.*

The label $s{:}\mathtt{p}!\mathtt{q}{:}\mathtt{m}$ indicates that $\mathtt{p}$ sends the message $\mathtt{m}$ to $\mathtt{q}$ on session s, while $s{:}\mathtt{p}{,}\mathtt{q}{:}\mathtt{m}$ denotes the reception of $\mathtt{m}$ from $\mathtt{q}$ by $\mathtt{p}$ on s. Additionally, the label t ($\in \mathbb{R}_{\geq 0}$) represents a time action modelling the passage of time.

The (highlighted) main modifications in the reduction rules for typing environments, compared to standard rules, concern time. Rule [Γ-$\oplus$] states that an entry can perform an output transition by appending a message at the respective queue within the time specified by the output clock constraint. Dually, rule [Γ-&] allows an entry to execute an input transition, consuming a message from the corresponding queue within the specified input clock constraint, provided that the payloads are compatible through subtyping. Note that in both rules, the associated clock valuation of the reduced entry must be updated according to the reset.

Rules [Γ-,x] and [Γ-,τ] pertain to *untimed* reductions, i.e. $\alpha \neq t$, within a larger environment. Rule [Γ-Ts] models time passing on an entry of timed session type by incrementing the associated clock valuation, while rule [Γ-Tq] specifies that an entry of queue type is not affected with respect to time progression. Thus, rule [Γ-Tc] captures the corresponding time behaviour for a timed-session/queue type entry. Additionally, rule [Γ-,T] ensures that time elapses uniformly across compatibly composed environments. Other rules are standard: [Γ-μ] is for recursion, and [Γ-struct] ensures that reductions are closed under congruence.

The reduction $\Gamma \to_s \Gamma'$ indicates that the typing environment Γ can advance on session s, involving any roles, while $\Gamma \to \Gamma'$ signifies Γ progressing on any session. This distinction helps in illustrating properties of typed processes discussed in § 4.5.

$$\frac{\Gamma, s[\mathtt{p}]{:}(\mathbb{V}, T\{^{\mu\mathbf{t}.T}/_{\mathbf{t}}\}) \xrightarrow{\alpha} \Gamma'}{\Gamma, s[\mathtt{p}]{:}(\mathbb{V}, \mu\mathbf{t}.T) \xrightarrow{\alpha} \Gamma'}\,[\Gamma\text{-}\mu] \qquad \frac{\Gamma \xrightarrow{\alpha} \Gamma' \quad \alpha \neq t}{\Gamma, x{:}(\mathbb{V}, T) \xrightarrow{\alpha} \Gamma', x{:}(\mathbb{V}, T)}\,[\Gamma\text{-},\mathrm{X}] \qquad \frac{\Gamma \xrightarrow{\alpha} \Gamma' \quad \alpha \neq t}{\Gamma, s[\mathtt{p}]{:}\tau \xrightarrow{\alpha} \Gamma', s[\mathtt{p}]{:}\tau}\,[\Gamma\text{-},\tau]$$

$$\frac{k \in I \quad \mathbb{V} \models \delta_k}{s[\mathtt{p}]{:}\mathcal{M}; (\mathbb{V}, \mathtt{q}\oplus\{\mathtt{m_i}(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i\in I}) \xrightarrow{s:\mathtt{p}!\mathtt{q}:\mathtt{m}_k} s[\mathtt{p}]{:}\mathcal{M}\cdot\mathtt{q}!\mathtt{m}_k(S_k)\cdot\oslash; (\mathbb{V}[\lambda_k \mapsto 0], T_k)}\,[\Gamma\text{-}\oplus]$$

$$\frac{k \in I \quad \mathbb{V} \models \delta_k \quad S_k \leqslant S'_k}{s[\mathtt{p}]{:}\mathtt{q}!\mathtt{m}_k(S_k)\cdot\mathcal{M}, s[\mathtt{q}]{:}(\mathbb{V}, \mathtt{p}\&\left\{\mathtt{m_i}(S'_i)\{\delta_i, \lambda_i\}.T_i\right\}_{i\in I}) \xrightarrow{s:\mathtt{q},\mathtt{p}:\mathtt{m}_k} s[\mathtt{p}]{:}\mathcal{M}, s[\mathtt{q}]{:}(\mathbb{V}[\lambda_k \mapsto 0], T_k)}\,[\Gamma\text{-}\&]$$

$$c{:}(\mathbb{V}, T) \xrightarrow{t} c{:}(\mathbb{V}+t, T)\;[\Gamma\text{-}\mathrm{Ts}] \qquad s[\mathtt{p}]{:}\mathcal{M} \xrightarrow{t} s[\mathtt{p}]{:}\mathcal{M}\;[\Gamma\text{-}\mathrm{TQ}] \qquad s[\mathtt{p}]{:}\mathcal{M}; (\mathbb{V}, T) \xrightarrow{t} s[\mathtt{p}]{:}\mathcal{M}; (\mathbb{V}+t, T)\;[\Gamma\text{-}\mathrm{Tc}]$$

$$\frac{\Gamma_1 \xrightarrow{t} \Gamma'_1 \quad \Gamma_2 \xrightarrow{t} \Gamma'_2}{\Gamma_1, \Gamma_2 \xrightarrow{t} \Gamma'_1, \Gamma'_2}\,[\Gamma\text{-},\mathrm{T}] \qquad \frac{\Gamma \equiv \Gamma_1 \quad \Gamma_1 \xrightarrow{\alpha} \Gamma'_1 \quad \Gamma'_1 \equiv \Gamma'}{\Gamma \xrightarrow{\alpha} \Gamma'}\,[\Gamma\text{-}\mathrm{STRUCT}]$$

- -

$$\langle \mathbb{V}; G\rangle \xrightarrow{t} \langle \mathbb{V}+t; G\rangle\;[\mathrm{GR\text{-}T}] \qquad \frac{\langle \mathbb{V}; G\{^{\mu\mathbf{t}.G}/_{\mathbf{t}}\}\rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G'\rangle}{\langle \mathbb{V}; \mu\mathbf{t}.G\rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G'\rangle}\,[\mathrm{GR}\text{-}\mu]$$

$$\frac{j \in I \quad \mathbb{V} \models \delta_{\mathrm{O}j} \quad \mathbb{V}' = \mathbb{V}[\lambda_{\mathrm{O}j} \mapsto 0]}{\langle \mathbb{V}; \mathtt{p}\to\mathtt{q}{:}\left\{\mathtt{m_i}(S_i)\{\mathcal{A}_i\}.G'_i\right\}_{i\in I}\rangle \xrightarrow{s:\mathtt{p}!\mathtt{q}:\mathtt{m}_j} \langle \mathbb{V}'; \mathtt{p}\rightsquigarrow\mathtt{q}{:}j\left\{\mathtt{m_i}(S_i)\{\mathcal{A}_i\}.G'_i\right\}_{i\in I}\rangle}\,[\mathrm{GR}\text{-}\oplus]$$

$$\frac{j \in I \quad \mathbb{V} \models \delta_{\mathrm{I}j} \quad \mathbb{V}' = \mathbb{V}[\lambda_{\mathrm{I}j} \mapsto 0]}{\langle \mathbb{V}; \mathtt{p}\rightsquigarrow\mathtt{q}{:}j\left\{\mathtt{m_i}(S_i)\{\mathcal{A}_i\}.G'_i\right\}_{i\in I}\rangle \xrightarrow{s:\mathtt{q},\mathtt{p}:\mathtt{m}_j} \langle \mathbb{V}'; G'_j\rangle}\,[\mathrm{GR}\text{-}\&]$$

$$\frac{\forall i \in I : \langle \mathbb{V}; G'_i\rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G''_i\rangle \quad \mathtt{p}, \mathtt{q} \notin \mathrm{subject}(\alpha) \quad \alpha \neq t}{\langle \mathbb{V}; \mathtt{p}\to\mathtt{q}{:}\left\{\mathtt{m_i}(S_i)\{\mathcal{A}_i\}.G'_i\right\}_{i\in I}\rangle \xrightarrow{\alpha} \langle \mathbb{V}'; \mathtt{p}\to\mathtt{q}{:}\left\{\mathtt{m_i}(S_i)\{\mathcal{A}_i\}.G''_i\right\}_{i\in I}\rangle}\,[\mathrm{GR\text{-}Ctx\text{-}I}]$$

$$\frac{\forall i \in I : \langle \mathbb{V}; G'_i\rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G''_i\rangle \quad \mathtt{q} \notin \mathrm{subject}(\alpha) \quad \alpha \neq t}{\langle \mathbb{V}; \mathtt{p}\rightsquigarrow\mathtt{q}{:}j\left\{\mathtt{m_i}(S_i)\{\mathcal{A}_i\}.G'_i\right\}_{i\in I}\rangle \xrightarrow{\alpha} \langle \mathbb{V}'; \mathtt{p}\rightsquigarrow\mathtt{q}{:}j\left\{\mathtt{m_i}(S_i)\{\mathcal{A}_i\}.G''_i\right\}_{i\in I}\rangle}\,[\mathrm{GR\text{-}Ctx\text{-}II}]$$

Figure 7 Top: typing environment semantics. Bottom: timed global type semantics, where $\mathcal{A}_i = \delta_{\mathrm{O}i}, \lambda_{\mathrm{O}i}, \delta_{\mathrm{I}i}, \lambda_{\mathrm{I}i}$.

4.3 Relating Timed Global Types and Typing Environments

One of our main results is establishing an operational relationship between the semantics of timed global types and typing environments, ensuring the *correctness* of processes typed by environments that reflect timed global types. To accomplish this, we begin by assigning LTS semantics to timed global types.

