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 MultiCrustyT, a RUST toolchain designed to facilitate the implementation of safe affine timed protocols. MultiCrustyT leverages generic types and the time library to handle timed communications, integrated with optional types for affinity. We evaluate MultiCrustyT 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

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 MultiCrustyT, our Rust toolchain designed for building asynchronous timed multiparty protocols under ATMP. MultiCrustyT 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 \textit{MultiCrusty\textsuperscript{T}}, a RUST implementation of ATMP designed for secure timed communications. \textit{MultiCrusty\textsuperscript{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, \textit{MultiCrusty\textsuperscript{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 \textit{MultiCrusty\textsuperscript{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 \textit{MultiCrusty\textsuperscript{T}} exhibits minimal overhead while providing significantly strengthened property checks.

\textbf{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 \textit{MultiCrusty\textsuperscript{T}}, our RUST implementation of ATMP. §6 showcases the compilation and execution benchmarks of \textit{MultiCrusty\textsuperscript{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 \textit{MultiCrusty\textsuperscript{T}} can be found in the full version of the paper [19]. Our toolchain and evaluation examples are available in an artifact.

\section{Overview}

In this section, we give an overview of affine timed multiparty session types (ATMP) and \textit{MultiCrusty\textsuperscript{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, \textit{remote data}. This real-world scenario is sourced from
Fearless Asynchronous Communications with Timed Multiparty Session Protocols

(a) Top-down view of ATMP (left) and MultiCrustyT (right).

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. $5 \leq C_{\text{Ser}} \leq 6$, where $C_{\text{Ser}}$ 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. $C_{\text{Ser}} := 0$, 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 $G$:

$$\text{Sat} \rightarrow \text{Ser}: \{\text{Data}(6 \leq C_{\text{Sat}} \leq 7, C_{\text{Sat}} := 0, 6 \leq C_{\text{Ser}} \leq 7, C_{\text{Ser}} := 0).\text{end}\}$$

where $C_{\text{Sat}}$ and $C_{\text{Ser}}$ 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:

\[
\begin{align*}
T_{\text{Sat}} &= \text{Sat}!\text{Data} \{ 6 \leq C_{\text{Sat}} \leq 7, C_{\text{Sat}} := 0 \}. \text{end} \quad T_{\text{Ser}} &= \text{Sat}!\text{Data} \{ 6 \leq C_{\text{Ser}} \leq 7, C_{\text{Ser}} := 0 \}. \text{end}
\end{align*}
\]

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

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 (AMST) \[28\] by incorporating processes that model time, timeouts, and asynchrony. In our example, \( T_{\text{Sat}} \) and \( T_{\text{Ser}} \) are used for the type-checking of \( s[\text{Sat}] \) and \( s[\text{Ser}] \), which respectively represent the channels (a.k.a. session endpoints) played by roles Sat and Ser in a multiparty session \( s \), within the processes:

\[
P_{\text{Sat}} = \text{delay}(C_1 = 6.5) \cdot s[\text{Sat}]^{0.4}[\text{Ser}]!\text{Data}.0 \quad P_{\text{Ser}} = \text{delay}(C_2 = 6) \cdot s[\text{Ser}]^{0.3}[\text{Sat}]\text{Data.0}
\]

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

**Solution to Stuck Processes Due to Time Failures** It appears that the parallel execution of \( P_{\text{Sat}} \) and \( P_{\text{Ser}} \) is impossible 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 \( s[\text{Ser}] \) waits for 6 time units and cannot receive Data within 0.3 time units, a timeout failure is raised (\( \text{timeout}[s[\text{Ser}]^{0.3}[\text{Sat}]\text{Data.0}] \)). Furthermore, we apply our time-failure handling approach to manage this timeout failure, initiating the termination of the channel \( s[\text{Ser}] \) and triggering the cancellation process of the session \( s \) (\( s! \)). 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\textsuperscript{T}: Toolchain Overview

To augment the theory, we introduce the MultiCrusty\textsuperscript{T} library, a toolchain for implementing communication protocols in Rust. MultiCrusty\textsuperscript{T} specifies protocols where communication operations must adhere to specific time limits (\textit{timed}), allowing for asynchronous message reception and runtime handling of certain failures (\textit{affine}). This library relies on two fundamental types: \texttt{Send} and \texttt{Recv}, representing message sending and receiving, respectively.
Fearless Asynchronous Communications with Timed Multiparty Session Protocols