Similar to that of typing environments, we define the LTS semantics for timed global types G over tuples of the form $\langle \mathbb{V}; G\rangle$, where $\mathbb{V}$ is a clock valuation. Additionally, we specify the subject of an action α as its responsible principal: $\mathrm{subject}(s{:}\mathtt{p}!\mathtt{q}{:}\mathtt{m}) = \mathrm{subject}(s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}) = \{\mathtt{p}\}$, and $\mathrm{subject}(t) = \emptyset$.

▶ **Definition 10** (Timed Global Type Reduction). *The* timed global type transition $\xrightarrow{\alpha}$ *is inductively defined by the rules in Fig. 7 (bottom). We denote* $\langle \mathbb{V}; G\rangle \longrightarrow \langle \mathbb{V}'; G'\rangle$ *if there exists* α *such that* $\langle \mathbb{V}; G\rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G'\rangle$, $\langle \mathbb{V}; G\rangle \longrightarrow$ *if there exists* $\langle \mathbb{V}'; G'\rangle$ *such that* $\langle \mathbb{V}; G\rangle \longrightarrow \langle \mathbb{V}'; G'\rangle$, *and* $\longrightarrow^*$ *as the transitive and reflexive closure of* $\longrightarrow$.

In Fig. 7 (bottom), the (highlighted) changes from the standard global type reduction rules [11] focus on time. Rule [GR-T] accounts for the passage of time by incrementing the clock valuation. Rules [GR-⊕] and [GR-&] model the sending and receiving of messages within specified clock constraints, respectively. Both rules also require the adjustment of the clock valuation using the reset predicate. Rule [GR-μ] handles recursion. Finally, rules [GR-Ctx-i] and [GR-Ctx-ii] allow reductions of (intermediate) global types causally independent of their prefixes. Note that the execution of any timed global type transition always starts with an initial clock valuation $\mathbb{V}^0$, i.e. all clocks in $\mathbb{V}$ are set to 0.

We are now ready to establish a *new* relationship, *association*, between timed global types and typing environments. This association, which is more general than projection (Def. 6) by incorporating subtyping $\leqslant$ (Def. 5), plays a crucial role in formulating the typing rules (§ 4.4) and demonstrating the properties of typed processes (§ 4.5).

▶ **Definition 11** (Association). *A typing environment Γ is* associated with *a timed global type $\langle \mathbb{V}; G\rangle$ for a multiparty session s, written $\langle \mathbb{V}; G\rangle \sqsubseteq_s \Gamma$, iff Γ can be split into three (possibly empty) sub-environments $\Gamma = \Gamma_G, \Gamma_\triangle, \Gamma_{\mathbf{end}}$ where:*

1. *Γ_G is associated with $\langle \mathbb{V}; G\rangle$ for s, provided as:* (i) $\mathrm{dom}(\Gamma_G) = \{s[\mathtt{p}] \mid \mathtt{p} \in \mathrm{roles}(G)\}$*;* (ii) $\forall s[\mathtt{p}] \in \mathrm{dom}(\Gamma_G) : \Gamma_G(s[\mathtt{p}]) = (\mathbb{V}_\mathtt{p}, T_\mathtt{p})$*;* (iii) $\forall \mathtt{p} \in \mathrm{roles}(G) : G{\upharpoonright}\mathtt{p} \leqslant T_\mathtt{p}$*; and* (iv) $\mathbb{V} = \sqcup_{\mathtt{p}\in\mathrm{roles}(G)} \mathbb{V}_\mathtt{p}$ *(recall that $\sqcup$ is an overriding union).*
2. *$\Gamma_\triangle$ is associated with G for s, given as follows:*
   (i) $\mathrm{dom}(\Gamma_\triangle) = \{s[\mathtt{p}] \mid \mathtt{p} \in \mathrm{roles}(G)\}$*;*
   (ii) $\forall s[\mathtt{p}] \in \mathrm{dom}(\Gamma_\triangle) : \Gamma_\triangle(s[\mathtt{p}]) = \mathcal{M}_\mathtt{p}$*;*
   (iii) *if $G = \mathbf{end}$ or $G = \mu\mathbf{t}.G'$, then $\forall s[\mathtt{p}] \in \mathrm{dom}(\Gamma_\triangle) : \Gamma_\triangle(s[\mathtt{p}]) = \oslash$;*
   (iv) *if $G = \mathtt{p}{\to}\mathtt{q}{:}\, \{\mathtt{m}_\mathtt{i}(S_i)\{\delta_{Oi}, \lambda_{Oi}, \delta_{Ii}, \lambda_{Ii}\}.G_i\}_{i\in I}$, then* (a1) $\mathtt{q} \notin \mathrm{receivers}(\Gamma_\triangle(s[\mathtt{p}]))$, *and* (a2) $\forall i \in I$*: $\Gamma_\triangle$ is associated with G_i for s;*
   (v) *if $G = \mathtt{p}{\rightsquigarrow}\mathtt{q}{:}j\ \{\mathtt{m}_\mathtt{i}(S_i)\{\delta_{Oi}, \lambda_{Oi}, \delta_{Ii}, \lambda_{Ii}\}.G_i\}_{i\in I}$, then* (b1) $\Gamma_\triangle(s[\mathtt{p}]) = \mathtt{q}!\mathtt{m}_j(S'_j)\cdot\mathcal{M}$ *with $S'_j \leqslant S_j$, and* (b2) $\Gamma_\triangle[s[\mathtt{p}] \mapsto \mathcal{M}]$ *is associated with G_j for s.*
3. $\forall s[\mathtt{p}] \in \mathrm{dom}(\Gamma_{\mathbf{end}}) : \Gamma_{\mathbf{end}}(s[\mathtt{p}]) = \oslash; (\mathbb{V}_\mathtt{p}, \mathbf{end})$.

The association $\cdot \sqsubseteq_. \cdot$ is a binary relation over timed global types $\langle \mathbb{V}; G\rangle$ and typing environments Γ, parameterised by multiparty sessions s. There are three requirements for the association: (1) the typing environment Γ must include two entries for each role of the global type G in s: one of timed session type and another of queue type; (2) the timed session type entries in Γ reflect $\langle \mathbb{V}; G\rangle$ by ensuring that: **a.** they align with the projections of G via subtyping, and **b.** their clock valuations match $\mathbb{V}$; (3) the queue type entries in Γ correspond to the transmissions en route in G. Note that $\Gamma_{\mathbf{end}}$ is specifically used to associate typing environments and **end**-types $\langle \mathbb{V}; \mathbf{end}\rangle$, as in this case, both Γ_G and $\Gamma_\triangle$ are empty.

▶ **Example 12.** Consider the timed global type $\langle \{C_{\mathtt{Sen}} = 0, C_{\mathtt{Sat}} = 0, C_{\mathtt{Ser}} = 0\}; G_{\mathrm{data}}\rangle$, where G_{data} is from Ex. 7, and a typing environment $\Gamma_{\mathrm{data}} = \Gamma_{G_{\mathrm{data}}}, \Gamma_{\triangle_{\mathrm{data}}}$, where:

$$\begin{aligned}
\Gamma_{G_{\mathrm{data}}} = {} & s[\mathtt{Sen}]{:}(\{C_{\mathtt{Sen}} = 0\}, \mathtt{Sat}{\oplus}\mathtt{Data}\{6 \le C_{\mathtt{Sen}} \le 7, C_{\mathtt{Sen}} := 0\}), \\
& s[\mathtt{Sat}]{:}(\{C_{\mathtt{Sat}} = 0\}, \mathtt{Sen}\&\begin{Bmatrix}\mathtt{Data}\{6 \le C_{\mathtt{Sat}} \le 7, \emptyset\}.\mathtt{Ser}{\oplus}\mathtt{Data}\{6 \le C_{\mathtt{Sat}} \le 7, C_{\mathtt{Sat}} := 0\} \\ \mathtt{fail}\{6 \le C_{\mathtt{Sat}} \le 7, \emptyset\}.\mathtt{Ser}{\oplus}\mathtt{fatal}\{6 \le C_{\mathtt{Sat}} \le 7, C_{\mathtt{Sat}} := 0\}\end{Bmatrix}), \\
& s[\mathtt{Ser}]{:}(\{C_{\mathtt{Ser}} = 0\}, \mathtt{Sat}\&\begin{Bmatrix}\mathtt{Data}\{6 \le C_{\mathtt{Ser}} \le 7, C_{\mathtt{Ser}} := 0\} \\ \mathtt{fatal}\{6 \le C_{\mathtt{Ser}} \le 7, C_{\mathtt{Ser}} := 0\}\end{Bmatrix}) \\
\Gamma_{\triangle_{\mathrm{data}}} = {} & s[\mathtt{Sen}]{:}\oslash, s[\mathtt{Sat}]{:}\oslash, s[\mathtt{Ser}]{:}\oslash
\end{aligned}$$

Γ_{data} is associated with $\langle \{C_{\mathtt{Sen}} = 0, C_{\mathtt{Sat}} = 0, C_{\mathtt{Ser}} = 0\}; G_{\mathrm{data}}\rangle$ for s, which can be formally verified by ensuring that Γ_{data} satisfies all conditions outlined in Def. 11.

We establish the operational correspondence between a timed global type and its associated typing environment, our main result for timed multiparty session types, through two theorems: Thm. 13 demonstrates that every possible reduction of a typing environment is mirrored by a corresponding action in reductions of the associated timed global type, while Thm. 14 indicates that the reducibility of a timed global type is equivalent to its associated environment.