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>`, and `NameSat`, `RoleSer<RoleSer<End>>`. (a) `Send` type (b) `Recv` type (c) `MeshedChannels` type for `Sat` Figure 2 Main types of `MultiCrusty^T` 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. 2e 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 `vScr` [43], a language for describing multiparty communication protocols, to include time constraints, resulting in `vScr^T`. A timed global protocol represented in `vScr^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 `C` denote a finite set of clocks, ranging over `C, C', C_1, ...,` that take non-negative real values in `R_{\geq 0}`. Additionally, let `t, t', t_1, ...` be time constants ranging over `R_{\geq 0}`. A clock constraint `δ` over `C` is defined as:
\[ \delta \equiv \text{true} \mid C > b \mid C = b \mid \neg \delta \mid \delta_1 \land \delta_2 \]

where \( C \in \mathbb{C} \) and \( b \) is a constant time bound ranging over non-negative rationals \( \mathbb{Q}_{\geq 0} \).

We define \text{false}, \text{<}, \text{\geq}, \text{\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 \( V : \mathbb{C} \rightarrow \mathbb{R}_{\geq 0} \) assigns time to each clock in \( \mathbb{C} \). We define \( V + t \) as the valuation that assigns to each \( C \in \mathbb{C} \) the value \( V(C) + t \). The initial valuation that maps all clocks to 0 is denoted as \( V^0 \), and the valuation that assigns a value of \( t \) to all clocks is denoted as \( V^t \). \( V \models \delta \) indicates that the constraint \( \delta \) is satisfied by the valuation \( V \). Additionally, we use \( \bigcup_{i \in I} V_i \) to represent the overlapping union of the valuations \( V_i \) for \( i \in I \).

A reset predicate \( \lambda \) over \( \mathbb{C} \) is a subset of \( \mathbb{C} \) 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 \( V \) but with the values of clocks in \( \lambda \) to 0, we use \( V[\lambda \mapsto 0] \).

Syntax of Processes Our session \( \pi \)-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.\(^1\)

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

\[
\begin{align*}
  &c,d := x \mid s[p] &\quad \text{(variable, channel with role } p) \\
  &P,Q := 0 \mid P \mid Q \mid (\nu s) P &\quad \text{(inaction, parallel composition, restriction)} \\
  &c^p[q] \oplus m(d).P &\quad \text{(timed selection towards role } q) \\
  &c^p[q] \sum_{i \in I} m_i(x_i).P_i &\quad \text{(timed branching from role } q \text{ with } I \neq \emptyset) \\
  &\text{def } D \text{ in } P &\quad \text{(process definition, process call)} \\
  &\text{delay}(\delta).P &\quad \text{(time-consuming delay, deterministic delay)} \\
  &\text{timeout}(P) &\quad \text{(timeout failure, try-catch)} \\
  &\text{cancel}(c).P &\quad \text{(cancel, communication error, kill)} \\
  &s[p] \triangleright \sigma &\quad \text{(output message queue of role } p \text{ in session } s) \\
  &D := X(\bar{c}) = P &\quad \text{(declaration of process variable } X) \\
  &\sigma := q[m(s[x])].\sigma &\quad \text{(message queue, non-empty or empty)}
\end{align*}
\]
\end{itemize}

Restriction, branching, and process definitions and declarations act as binders; \( \text{fc}(P) \) is the set of free channels with roles in \( P \), \( \text{fv}(P) \) 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:

\begin{itemize}
  \item \textbf{Channels} \( c,d \), being either variables \( x \) or channels with roles (a.k.a. session endpoints) \( s[p] \).
  \item \textbf{Standard} processes as in [37, 28], including inaction \( 0 \), parallel composition \( P \mid Q \), session scope restriction \((\nu s) P \), process definition \text{def } D \text{ in } P \), process call \( X(\bar{c}) \), and communication error \( \text{cerr} \).
  \item \textbf{Time} processes that follow the program time behaviour of Fig. 2c:
\end{itemize}

\(^1\) 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) \( c^n[q]@m(d).P \) indicates that a message \( m \) with payload \( d \) is sent to role \( q \) via endpoint \( c \), whereas timed branching (or timed external choice) \( c^n[q]\sum_{i \in I}m(x_i).P_i \) waits to receive a message \( m \) from role \( q \) via endpoint \( c \) and then proceeds as \( P_i \).

The parameter \( n \) in both timed selection and branching is a timeout that allows modelling different types of communication primitives: blocking with a timeout (\( n \in \mathbb{R}_{\geq 0} \)), blocking (\( n = \infty \)), or non-blocking (\( n = 0 \)). When \( n \in \mathbb{R}_{\geq 0} \), the timed selection (or timed branching) process waits for up to \( n \) 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 \( n \) is set to \( \infty \), 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.

- delay(\( \delta \)). \( P \) represents a time-consuming delay action, such as method invocation or sleep. Here, \( \delta \) is a clock constraint involving a single clock variable \( C \), used to specify the interval for the delay. When executing delay(\( \delta \)). \( P \), any time value \( t \) that satisfies the constraint \( \delta \) can be consumed. Consequently, the runtime deterministic delay process delay(\( t \)). \( P \), arising during the execution of delay(\( \delta \)). \( P \), is introduced. In delay(\( t \)). \( P \), \( t \) is a constant and a solution to \( \delta \), and \( P \) is executed after a precise delay of \( t \) time units.

- 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 0 catch \( Q \) is not allowed within our calculus.

- cancel(\( c \)). \( P \) performs the cancellation of other processes with channel \( c \).

- \( s_j \) kills (terminates) all processes with session \( s \), and is dynamically generated only at runtime from timeout failure or cancel processes.

Message queues: \( s[p] \| \sigma \) represents the output message queue of role \( p \) in session \( s \). It contains all the messages previously sent by \( p \). The queue \( \sigma \) can be a sequence of messages of the form \( q[m(s[x])] \), where \( q \) is the receiver, or \( \epsilon \), indicating an empty message queue. The set of receivers in \( \sigma \), denoted as receivers(\( \sigma \)), is defined in a standard way as:

\[
\text{receivers}(q[m(s[x]) \cdot \sigma]) = \{q\} \cup \text{receivers}(\sigma) \quad \text{receivers}(\epsilon) = \emptyset
\]

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

Definition 2 (Semantics). A try-catch context \( E \) is defined as \( E ::= \text{try } E \text{ catch } P \mid [] \), and a reduction context \( C \) is defined as \( C ::= C \mid P \mid \text{def } D \text{ in } C \mid [] \). The reductions +/- and \(-\rightarrow\) are inductively defined in Fig. 3 (top), with respect to a structural congruence \( \equiv \) depicted in Fig. 3 (bottom). We write \( \rightarrow^* \) and \( \rightarrow^\star \) for their reflexive and transitive closures, respectively. \( P \rightarrow^* P' \) (or \( P \rightarrow^\star P' \)) means \( \exists P'' \) such that \( P \rightarrow P'' \) (or \( P \rightarrow P'' \)) is derivable. We say \( P \) has a communication error if \( P = C[\text{cerr}] \).

We decompose the reduction rules in Fig. 3 into three relations: \( \rightarrow\) represents instantaneous reductions without time consumption, \( \rightarrow\) handles time-consuming steps, and \( \rightarrow\) is a
general relation that can arise either from \( \rightarrow \) by [R-In] or \( \rightarrow \) by [R-Term]. Now let us explain the operational semantics rules for our π-calculus.

**Communication**: Rules [R-Out] and [R-In] model asynchronous communication by queuing and dequeuing pending messages, respectively. Rule [R-Enr] is triggered by a message label mismatch, resulting in a fatal communication error.