▶ **Theorem 13** (Completeness of Association). *Given associated timed global type $\langle \mathbb{V}; G\rangle$ and typing environment Γ: $\langle \mathbb{V}; G\rangle \sqsubseteq_s \Gamma$. If $\Gamma \xrightarrow{\alpha} \Gamma'$, then there exists $\langle \mathbb{V}'; G'\rangle$ such that $\langle \mathbb{V}; G\rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G'\rangle$ and $\langle \mathbb{V}'; G'\rangle \sqsubseteq_s \Gamma'$.*

▶ **Theorem 14** (Soundness of Association). *Given associated timed global type $\langle \mathbb{V}; G\rangle$ and typing environment Γ: $\langle \mathbb{V}; G\rangle \sqsubseteq_s \Gamma$. If $\langle \mathbb{V}; G\rangle \rightarrow$, then there exists α', $\mathbb{V}'$, $\langle \mathbb{V}''; G''\rangle$, Γ', and Γ'', such that $\langle \mathbb{V}'; G\rangle \sqsubseteq_s \Gamma'$, $\langle \mathbb{V}'; G\rangle \xrightarrow{\alpha'} \langle \mathbb{V}''; G''\rangle$, $\Gamma' \xrightarrow{\alpha'} \Gamma''$, and $\langle \mathbb{V}''; G''\rangle \sqsubseteq_s \Gamma''$.*

▶ Remark 15. We formulate a soundness theorem that does not mirror the completeness theorem, differing from prior work such as [11]. This choice stems from our reliance on subtyping (Def. 5), notably [Sub-$\oplus$]. In our framework, a timed local type in the typing environment might offer fewer selection branches compared to the corresponding projected timed local type. Consequently, certain sending actions with their associated clock valuations may remain uninhabited within the timed global type. Consider, e.g. a timed global type:

$$\langle \mathbb{V}_{\mathrm{r}}; G_{\mathrm{r}} \rangle = \langle \{C_{\mathtt{p}} = 3, C_{\mathtt{q}} = 3\}; \mathtt{p}\to\mathtt{q}: \left\{ \begin{matrix} \mathtt{m}_1\{0 \le C_{\mathtt{p}} \le 1, \emptyset, 1 \le C_{\mathtt{q}} \le 2, \emptyset\}.\mathbf{end} \\ \mathtt{m}_2\{2 \le C_{\mathtt{p}} \le 4, \emptyset, 5 \le C_{\mathtt{q}} \le 6, \emptyset\}.\mathbf{end} \end{matrix} \right\} \rangle$$

An associated typing environment Γ_{r} may have:

$$\Gamma_{\mathrm{r}}(s[\mathtt{p}]) = (\{C_{\mathtt{p}} = 3\}, \mathtt{q}\oplus\mathtt{m}_1\{0 \le C_{\mathtt{p}} \le 1, \emptyset\}.\mathbf{end}); \oslash \geqslant (\{C_{\mathtt{p}} = 3\}, \mathtt{q}\oplus \left\{ \begin{matrix} \mathtt{m}_1\{0 \le C_{\mathtt{p}} \le 1, \emptyset\}.\mathbf{end} \\ \mathtt{m}_2\{2 \le C_{\mathtt{p}} \le 4, \emptyset\}.\mathbf{end} \end{matrix} \right\}); \oslash$$

While the timed global type $\langle \mathbb{V}_{\mathrm{r}}; G_{\mathrm{r}} \rangle$ might transition through $s{:}\mathtt{p}!\mathtt{q}{:}\mathtt{m}_2$, the associated environment Γ_{r} cannot. Nevertheless, our soundness theorem *adequately* guarantees communication safety (communication matches) via association.

4.4 Affine Timed Multiparty Session Typing System

We now present a typing system for ATMP, which relies on *typing judgments* of the form:

$$\Theta \cdot \Gamma \vdash P \quad \text{(with } \Theta \text{ omitted when empty)}$$

This judgement indicates that the process P adheres to the usage of its variables and channels as specified in Γ (Def. 8), guided by the process types in Θ (Def. 8). Our typing system is defined inductively by the typing rules shown in Fig. 8, with channels annotated for convenience, especially those bound by process definitions and restrictions.

The innovations (highlighted) in Fig. 8 primarily focus on typing processes with time, timeout failures, message queues, and using association (Def. 11) to enforce session restrictions.

Standard from [37]: Rule [T-X] retrieves process variables. Rule [T-Sub] applies subtyping within a singleton typing environment $c{:}(\mathbb{V}, T)$. Rule [T-end] introduces a predicate end$(\cdot)$ for typing environments, signifying the termination of all endpoints. This predicate is used in [T-$\mathbf{0}$] to type an inactive process $\mathbf{0}$. Rules [T-**def**] and [T-X-Call] deal with recursive processes declarations and calls, respectively. Rule [T-$|$] partitions the typing environment into two, each dedicated to typing one sub-process.

Session Restriction: Rule [T-ν-G] depends on a typing environment associated with a timed global type in a given session s to validate session restrictions.

Delay: Rule [T-δ] ensures the typedness of time-consuming delay $\mathbf{delay}(\delta) \,.\, P$ by checking every deterministic delay $\mathbf{delay}(t) \,.\, P$ with t as a possible solution to δ. Rule [T-t] types a deterministic delay $\mathbf{delay}(t) \,.\, P$ by adjusting the clock valuations in the environment used to type P. Here, $\Gamma + t$ denotes the typing environment obtained from Γ by increasing the associated clock valuation in each entry by t.

Timed Branching and Selection: Rules [T-&] and [T-$\oplus$] are for timed branching and selection, respectively. We elaborate on [T-&], as [T-$\oplus$] is its dual. The first premise in [T-&] specifies a time interval $[\mathbb{V}, \mathbb{V} + \mathfrak{n}]$ within which the message must be received, in accordance with each δ_i. The last premise requires that each continuation process be well-typed against the continuation of the type in all possible typing environments where the time falls between $[\mathbb{V}, \mathbb{V} + \mathfrak{n}]$. Here, the clock valuation $\mathbb{V}$ is reset based on each λ_i. The remaining premises stipulate that the clock valuation $\mathbb{V}'_i$ of each delegated receiving session must satisfy δ'_i, and that c is typed.

Try-Catch, Cancellation, and Kill: Rules [T-Try], [T-Cancel], and [T-Kill] pertain to try-catch, cancellation, and kill processes, respectively, analogous to the corresponding rules in [28]. [T-Cancel] is responsible for generating a kill process at its declared session. [T-Kill]

$$\frac{\Theta(X) = (\mathbb{V}_1, T_1), \ldots, (\mathbb{V}_n, T_n)}{\Theta \vdash X{:}(\mathbb{V}_1, T_1), \ldots, (\mathbb{V}_n, T_n)}\;[\text{T-}X] \qquad \frac{\forall i \in 1..n \quad c_i{:}(\mathbb{V}_i, T_i) \vdash c_i{:}(\mathbb{V}'_i, \mathbf{end})}{\text{end}(c_1{:}(\mathbb{V}_1, T_1), \ldots, c_n{:}(\mathbb{V}_n, T_n))}\;[\text{T-end}]$$

$$\frac{(\mathbb{V}, T) \leqslant (\mathbb{V}', T')}{c{:}(\mathbb{V}, T) \vdash c{:}(\mathbb{V}', T')}\;[\text{T-Sub}] \qquad \frac{\begin{array}{c}\Theta, X{:}(\mathbb{V}_1, T_1), \ldots, (\mathbb{V}_n, T_n) \cdot x_1{:}(\mathbb{V}_1, T_1), \ldots, x_n{:}(\mathbb{V}_n, T_n) \vdash P \\ \Theta, X{:}(\mathbb{V}_1, T_1), \ldots, (\mathbb{V}_n, T_n) \cdot \Gamma \vdash Q\end{array}}{\Theta \cdot \Gamma \vdash \mathbf{def}\ X(x_1{:}(\mathbb{V}_1, T_1), \ldots, x_n{:}(\mathbb{V}_n, T_n)) = P\ \mathbf{in}\ Q}\;[\text{T-}\mathbf{def}]$$

$$\frac{\text{end}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}}\;[\text{T-}\mathbf{0}] \qquad \frac{\Theta \vdash X{:}(\mathbb{V}_1, T_1), \ldots, (\mathbb{V}_n, T_n) \quad \text{end}(\Gamma_0) \quad \forall i \in 1..n \quad \Gamma_i \vdash c_i{:}(\mathbb{V}_i, T_i)}{\Theta \cdot \Gamma_0, \Gamma_1, \ldots, \Gamma_n \vdash X\langle c_1, \ldots, c_n\rangle}\;[\text{T-}X\text{-Call}]$$

$$\frac{\forall t \text{ s.t. } \models \delta[t/C] : \Theta \cdot \Gamma \vdash \mathbf{delay}(t)\,.\,P}{\Theta \cdot \Gamma \vdash \mathbf{delay}(\delta)\,.\,P}\;[\text{T-}\delta] \qquad \frac{\Theta \cdot \Gamma + t \vdash P}{\Theta \cdot \Gamma \vdash \mathbf{delay}(t)\,.\,P}\;[\text{T-}t]$$

$$\frac{\begin{array}{c}\forall i{\in}I \ \ \forall t : t \leq \mathfrak{n} \Longrightarrow \mathbb{V} + t \models \delta_i \\ \Gamma_1 \vdash c{:}(\mathbb{V}, \mathtt{q}\&\{\mathtt{m_i}(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i\in I}) \quad \forall i{\in}I : S_i = (\delta'_i, T'_i) \quad \mathbb{V}'_i \models \delta'_i \\ \forall i{\in}I\ \forall t \leq \mathfrak{n} : \Theta \cdot \Gamma + t, y_i{:}(\mathbb{V}'_i, T'_i), c{:}(\mathbb{V} + t[\lambda_i \mapsto 0], T_i) \vdash P_i\end{array}}{\Theta \cdot \Gamma, \Gamma_1 \vdash c^{\mathfrak{n}}[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(y_i).P_i}\;[\text{T-}\&]$$