**Time**: Rule [R-Der] specifies a deterministic delay of a specific duration \( t \), when \( t \) is a solution to the clock constraint \( \delta \). Rule [R-Term] 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 duration \( t \) that may occur at different parts of the process. It is *undefined* only if \( P \) is a time-consuming delay, i.e., \( P = \text{delay}(\delta) \cdot P' \), or if the specified delay time \( t \) exceeds the duration of a runtime deterministic delay, i.e., \( P = \text{delay}(t') \cdot P' \) 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 \( \text{timeout}[P] \), 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 \( q \) via the endpoint \( \pivp() \). 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.
Fearless Asynchronous Communications with Timed Multiparty Session Protocols

Example 3. Consider the processes: \(P_1 = s[Sat]|^0.4 [\text{Ser}]|^{0.3}[\text{Data}].0, P_2 = s[\text{Ser}]|^{0.3}[\text{Sat}]|^{0.4}[\text{Data}].0\), and \(P_3 = s[\text{Sat}]|^{0.4}[\text{Data}].0\). Rule [C-Cart] can be applied to try \(P_1\) catch \(Q\) \(s_4\), as subj\(P(P_1) = \{s[\text{Sat}]\}\) satisfies its side condition. However, neither timeout\(P_1 | P_2\) nor timeout\(P_3\) can generate the killing process \(s_4\), as subj\(P(P_1 | P_2) = \{s[\text{Sat}], s[\text{Ser}]\}\), whereas subj\(P(P_3) = \{s[\text{Sat}]|^{0.4}\}\).
Example 4. Processes $Q_{\text{Sen}}$, $Q_{\text{Sat}}$, and $Q_{\text{Ser}}$ interact on a session $s$:

$Q_{\text{Sen}} = \text{delay}(C_{\text{Sen}} = 5.6) \cdot Q_{\text{Sen}} | s[\text{Sen}]$ where $Q_{\text{Sen}} = \text{try} s[\text{Sen}]^3 \cdot \text{catch cancel}(s[\text{Sen}])$

$Q_{\text{Sat}} = \text{delay}(C_{\text{Sat}} = 6) \cdot Q_{\text{Sat}} | s[\text{Sat}]$ where $Q_{\text{Sat}} = s[\text{Sat}]^4 \cdot Q_{\text{Sat}}^2 | \text{Data}$

$Q_{\text{Ser}} = \text{delay}(C_{\text{Ser}} = 6) \cdot Q_{\text{Ser}} | s[\text{Ser}]$ where $Q_{\text{Ser}} = s[\text{Ser}]^8 \cdot Q_{\text{Ser}}^4 | \text{Data}$

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

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

$Q_{\text{Sen}} \cdot Q_{\text{Sat}} \cdot Q_{\text{Ser}} \rightarrow \text{delay}(6.5) \cdot Q_{\text{Sen}} \cdot s[\text{Sen}] \cdot \epsilon \cdot \text{delay}(6) \cdot Q_{\text{Sat}} \cdot s[\text{Sat}] \cdot \epsilon \cdot \text{delay}(6) \cdot Q_{\text{Ser}} \cdot s[\text{Ser}] \cdot \epsilon$

$\equiv Q_{\text{Sen}} \cdot s[\text{Sen}] \cdot \epsilon \cdot \text{timeout}(Q_{\text{Sat}}) \cdot s[\text{Sat}] \cdot \epsilon \cdot \Psi_{0.5}(Q_{\text{Ser}}) \cdot s[\text{Ser}] \cdot \epsilon$

Therefore, the kill process $s_\delta^\epsilon$ is generated from timeout[$Q'_{\text{Sat}}$], successfully terminating the process $Q_{\text{Sen}} \cdot Q_{\text{Sat}} \cdot Q_{\text{Ser}}$ by the following reductions:

$Q_{\text{Sen}} \cdot s[\text{Sen}] \cdot \epsilon \cdot \text{timeout}(Q'_{\text{Sat}}) \cdot s[\text{Sat}] \cdot \epsilon \cdot \Psi_{0.5}(Q_{\text{Ser}}) \cdot s[\text{Ser}] \cdot \epsilon$

$\Rightarrow Q_{\text{Sen}} \cdot s[\text{Sen}] \cdot \epsilon \cdot s_\delta \cdot s[\text{Sat}] \cdot \epsilon \cdot \Psi_{0.5}(Q_{\text{Ser}}) \cdot s[\text{Ser}] \cdot \epsilon$

$\Rightarrow \text{cancel}(s[\text{Sen}]) \cdot s[\text{Sen}] \cdot \epsilon \cdot s_\delta \cdot s[\text{Sat}] \cdot \epsilon \cdot s[\text{Ser}] \cdot \epsilon$

$\Rightarrow s_\delta \cdot s[\text{Sen}] \cdot \epsilon \cdot s_\delta \cdot s[\text{Sat}] \cdot \epsilon \cdot 0 \cdot s[\text{Ser}] \cdot \epsilon \equiv 0 \cdot s_\delta$

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 $S, S', S_1, \ldots$, and facilitate the delegation of the remaining behaviour $T$ to the receiver, who can execute it under any clock assignment satisfying $\delta$.

Timed Global Types are ranged over $G, G', G_1, \ldots$, and describe an overview of the behaviour for all roles $(p, q, s, t, \ldots)$ belonging to a (fixed) set $R$. The set of roles in a timed global type $G$ is denoted as roles$(G)$, while the set of its free variables as fv($G$).
entails fewer external choices but more internal choices. This relation indicates that a smaller type
of message types mirror those in timed global types. In requirements for the index set, labels, clock constraints, and reset predicates in timed local
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 \( p \rightarrow q \{ m_i (S_i) \{ \delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i} \}. G_i \}_{i \in I} \) (\( j \in I \)) is a runtime
construct to represent a message \( m_j \) sent by \( p \), and yet to be received by \( q \). Recursion \( \mu T \) and termination \( \text{end} \) (omitted where unambiguous) are standard [11]. Note that contractive
requirements [34, §21.8], i.e. ensuring that each recursion variable \( t \) is bound within a \( \mu T \ldots \) 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) \( p \oplus \{ m_i (S_i) \{ \delta_i, \lambda_i \}. T_i \}_{i \in I} \) (or external choice (branching) \( p \& \{ m_i (S_i) \{ \delta_i, \lambda_i \}. T_i \}_{i \in I} \) states that the current role is to
send to (or receive from) the role \( p \) when \( \delta_i \) is satisﬁed, followed by resetting the clocks in \( \lambda_i \). 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 \( \mathcal{M}, \mathcal{M}', \mathcal{M}_1, \ldots \), and represent (possibly empty) sequences of message types \( p \mathcal{m}(S) \) having receiver \( p \), label \( m \), and payload type \( S \) (omitted when \( S = (\delta, \text{end}) \)). As interactions in our formalisation are asynchronous, queue types are used to
characterise the states in which messages are in transit. We adopt the notation \( \text{receivers}(\cdot) \) for
§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 \( \preceq \) on timed local types in Def. 5, based on the
standard behaviour-preserving subtyping [37]. This relation indicates that a smaller type
tails fewer external choices but more internal choices.

**Definition 5 (Subtyping).** The subtyping relation \( \preceq \) is coinductively deﬁned:

\[
\begin{align*}
\forall i \in I \quad S'_i \subseteq S_i & \quad \delta_i = \delta'_i \quad \lambda_i = \lambda'_i \quad T_i \leq T'_i & \quad [\text{Sub-S}] \\
p \oplus \{ m_i (S_i) \{ \delta_i, \lambda_i \}. T_i \}_{i \in I} \preceq p \oplus \{ m_i (S_i) \{ \delta'_i, \lambda'_i \}. T'_i \}_{i \in I} & \quad [\text{Sub-\oplus}] \\
p \& \{ m_i (S_i) \{ \delta_i, \lambda_i \}. T_i \}_{i \in I} \preceq p \& \{ m_i (S_i) \{ \delta'_i, \lambda'_i \}. T'_i \}_{i \in I} & \quad [\text{Sub-\&}] \\
T' \leq T' & \quad [\text{Sub-L}] \\
T = T' & \quad [\text{Sub-R}] \\
\end{align*}
\]
Projection of a timed global type $G$ onto a role $p$ 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 $p$, written as $G | p$, is:

\[
(q \mapsto x : (\{m_i(S_i)\{o_{i_1}, o_{i_2}, \ldots, o_{i_n}.G_i\}_{i \in I}) | p =
\begin{cases}
  \{r \mapsto \{m_i(S_i)\{o_{i_1}, o_{i_2}, \ldots, o_{i_n}.G_i\}_{i \in I}) | p \text{ if } p = q \\
  \bigcap_{i \in I} G_i | p \text{ otherwise}
\end{cases}
\]