$$\frac{\begin{array}{c}\forall t : t \leq \mathfrak{n} \Longrightarrow \mathbb{V} + t \models \delta \\ \Gamma_1 \vdash c{:}(\mathbb{V}, \mathtt{q}\oplus\{\mathtt{m}(S)\{\delta, \lambda\}.T\}) \quad S = (\delta', T') \quad \Gamma_2 \vdash d{:}(\mathbb{V}', T') \quad \mathbb{V}' \models \delta' \\ \forall t \leq \mathfrak{n} : \Theta \cdot \Gamma + t, c{:}(\mathbb{V} + t[\lambda \mapsto 0], T) \vdash P\end{array}}{\Theta \cdot \Gamma, \Gamma_1, \Gamma_2 \vdash c^{\mathfrak{n}}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P}\;[\text{T-}\oplus]$$

$$\frac{\Theta \cdot \Gamma_1 \vdash P_1 \quad \Theta \cdot \Gamma_2 \vdash P_2}{\Theta \cdot \Gamma_1, \Gamma_2 \vdash P_1 \mid P_2}\;[\text{T-}|] \qquad \frac{\text{end}(\Gamma) \quad n \geq 0}{\Theta \cdot \Gamma, s[\mathtt{p}_1]{:}\tau_1, \ldots, s[\mathtt{p}_n]{:}\tau_n \vdash s\lightning}\;[\text{T-Kill}]$$

$$\frac{\Theta \cdot \Gamma \vdash P \quad \text{subjP}(P) = \{c\} \quad \Theta \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \mathsf{try}\ \ P\ \ \mathsf{catch}\ \ Q}\;[\text{T-Try}] \qquad \frac{\Theta \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma, s[\mathtt{p}]{:}\tau \vdash \mathsf{cancel}(s[\mathtt{p}])\,.\,Q}\;[\text{T-Cancel}]$$

$$\frac{}{\Theta \cdot \Gamma \vdash \mathtt{timeout}[P]}\;[\text{T-Failed}] \qquad \frac{\langle \mathbb{V}; G\rangle \sqsubseteq_s \Gamma' \quad s \notin \Gamma \quad \Theta \cdot \Gamma, \Gamma' \vdash P}{\Theta \cdot \Gamma \vdash \left(\nu s{:}\Gamma'\right)\, P}\;[\text{T-}\nu\text{-}G]$$

$$\frac{\text{end}(\Gamma)}{\Theta \cdot \Gamma, s[\mathtt{p}]{:}\oslash \vdash s[\mathtt{p}] \blacktriangleright \epsilon}\;[\text{T-}\epsilon] \qquad \frac{\Theta \cdot \Gamma \vdash s[\mathtt{p}] \blacktriangleright \sigma \quad S = (\delta, T) \quad \mathbb{V} \models \delta \quad \Gamma' \vdash s'[\mathtt{r}]{:}(\mathbb{V}, T)}{\Theta \cdot \Gamma[s[\mathtt{p}] \mapsto \mathtt{q}!\mathtt{m}(S)\cdot\Gamma(s[\mathtt{p}])], \Gamma' \vdash s[\mathtt{p}] \blacktriangleright \mathtt{q}!\mathtt{m}\langle s'[\mathtt{r}]\rangle \cdot \sigma}\;[\text{T-}\sigma]$$

Figure 8 ATMP typing rules.

types a kill process arising during reductions: it involves broadcasting the cancellation of $s[\mathtt{p}]$ to all processes that belong to s. [T-Try] handles a **try-catch** process **try** P **catch** Q by ensuring that the **try** process P and the **catch** process Q maintain consistent session typing. Additionally, P cannot be a queue or parallel composition, as indicated by $\text{subjP}(P) = \{c\}$.

Timeout Failure: Rule [T-Failed] indicates that a process raising timeout failure can be typed by *any* typing environment.

Queue: Rules [T-ϵ] and [T-σ] concern the typing of queues. [T-ϵ] types an empty queue under an ended typing environment, while [T-σ] types a non-empty queue by inserting a message type into Γ. This insertion may either prepend the message to an existing queue type in Γ or add a queue-typed entry to Γ if not present.

▶ **Example 16.** Take the typing environment Γ_{data} from Ex. 12, along with the processes $Q_{\mathtt{Sen}}$, $Q_{\mathtt{Sat}}$, $Q_{\mathtt{Ser}}$ from Ex. 4. Verifying the typing of $Q_{\mathtt{Sen}} \mid Q_{\mathtt{Sat}} \mid Q_{\mathtt{Ser}}$ by Γ_{data} is easy. Moreover, since Γ_{data} is associated with a timed global type $\langle\{C_{\mathtt{Sen}} = 0, C_{\mathtt{Sat}} = 0, C_{\mathtt{Ser}} = 0\}; G_{\text{data}}\rangle$ for session s (as demonstrated in Ex. 12), i.e. $\langle\{C_{\mathtt{Sen}} = 0, C_{\mathtt{Sat}} = 0, C_{\mathtt{Ser}} = 0\}; G_{\text{data}}\rangle \sqsubseteq_s \Gamma_{\text{data}}$, following [T-$\nu$-$G$], $Q_{\mathtt{Sen}} \mid Q_{\mathtt{Sat}} \mid Q_{\mathtt{Ser}}$ is closed under Γ_{data}, i.e. $\vdash (\nu s{:}\Gamma_{\text{data}})\, Q_{\mathtt{Sen}} \mid Q_{\mathtt{Sat}} \mid Q_{\mathtt{Ser}}$.

4.5 Typed Process Properties

We demonstrate that processes typed by the ATMP typing system exhibit the desirable properties: *subject reduction* (Thm. 17), *session fidelity* (Thm. 21), and *deadlock-freedom* (Thm. 24).

Subject Reduction ensures the preservation of well-typedness of processes during reductions. Specifically, it states that if a well-typed process P reduces to P', this reduction is reflected in the typing environment Γ used to type P. Notably, in our subject reduction theorem, P is constructed from a timed global type, i.e. typed by an environment associated with a timed global type, and this construction approach persists as an invariant property throughout reductions. Furthermore, the theorem does not require P to contain only a single session; instead, it includes all restricted sessions in P, ensuring that reductions on these sessions uphold their respective restrictions. This enforcement is facilitated by rule [T-ν-G] in Fig. 8.

▶ **Theorem 17** (Subject Reduction). *Assume $\Theta \cdot \Gamma \vdash P$ where $\forall s \in \Gamma : \exists\langle \mathbb{V}; G\rangle : \langle \mathbb{V}; G\rangle \sqsubseteq_s \Gamma_s$. If $P \to P'$, then $\exists \Gamma'$ such that $\Gamma \to^* \Gamma'$, $\Theta \cdot \Gamma' \vdash P'$, and $\forall s \in \Gamma' : \exists\langle \mathbb{V}'; G'\rangle : \langle \mathbb{V}'; G'\rangle \sqsubseteq_s \Gamma'_s$.*

▶ **Corollary 18** (Type Safety). *Assume $\emptyset \cdot \emptyset \vdash P$. If $P \to^* P'$, then P' has no communication error.*

▶ **Example 19.** Take the typed process $Q_{\mathtt{Sen}} \mid Q_{\mathtt{Sat}} \mid Q_{\mathtt{Ser}}$ and the typing environment Γ_{data} from Exs. 4, 12, and 16. After a reduction using [R-Det], $Q_{\mathtt{Sen}} \mid Q_{\mathtt{Sat}} \mid Q_{\mathtt{Ser}}$ transitions to $\mathbf{delay}(6.5)\,.\,Q'_{\mathtt{Sen}} \,|\, s[\mathtt{Sen}] \blacktriangleright \epsilon \,|\, \mathbf{delay}(6)\,.\,Q'_{\mathtt{Sat}} \,|\, s[\mathtt{Sat}] \blacktriangleright \epsilon \,|\, \mathbf{delay}(6)\,.\,Q'_{\mathtt{Ser}} \,|\, s[\mathtt{Ser}] \blacktriangleright \epsilon = Q_2$, which remains typable by Γ_{data} ($\Gamma_{\text{data}} \to^* \Gamma_{\text{data}}$). Then, applying [R-Time], Q_2 evolves to $\Psi_{6.5}(Q_2)$, typed as $\Gamma_{\text{data}}+6.5$, derived from $\Gamma_{\text{data}} \xrightarrow{6.5} \Gamma_{\text{data}}+6.5$. Further reduction through [R-Fail] leads $\Psi_{6.5}(Q_2)$ to $Q'_{\mathtt{Sen}} \,|\, s[\mathtt{Sen}] \blacktriangleright \epsilon \,|\, s\lightning \,|\, s[\mathtt{Sat}] \blacktriangleright \epsilon \,|\, \Psi_{0.5}(Q'_{\mathtt{Ser}}) \,|\, s[\mathtt{Ser}] \blacktriangleright \epsilon = Q_3$, typable by $\Gamma_{\text{data}} + 6.5$. Later, via [C-Cat], Q_3 reduces to $\mathsf{cancel}(s[\mathtt{Sen}]) \,|\, s[\mathtt{Sen}] \blacktriangleright \epsilon \,|\, s\lightning \,|\, s[\mathtt{Sat}] \blacktriangleright \epsilon \,|\, \Psi_{0.5}(Q'_{\mathtt{Ser}}) \,|\, s[\mathtt{Ser}] \blacktriangleright \epsilon$, which can be typed by Γ''_{data}, obtained from $\Gamma_{\text{data}} + 6.5 \xrightarrow{s:\mathtt{Sen!Sat:Data}} \cdot \xrightarrow{s:\mathtt{Sat,Sen:Data}} \Gamma''_{\text{data}}$.

Session Fidelity states the converse implication of subject reduction: if a process P is typed by Γ and Γ can reduce, then P can simulate at least one of the reductions performed by Γ – although not necessarily all such reductions, as Γ over-approximates the behavior of P. Consequently, we can infer P's behaviour from that of Γ. However, this result does not hold for certain well-typed processes, such as those that get trapped in recursion loops like $\mathbf{def}\ X(...) = X\ \mathbf{in}\ X$, or deadlock due to interactions across multiple sessions [8]. To address this, similarly to [37] and most session type works, we establish session fidelity specifically for processes featuring guarded recursion and implementing a single multiparty session as a parallel composition of one sub-process per role. The formalisation of session fidelity is provided in Thm. 21, building upon the concepts introduced in Def. 20.

▶ **Definition 20** (From [37]). *Assume $\emptyset \cdot \Gamma \vdash P$. We say that P:*

1. has guarded definitions *if and only if in each process definition in P of the form $\mathbf{def}\ X(x_1{:}(\mathbb{V}_1, T_1), ..., x_n{:}(\mathbb{V}_n, T_n)) = Q\ \mathbf{in}\ P'$, for all $i \in 1...n$, $T_i \not\leqslant \mathbf{end}$ implies that a call $Y\langle ..., x_i, ...\rangle$ can only occur in Q as a subterm of $x_i{}^{\mathtt{n}}[\mathtt{q}]\sum_{j\in J}\mathtt{m}_j(y_j).P_j$ or $x_i{}^{\mathtt{n}}[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P''$ (i.e. after using x_i for input or output);*
2. only plays role $\mathtt{p}$ in s, by Γ, *if and only if* (i) *P has guarded definitions;* (ii) $\mathrm{fv}(P) = \emptyset$; (iii) $\Gamma = \Gamma_0, s[\mathtt{p}]{:}\tau$ *with* $\tau \not\leqslant (\mathbb{V}, \mathbf{end})$ *and* $\mathrm{end}(\Gamma_0)$; (iv) *in all subterms* $(\nu s'{:}\Gamma')\,P'$ *of P, we have* $\Gamma' \leqslant s'[\mathtt{p}']{:}\oslash; (\mathbb{V}', \mathbf{end})$ *or* $\Gamma' \leqslant s'[\mathtt{p}']{:}(\mathbb{V}', \mathbf{end})$ *(for some $\mathtt{p}', \mathbb{V}'$).*

We say "P only plays role $\mathtt{p}$ in s*" if and only if $\exists\Gamma : \emptyset \cdot \Gamma \vdash P$, and item 2 holds.*

In Def. 20, item 1 describes guarded recursion for processes, while item 2 specifies a process limited to playing exactly *one* role within *one* session, preventing an ensemble of such processes from deadlocking by waiting for each other on multiple sessions.

We proceed to present our session fidelity result, taking kill processes into account. We denote $Q\lightning$ to indicate that Q consists only of a parallel composition of kill processes. Similar to subject reduction (Thm. 17), our session fidelity relies on a typing environment associated with a timed global type for a specific session s to type the process, ensuring the fulfilment of single-session requirements (Def. 20) and maintaining invariance during reductions.

▶ **Theorem 21** (Session Fidelity). *Assume* $\emptyset \cdot \Gamma \vdash P$, *with* $\langle \mathbb{V}; G \rangle \sqsubseteq_s \Gamma$, $P \equiv \Pi_{\mathtt{p} \in I} P_{\mathtt{p}} \mid Q \lightning$, *and* $\Gamma = \bigcup_{\mathtt{p} \in I} \Gamma_{\mathtt{p}} \cup \Gamma_0$, *such that* $\emptyset \cdot \Gamma_0 \vdash Q \lightning$, *and for each* $P_{\mathtt{p}}$: (1) $\emptyset \cdot \Gamma_{\mathtt{p}} \vdash P_{\mathtt{p}}$, *and* (2) *either* $P_{\mathtt{p}} \equiv \mathbf{0}$, *or* $P_{\mathtt{p}}$ *only plays role* $\mathtt{p}$ *in* s, *by* $\Gamma_{\mathtt{p}}$. *Then,* $\Gamma \rightarrow_s$ *implies* $\exists \Gamma', \langle \mathbb{V}'; G' \rangle, P'$ *such that* $\Gamma \rightarrow_s \Gamma'$, $P \rightarrow^* P'$, *and* $\emptyset \cdot \Gamma' \vdash P'$, *with* $\langle \mathbb{V}'; G' \rangle \sqsubseteq_s \Gamma'$, $P' \equiv \Pi_{\mathtt{p} \in I} P'_{\mathtt{p}} \mid Q' \lightning$, *and* $\Gamma' = \bigcup_{\mathtt{p} \in I} \Gamma'_{\mathtt{p}} \cup \Gamma'_0$ *such that* $\emptyset \cdot \Gamma'_0 \vdash Q' \lightning$, *and for each* $P'_{\mathtt{p}}$: (1) $\emptyset \cdot \Gamma'_{\mathtt{p}} \vdash P'_{\mathtt{p}}$, *and* (2) *either* $P'_{\mathtt{p}} \equiv \mathbf{0}$, *or* $P'_{\mathtt{p}}$ *only plays role* $\mathtt{p}$ *in* s, *by* $\Gamma'_{\mathtt{p}}$.

▶ **Example 22.** Consider the processes $Q_{\mathtt{Sen}}$, $Q_{\mathtt{Sat}}$, $Q_{\mathtt{Ser}}$ from Ex. 4, the process Q_3 from Ex. 19, and the typing environment Γ_{data} from Ex. 12. $Q_{\mathtt{Sen}}$, $Q_{\mathtt{Sat}}$, and $Q_{\mathtt{Ser}}$ only play roles $\mathtt{Sen}$, $\mathtt{Sat}$, and $\mathtt{Ser}$, respectively, in s, which can be easily verified. As demonstrated in Ex. 19, Q_3 is typed by $\Gamma_{\text{data}} + 6.5$, satisfying all prerequisites specified in Thm. 21. Consequently, given $\Gamma_{\text{data}} + 6.5 \xrightarrow{s:\mathtt{Sen}!\mathtt{Sat}:\mathtt{Data}} \Gamma'_{\text{data}}$, there exists Q_4, resulting from $Q_3 \rightarrow Q_4$ via [R-Out], such that Γ'_{data} can type Q_4, with Γ'_{data} and Q_4 fulfilling the single session requirements of session fidelity.

Deadlock-Freedom ensures that a process can always either progress via reduction or terminate properly. In our system, where time can be infinitely reduced and session killings may occur during reductions, deadlock-freedom implies that if a process cannot undergo any further instantaneous (communication) reductions, and if any subsequent time reduction leaves it unchanged, then it contains only inactive or kill sub-processes. This desirable runtime property is guaranteed by processes constructed from timed global types. We formalise the property in Def. 23, and conclude, in Thm. 24, that a typed ensemble of processes interacting on a single session, restricted by a typing environment Γ associated with a timed global type $\langle \mathbb{V}^0; G \rangle$, is deadlock-free.

▶ **Definition 23** (Deadlock-Free Process). *P is* deadlock-free *if and only if* $P \rightarrow^* P' \not\rightarrow$ *and* $\forall t \geq 0 : \Psi_t(P') = P'$ *(recall that* $\Psi_t(\cdot)$ *is a time-passing function defined in Fig. 4) implies* $P' \equiv \mathbf{0} \mid \Pi_{i \in I} s_i \lightning$.

▶ **Theorem 24** (Deadlock-Freedom). *Assume* $\emptyset \cdot \emptyset \vdash P$, *where* $P \equiv (\nu s{:}\Gamma)\, \Pi_{\mathtt{p} \in \text{roles}(G)} P_{\mathtt{p}}$, $\langle \mathbb{V}^0; G \rangle \sqsubseteq_s \Gamma$, *and each* $P_{\mathtt{p}}$ *is either* $\mathbf{0}$ *(up to* $\equiv$*), or only plays* $\mathtt{p}$ *in* s. *Then,* P *is deadlock-free.*

▶ **Example 25.** Given the processes $Q_{\mathtt{Sen}}$, $Q_{\mathtt{Sat}}$, and $Q_{\mathtt{Ser}}$ from Ex. 4, along with the typing environment Γ_{data} from Ex. 12, $(\nu s{:}\Gamma_{\text{data}})\, Q_{\mathtt{Sen}} \mid Q_{\mathtt{Sat}} \mid Q_{\mathtt{Ser}}$ is deadlock-free.

5 Design and Implementation of `MultiCrusty`^{T}

In this section, we present our toolchain, `MultiCrusty`^{T}, a Rust implementation of ATMP. `MultiCrusty`^{T} is designed with two main goals: correctly cascading failure notifications, and effectively handling time constraints. To achieve the first goal, we use Rust's native `?`-operator along with optional types, inspired by [28]. For the second objective, we begin by discussing the key challenges encountered during implementation.

Challenge 1: Representation of Time Constraints. To handle asynchronous timed communications using ATMP, we define a time window (δ in ATMP) and a corresponding behaviour for each operation. Addressing this constraint involves two subtasks: creating and using clocks in Rust, and representing all clock constraints as shown in § 3. Rust allows the creation of virtual clocks that rely on the CPU's clock and provide nanosecond-level accuracy. Additionally, it is crucial to ensure that different behaviours can involve blocking or non-blocking communications, pre- or post-specific time tags, or adherence to specified time bounds.

Challenge 2: Enforcement of Time Constraints. To effectively enforce time windows, implementing reliable and accurate clocks and using them correctly is imperative. This requires addressing all cases related to time constraints properly: clocks may be considered unreliable if they skip ticks, do not strictly increase, or if the API for clock comparison does not yield results quickly enough. Enforcing time constraints in `MultiCrusty`^{T} involves using two libraries: the `crossbeam_channel` Rust library [9] for *asynchronous* messaging, and the Rust standard library `time` [39] for handling and comparing virtual clocks.

5.1 Time Bounds in `MultiCrusty`^{T}

Implementing Time Bounds To demonstrate the integration of time bounds in `MultiCrusty`^{T}, we consider the final interaction between `Sen` and `Sat` in Fig. 1b, specifically from `Sat`'s perspective: `Sat` sends a `Close` message between time units 5 and 6 (both inclusive), following clock $C_{\texttt{Sat2}}$, which is not reset afterward.

In `MultiCrusty`^{T}, we define the `Send` type for message transmission, incorporating various parameters to specify requirements as `Send<[parameter1],[parameter2],...>`. Assuming the (payload) type `Close` is defined, sending it using the `Send` type initiates with `Send<Close,...>`. If $C_{\texttt{Sat2}}$ is denoted as `'b'`, the clock `'b'` is employed for time constraints, expressed as `Send<Close,'b',...>`. Time bounds parameters in the `Send` type follow the clock declaration. In this case, both bounds are integers within the time window, resulting in the `Send` type being parameterised as `Send<Close,'b',0,true,1,false,...>`. Notably, bounds are integers due to the limitations of Rust's generic type parameters. To ensure that the clock `'b'` is not reset after triggering the `send` operation, we represent this with a whitespace char value in the `Send` type: `Send<Close,'b',0,true,1,false,' ',...>`. The last parameter, known as the *continuation*, specifies the operation following the sending of the integer. In this case, closing the connection is achieved with an `End` type. The complete sending operation is denoted as `Send<Close, 'b', 0, true, 1, false, ' ', End>`.

Similarly, the `Recv` type is instantiated as `Recv<Close,'b',0,true,1,false,' ',End>`. The inherent mirroring of `Send` and `Recv` reflects their dual compatibility. Figs. 2a and 2b provide an analysis of the functioning of `Send` and `Recv`, detailing their parameters and features. Generic type parameters preceded by `const` within `Send` and `Recv` types also serve as values, representing general type categories supported by Rust. This type-value duality facilitates easy verification during compilation, ensuring compatibility between communicating parties.

Enforcing Time Bounds It is crucial to rely on dependable clocks and APIs to enforce time constraints. Rust's standard library provides the time module [39], enabling developers to manage clocks and measure durations between events. This library, utilising the OS API, offers two clock types: `Instant` (monotonic but non-steady) and `SystemTime` (steady but non-monotonic). In `MultiCrusty`^{T}, the `Instant` type serves for both correctly prioritising event order and implementing virtual clocks. Virtual clocks are maintained through a dictionary (`HashMap` in Rust). Table 1 details the primitives provided by `MultiCrusty`^{T} for sending and receiving payloads, implementing branching, or closing connections. All primitives, except for `close`, require a specific `HashMap` of clocks to enforce time constraints.

Verifying Time Bounds Our `send` and `recv` primitives use a series of conditions to ensure the integrity of a time window. The verification process adopts a *divide-and-conquer* strategy, validating the left-hand side time constraint for each clock before assessing the right-hand side

Table 1 Primitives available in `MultiCrusty`^{T}.

`let s = s.send(p, clocks)?;`	If allowed by the time constraint compared to the given clock in `clocks`, sends a payload `p` on a channel `s` and assigns the continuation of the session (a new meshed channel) to a new variable `s`.
`let (p, s) = s.recv(clocks)?;`	If allowed by the time constraint compared to the given clock in `clocks`, receives a payload `p` on channel `s` and assigns the continuation of the session to a new variable `s`.
`s.close()`	Closes the channel `s` and returns a unit type.
`offer!(s, clocks, {` $\texttt{enum}_\texttt{i} :: \texttt{variant}_\texttt{k}(\texttt{e}) \Rightarrow \{\ldots\}_{k\in K}$ `})`	If allowed by the time constraint compared to the given clock in `clocks`, role i receives a choice as a message label on channel `s`, and, depending on the label value which should match one of the variants $\texttt{variant}_\texttt{k}$ of $\texttt{enum}_\texttt{i}$ ::, runs the related block of code.
`choose_X!(s, clocks, {` $\texttt{enum}_\texttt{i} :: \texttt{variant}_\texttt{k}(\texttt{e})\}_{i\in I}$ `)`	For role `X`, if allowed by the time constraint compared to the given clock in `clocks`, sends the chosen label, corresponding to $\texttt{variant}_\texttt{k}$ to all other roles.

```
1   type EndpointSerData = MeshedChannels<
2    Send<GetData, 'a', 5, true,5, true, ' ',
3     Recv<Data, 'a', 6, true, 7, true, 'a', End
    >>,
4     End,
5    RoleSat<RoleSat<RoleBroadcast>>,
6    NameSer>;
```

```
7   fn endpoint_data_ser(
8    s: EndpointSerData,
9    clocks: &mut HashMap<char, Instant>,
10   ) -> Result<(), Error> { [...]
11   let s = s.send(GetData {}, clocks)?;
12   let (_data, s) = s.recv(clocks)?;[...]}
```

Figure 9 Types (left) and primitives (right) for `Ser`.

constraint. The corresponding operation, whether sending or receiving a payload, is executed only after satisfying these conditions. This approach guarantees the effective enforcement of time constraints without requiring complex solver mechanisms.

5.2 Remote Data Implementation

Implementation of Server Fig. 9 explores our `MultiCrusty`^{T} implementation of `Ser` in the remote data protocol (Fig. 1b). Specifically, the left side of Fig. 9 delves into the `MeshedChannels` type, representing the behaviour of `Ser` in the first branch and encapsulating various elements. In `MultiCrusty`^{T}, the `MeshedChannels` type incorporates $n + 1$ parameters, where n is the count of roles in the protocol. These parameters include the role's name, $n - 1$ binary channels for interacting with other roles, and a stack dictating the sequence of binary channel usage. All types relevant to `Ser` are depicted in Fig. 9 (left).

The alias `EndpointSerData`, as indicated in Line 1, represents the `MeshedChannels` type. Binary types, defined in Lines 2–4, facilitate communication between `Ser`, `Sat`, and `Sen`. When initiating communication with `Sat`, `Ser` sends a `GetData` message in Line 2, receives a `Data` response, and ends communication on this binary channel. These operations use the clock `'a'` and adhere to time windows between 5 and 6 seconds for the first operation and between 6 and 7 seconds for the second. Clock `'a'` is reset only within the second operation. The order of operations is outlined in Line 5, where `Ser` interacts twice with `Sat` using `RoleSat` before initiating a choice with `RoleBroadcast`. Line 6 designates `Ser` as the owner of the `MeshedChannels` type. The behaviour of all roles in each branch can be specified similarly.

The right side of Fig. 9 illustrates the usage of `EndpointSerData` as an input type in the Rust function `endpoint_data_ser`. The function's output type, `Result<(), Error>`, indicates the utilization of affinity in Rust. In Line 11, variable `'s'`, of type `EndpointSerData`, attempts to send a contentless message `GetData`. The `send` function can return either a value resembling `EndpointSerData` or an `Error`. If the clock's time does not adhere to the time constraint displayed in Line 2 with respect to the clock `'a'` from the set of clocks `clocks`, an `Error` is raised. Similarly, in Line 12, `Ser` attempts to receive a message using the same set of clocks. Both `send` and `recv` functions verify compliance with time constraints by comparing the

```
global protocol RemoteData(role Sen, role Sat, role Ser){
  rec Loop {
    choice at Ser {
      GetData() from Ser to Sat within [5;6] using a and resetting ();
      GetData() from Sat to Sen within [5;6] using b and resetting ();
      Data() from Sen to Sat within [6;7] using b and resetting (b);
      Data() from Sat to Ser within [6;7] using a and resetting (a);
      continue Loop
    } or {
      Close() from Ser to Sat within [5;6] using a and resetting ();
      Close() from Sat to Sen within [5;6] using b and resetting ();    } } }
```

Figure 10 Remote data protocol in νScr^T.

relevant clock provided in the type for the time window and resetting the clock if necessary.

Error Handling The error handling capabilities of `MultiCrusty`^{T} cover various potential errors that may arise during protocol implementation and execution. These errors include the misuse of generated types and timeouts, showcasing the flexibility of our implementation in verifying communication protocols. For instance, if Lines 11 and 12 in Fig. 9 are swapped, the program will fail to compile because it expects a `send` primitive in Line 11, as indicated by the type of `'s'`. Another compile-time error occurs when a payload with the wrong type is sent. For example, attempting to send a `Data` message instead of a `GetData` in Line 11 will result in a compilation error. `MultiCrusty`^{T} can also identify errors at runtime. If the content of the function `endpoint_data_ser`, spanning in Lines 10–12, is replaced with a single `Ok(())`, the code will compile successfully. However, during runtime, the other roles will encounter failures as they consider `Ser` to have failed.

Timeouts are handled dynamically within `MultiCrusty`^{T}. If a time-consuming task with a 10-second delay is introduced between Lines 11 and 12, `Ser` will enter a sleep state for the same duration. Consequently, the `recv` operation in Line 12 will encounter a time constraint violation, resulting in the failure and termination of `Ser`. Furthermore, the absence of clock `'a'` in the set of clocks, where it is required for a specific primitive, will trigger a runtime error, as the evaluation of time constraints depends on the availability of the necessary clocks.

Timed Protocol Specification To specify timed multiparty protocols, we extend νScr [43], a multiparty protocol description language, with time features, resulting in νScr^T. Additional keywords such as `within`, `using`, and `and resetting` are incorporated in νScr to support the specification of time windows, clocks, and resets, respectively. In Fig. 10, we illustrate the νScr^T protocol for remote data, showcasing the application of these enhancements. νScr^T ensures the accuracy of timed multiparty protocols by verifying interactions, validating time constraints, handling clock increments, and performing standard MPST protocol checks.

6 Evaluation: Expressiveness, Case Studies and Benchmarks

We evaluate our toolchain `MultiCrusty`^{T} from two perspectives: *expressivity* and *feasibility*. In terms of expressivity, we implement protocols from the session type literature [20, 33, 13, 24, 21, 36], as well as newly introduced protocols derived from real-world applications [7, 38, 2, 35, 41]. Regarding feasibility, we compare our system to `MultiCrusty` [28], an untimed implementation of affine synchronous MPST, demonstrating that our tool introduces negligible compile-time and runtime overhead in all cases, as expected.

6.1 Performance: `MultiCrusty`^{T} vs. `MultiCrusty`

When comparing `MultiCrusty`^{T} with `MultiCrusty`, we evaluate their performance on two standard benchmark protocols: the *ring* protocol, which involves sequentially passing a

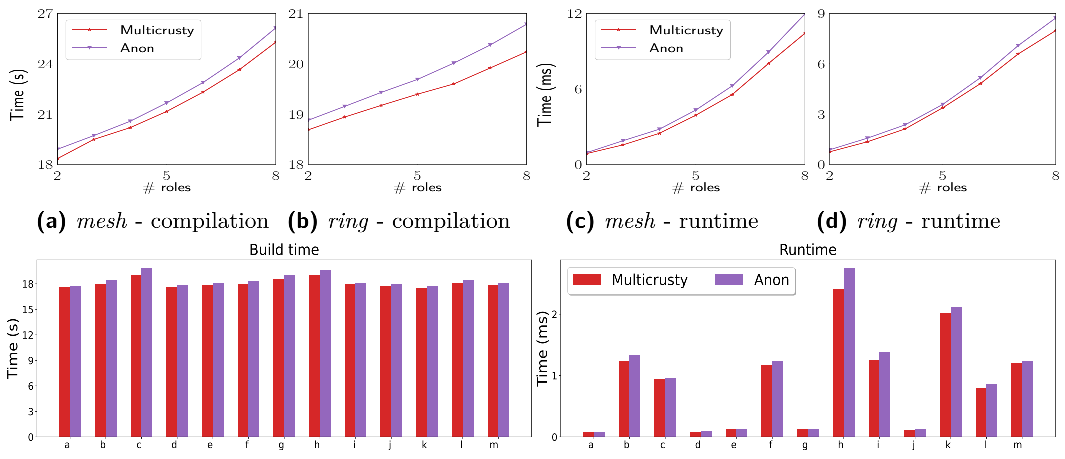


Figure 11 Top: microbenchmark results for mesh and ring protocols. Bottom: benchmark results for Calculator [20] (a), Online wallet [33] (b), SMTP [36] (c), Simple voting [20] (d), Three buyers [24] (e), Travel agency [21] (f), OAuth [33] (g), HTTP [13] (h), Remote data [7] (i), Servo [38] (j), Gravity sensor [2] (k), PineTime Heart Rate [35] (l), and Proximity Based Car Key [41] (m).

message through roles, and the *mesh* protocol, where each participant sends a message to every other. Both protocols underwent 100 iterations within a time window of 0 to 10 seconds. Fig. 11 (top) displays benchmark results for roles ranging from 2 to 8.

In the *ring* protocol, compile-time benchmarks (Fig. 11b) indicate that `MultiCrusty`^{T} experiences a marginal slowdown of less than 2% with 2 roles, but achieves approximately 5% faster compilation time with 8 roles. Regarding runtime benchmarks (Fig. 11d), `MultiCrusty` demonstrates a 15% speed advantage with 2 roles, which decreases to 5% with 8 roles. The overhead remains consistent, with a difference of less than 0.5 ms at 6, 7, and 8 roles.

In the *mesh* protocol, where all roles send and receive messages (compile-time benchmarks in Fig. 11a and runtime benchmarks in Fig. 11c), `MultiCrusty`^{T} compiles slightly slower (less than 1% at 2 roles, 4% at 8 roles) and runs slower as well (less than 1% at 2 roles, 15% at 8 roles). Compile times for `MultiCrusty`^{T} range from 18.9 s to 26 s, with running times ranging between 0.9 ms and 11.9 ms. The performance gap widens exponentially with the increasing number of enforced time constraints. In summary, as the number of roles increases, `MultiCrusty`^{T} demonstrates a growing overhead, mainly attributed to the incorporation of additional time constraint checks.

6.2 Expressivity and Feasibility with Case Studies

We implement a variety of protocols to showcase the expressivity, feasibility, and capabilities of `MultiCrusty`^{T}, conducting benchmarking using both `MultiCrusty`^{T} and `MultiCrusty`. The `send` and `recv` operations in both libraries are ordered, directed, and involve the same set of participants. Additionally, when implemented with `MultiCrusty`^{T}, these operations are enriched with time constraints and reset predicates. The benchmark results for the selected case studies, including those from prior research and five additional protocols sourced from industrial use cases [7, 38, 2, 35, 41], are presented in the bottom part of Fig. 11. To ensure a fair comparison between `MultiCrusty`^{T} (🟪) and `MultiCrusty` (🟥), time constraints are enforced for all examples without introducing any additional sleep or timeouts.

Note that rate-based protocols ((k), (l), (m) in Fig. 11 (bottom)) from real-time systems [2, 35, 41] are implemented in `MultiCrusty`^{T}, showcasing its expressivity in real-time applications. These implementations feature the establishment of consistent time constraints and a shared clock for operations with identical rates. For example, in the Car Key protocol [41], where the car periodically sends a wake-up message to probe the presence of the key, all interactions between two wake-up signals must occur within a period of e.g. 100 ms. Consequently, when implementing this protocol with `MultiCrusty`^{T}, all time constraints are governed by a single clock ranging from 0 to 100 ms, with the clock resetting at the end of each loop.

The feasibility of our tool, `MultiCrusty`^{T}, is demonstrated in Fig. 11 (bottom). The results indicate that `MultiCrusty`^{T} incurs minimal compile-time overhead, averaging approximately 1.75%. Moreover, the runtime for each protocol remains within milliseconds, ensuring negligible impact. Notably, in the HTTP protocol, the runtime comparison percentage with `MultiCrusty` is 87.60%, primarily due to the integration of 126 time constraints within it. The relevant implementation metrics, including multiple participants (MP), branching, recursion (Rec), and time constraints, are illustrated in Table 2.

Table 2 Metrics for protocols implemented in `MultiCrusty`^{T}.

Protocol	Generated Types	Implemented Lines of Code	MP	Branching	Rec	Time Constraints
Calculator [20]	52	51	✗	✓	✓	11
Online wallet [33]	142	160	✓	✓	✓	24
SMTP [36]	331	475	✗	✓	✓	98
Simple voting [20]	73	96	✗	✓	✗	16
Three buyers [24]	108	78	✓	✓	✗	22
Travel agency [21]	148	128	✓	✓	✓	30
oAuth [33]	199	89	✓	✓	✗	30
HTTP [13]	648	610	✓	✓	✓	126
Remote data [7]	100	119	✓	✓	✓	16
Servo [38]	74	48	✓	✗	✗	10
Gravity sensor [2]	61	95	✗	✓	✓	9
PineTime Heart Rate [35]	101	111	✗	✓	✓	17
Proximity Based Car Key [41]	70	134	✗	✓	✓	22

7 Related Work and Conclusion

Time in Session Types Bocchi et al. [4] propose a timed extension of MPST to model real-time choreographic interactions, while Bocchi et al. [3] extend *binary* session types with time constraints, introducing a subtyping relation and a blocking receive primitive with timeout in their calculus. In contrast to their strategies to avoid time-related failures, as discussed in § 1 and 2, ATMP focuses on actively managing failures as they occur, offering a distinct approach to handling timed communication.

Iraci et al. [22] extend *synchronous binary* session types with a periodic recursion primitive to model rate-based processes. To align their design with real-time systems, they encode time into a periodic construct, synchronised with a global clock. With *rate compatibility*, a relation that facilitates communication between processes with different periods by synthesising and verifying a common superperiod type, their approach ensures that well-typed processes remain free from rate errors during any specific period. On the contrary, ATMP integrates time constraints directly into communication through local clocks, resulting in distinct time behaviour. Intriguingly, our method of time encoding can adapt to theirs, while the opposite is not feasible. Consequently, not all the timed protocols in our paper, e.g. Fig. 1b, can be accurately represented in their system. Moreover, due to its *binary* and *synchronous* features, their theory does not directly model and ensure the properties of real distributed systems.

Le Brun et al. [30] develop a theory of multiparty session types that accounts for different failure scenarios, including message losses, delays, reordering, as well as link failures and network partitioning. Unlike ATMP, their approach does not integrate time specifications or address failures specifically related to time. Instead, they use *timeout* as a generic message label (🕒) for failure branches, which triggers the failure detection mechanism. Except for [22], all the mentioned works on session types with time are purely theoretical.

Affinity, Exceptions and Error-Handling in Session Types Mostrous and Vasconcelos [31] propose affine binary session types with explicit cancellation, which Fowler et al. [14] extend to define Exceptional GV for binary asynchronous communication. Exceptions can be nested and handled over multiple communication actions, and their implementation is an extension of the research language Links. Harvey et al. [15] incorporate MPST with explicit connection actions to facilitate multiparty distributed communication, and develop a code generator based on the actor-like research language Ensemble to implement their approach. The work in [31] remains theoretical, and both [31, 14] are limited to binary and linear logic-based session types. Additionally, none of these works considers time specifications or addresses the handling of time-related exceptions in their systems, which are key aspects of our work.

Session Types in Rust `MultiCrusty`, extensively compared to `MultiCrusty`^{T}, is a Rust implementation based on affine MPST by Lagaillardie et al. [28]. Their approach relies on *synchronous* communication, rendering time and timeout exceptions unnecessary.

Cutner et al. [10] introduce `Rumpsteak`, a Rust implementation based on the `tokio` Rust library, which uses a different design for asynchronous multiparty communications compared to `MultiCrusty`^{T}, relying on the `crossbeam_channel` Rust library. The main goal of [10] is to compare the performance of `Rumpsteak`, mainly designed to analyse asynchronous message reordering, to existing tools such as the k-MC tool developed in [29]. Unlike `MultiCrusty`^{T}, `Rumpsteak` lacks formalisation, or handling of timed communications and failures.

`Typestate` is a Rust library implemented by Duarte and Ravara [12], focused on helping developers to write safer APIs using typestates and their macros `#[typestate]`, `#[automaton]` and `#[state]`. `MultiCrusty`^{T} and `Typestate` are fundamentally different, with `Typestate` creating a state machine for checking possible errors in APIs and not handling affine or timed communications. `Ferrite`, a Rust implementation introduced by Chen et al. [6], is limited to binary session types and forces the use of linear channels. The modelling of `Ferrite` is based on the shared binary session type calculus $\texttt{SILL}_s$.

Jespersen et al. [23] and Kokke [25] propose Rust implementations of binary session types for synchronous communication protocols. [22] extends the framework from [23] to encode the *rate compatibility* relation as a Rust trait and check whether two types are rate compatible. Their approach is demonstrated with examples from rate-based systems, including [2, 35, 41]. Motivated by these applications, we formalise and implement the respective timed protocols in `MultiCrusty`^{T}, showcasing the expressivity and feasibility of our system in real-time scenarios.

Conclusion and Future Work To address time constraints and timeout exceptions in asynchronous communication, we propose *affine timed multiparty session types* (ATMP) along with the toolchain `MultiCrusty`^{T}, an implementation of ATMP in Rust. Thanks to the incorporation of affinity and failure handling mechanisms, our approach renders impractical conditions such as *wait-freedom* and *urgent receive* obsolete while ensuring communication safety, protocol conformance, and deadlock-freedom, even in the presence of (timeout) failures. Compared to a synchronous toolchain without time, `MultiCrusty`^{T} exhibits negligible overhead in various complex examples including those from real-time systems, while enabling the verification of time constraints under asynchronous communication. As future work, we plan to explore automatic recovery from errors and timeouts instead of simply terminating processes, which will involve extending the analysis of communication causality to timed global types and incorporating reversibility mechanisms into our system.

References

1 Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.

2 Android. Motion Sensors, 2009. https://developer.android.com/guide/topics/sensors/sensors_motion. URL: https://developer.android.com/guide/topics/sensors/sensors_motion.

3 Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Asynchronous timed session types. In Luís Caires, editor, *Programming Languages and Systems*, pages 583–610, Cham, 2019. Springer International Publishing.

4 Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, volume 8704 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2014. doi:10.1007/978-3-662-44584-6_29.

5 David Castro, Raymond Hu, SungShik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role-Parametric Session Types in Go: Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. Place: New York, NY, USA Publisher: Association for Computing Machinery. doi:10.1145/3290342.

6 Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite: A Judgmental Embedding of Session Types in Rust. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16250, doi:10.4230/LIPIcs.ECOOP.2022.22.

7 Yingying Chen, Minghu Zhang, Xin Li, Tao Che, Rui Jin, Jianwen Guo, Wei Yang, Baosheng An, and Xiaowei Nie. Satellite-enabled internet of remote things network transmits field data from the most remote areas of the tibetan plateau. *Sensors*, 22(10):3713, 2022. URL: https://doi.org/10.3390/s22103713, doi:10.3390/S22103713.

8 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016. doi:10.1017/S0960129514000188.

9 The Developers of Crossbeam. Crate: Crossbeam channel, 2022. Last accessed: October 2022. URL: https://crates.io/crates/crossbeam-channel.

10 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693 of *PPoPP '22*, pages 261–246. ACM, 2022. doi:10.1145/3503221.3508404.

11 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 174–186, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39212-2_18.

12 José Duarte and António Ravara. Taming stateful computations in rust with typestates. *Journal of Computer Languages*, 72:101154, 2022. URL: https://www.sciencedirect.com/science/article/pii/S259011842200051X, doi:10.1016/j.cola.2022.101154.

13 Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Technical Report RFC7230, RFC Editor, June 2014. URL: https://www.rfc-editor.org/info/rfc7230, doi:10.17487/rfc7230.

14 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional Asynchronous Session Types: Session Types Without Tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, January 2019. Place: New York, NY, USA Publisher: ACM. doi:10.1145/3290341.

15 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, page 30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://2021.ecoop.org/details/ecoop-2021-ecoop-research-papers/12/Multiparty-Session-Types-for-Safe-Runtime-Adaptation-in-an-Actor-Language`, `doi:10.4230/LIPIcs.ECOOP.2021.10`.

16 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. URL: `https://doi.org/10.1007/BFb0053567`, `doi:10.1007/BFB0053567`.

17 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. Full version in [18]. `doi:10.1145/1328438.1328472`.

18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1), 2016. `doi:10.1145/2827695`.

19 Ping Hou, Nicolas Lagaillardie, and Nobuko Yoshida. Fearless asynchronous communications with timed multiparty session protocols, 2024. URL: `https://arxiv.org/abs/2406.19541`, `arXiv:2406.19541`.

20 Raymond Hu and Nobuko Yoshida. Hybrid Session Verification Through Endpoint API Generation. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering*, volume 9633, pages 401–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. URL: `http://link.springer.com/10.1007/978-3-662-49665-724`, `doi:10.1007/978-3-662-49665-724`.

21 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. `doi:10.1007/978-3-540-70592-5_22`.

22 Grant Iraci, Cheng-En Chuang, Raymond Hu, and Lukasz Ziarek. Validating iot devices with rate-based session types. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1589–1617, 2023. `doi:10.1145/3622854`.

23 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 13–22, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2808098.2808100`.

24 Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and Blame Assignment for Higher-Order Session Types. *SIGPLAN Not.*, 51(1):582–594, January 2016. `doi:10.1145/2914770.2837662`.

25 Wen Kokke. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science*, 304:48–60, Sep 2019. URL: `http://dx.doi.org/10.4204/EPTCS.304.4`, `doi:10.4204/eptcs.304.4`.

26 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking Protocols with Mungo and stmungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 146–159, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2967973.2968595`.

27 Pavel Krcál and Wang Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In Thomas Ball and Robert B. Jones, editors, *Computer Aided*

Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, volume 4144 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2006. doi:10.1007/11817963_24.

28 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16232, doi:10.4230/LIPIcs.ECOOP.2022.4.

29 Julien Lange and Nobuko Yoshida. Verifying Asynchronous Interactions via Communicating Session Automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117, Cham, 2019. Springer. doi:10.1007/978-3-030-25540-4_6.

30 Matthew Alan Le Brun and Ornela Dardha. MAGπ: Types for Failure-Prone Communication. In Thomas Wies, editor, *Programming Languages and Systems*, pages 363–391, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-30044-8_14.

31 Dimitris Mostrous and Vasco T. Vasconcelos. Affine Sessions. *Logical Methods in Computer Science ; Volume 14*, 8459:Issue 4 ; 18605974, 2018. Medium: PDF Publisher: Episciences.org. URL: https://lmcs.episciences.org/4973, doi:10.23638/LMCS-14(4:14)2018.

32 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing*, 29(5):877–910, 2017.

33 Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local Verification of Global Protocols. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, volume 8174 of *LNCS*, pages 358–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40787-1_25.

34 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

35 Pine64. PineTime, 2019. https://www.pine64.org/pinetime/. URL: https://www.pine64.org/pinetime/.

36 Jonathan Postel. Rfc0821: Simple mail transfer protocol, 1982.

37 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

38 Servo. Servo Web Browser commit, 2015. https://github.com/servo/servo/commit/434a5f1d8b7fa3e2abd36d832f16381337885e3d.

39 Developers Rust of the library Time. Module std::time documentation, 2023. https://doc.rust-lang.org/std/time/index.html. URL: https://doc.rust-lang.org/std/time/index.html.

40 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, October 2021. URL: https://dl.acm.org/doi/10.1145/3485501, doi:10.1145/3485501.

41 Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. Fast, furious and insecure: Passive keyless entry and start systems in modern supercars. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):66–85, 2019. URL: https://doi.org/10.13154/tches.v2019.i3.66-85, doi:10.13154/TCHES.V2019.I3.66-85.

42 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, pages 22–41, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-05119-2_3.

43 Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In Evripidis Bampis and Aris Pagourtzis, editors, *Fundamentals of Computation Theory*, pages 18–35, Cham, 2021. Springer International Publishing.