\[
(q \mapsto x : (\{m_i(S_i)\{o_{i_1}, o_{i_2}, \ldots, o_{i_n}.G_i\}_{i \in I}) | p =
\begin{cases}
  \{G_j | p \text{ if } p = q \\
  \bigcap_{i \in I} G_i | p \text{ otherwise}
\end{cases}
\]

\[
(mut.G) | p = \begin{cases}
  \text{end if } p \in \text{roles}(G) \text{ or } \forall (mut.G) \neq \emptyset \text{ t p p = t} \\
  \text{end if } p = t
\end{cases}
\]

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

\[
p \& \{m_i(S_i)\{o_{i_1}, o_{i_2}, \ldots, o_{i_n}.T_i\}_{i \in I} \cap \{m_j(S_j)\{o_{j_1}, o_{j_2}, \ldots, o_{j_n}.T_j\}_{j \in J} =
\]

\[
p \& \{m_i(S_i)\{o_{i_1}, o_{i_2}, \ldots, o_{i_n}.T_i\}_{i \in I} \cap \{m_j(S_j)\{o_{j_1}, o_{j_2}, \ldots, o_{j_n}.T_j\}_{j \in J}
\]

4.2 Typing Environments

To reflect the behaviour of timed global types (§4.3), present a typing system for our session $\pi$-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 $\Theta$ and $\Gamma$ are defined as:

\[
\Theta ::= \emptyset \mid \Theta, X:(V_1,T_1),\ldots,(V_n,T_n) \quad \Gamma ::= \emptyset \mid \Gamma, x:(V,T) \mid \Gamma, s[p]:\tau
\]

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

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

We write $\text{dom}(\Gamma) = \{s\}$ if for any $c \in \text{dom}(\Gamma)$, there is $p$ such that $c = s[p]$ (i.e. $\Gamma$ only contains session $s$). We write $s \not\in \Gamma$ if $\forall p : s[p] \not\in \text{dom}(\Gamma)$ (i.e. session $s$ does not occur in $\Gamma$). We write $\Gamma_s \text{ iff } \text{dom}(\Gamma_s) = \{s\}$, $\text{dom}(\Gamma_s) \subseteq \text{dom}(\Gamma)$, and $\forall s[p] \in \text{dom}(\Gamma) : \Gamma(s[p]) = \Gamma_s(s[p])$ (i.e. restriction of $\Gamma$ 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 $c \neq c'$).

Concurrency and subtyping are imposed on typing environments: $\Gamma \equiv \Gamma'$ (resp. $\Gamma \subset \Gamma'$) iff $\text{dom}(\Gamma) = \text{dom}(\Gamma')$ and $\forall c \in \text{dom}(\Gamma) : \Gamma(c) \equiv \Gamma'(c)$ (resp. $\Gamma(c) \subset \Gamma'(c)$), incorporating additional congruence and subtyping rules for time-session/queue types, as depicted in Fig. 6.
Fearless Asynchronous Communications with Timed Multiparty Session Protocols

(\(V, T\) \(\equiv (V, T)\) \(\quad p \neq q\)
\(p_{\text{lm}}(S_1) \cdot q_{\text{lm}}(S_2) \cdot M \equiv q_{\text{lm}}(S_2) \cdot p_{\text{lm}}(S_1) \cdot M\)
\(M \equiv M'\) \(\quad (V, T) \equiv (V, T')\)
\(\diamond \cdot \diamond \equiv \diamond \quad p_{\text{lm}}(S) \cdot \diamond \equiv \diamond \cdot p_{\text{lm}}(S) \cdot \diamond\)
\(M \in M'\) \(\quad (V, T) \leq (V, T')\)
\(q_{\text{lm}}(S) \cdot M \leq q_{\text{lm}}(S') \cdot M'\)
\(\quad (V, T) \leq (V, T')\)
\(\quad M, (V, T) \in (V, T')\)

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

In Def. 8, the typing environment \(\Theta\) maps process variables to \(n\)-tuples of timed session types, while \(\Gamma\) 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 \((V, T)\). 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 \([\text{Sub-M}]\) 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 \(\alpha\) be a transition label of the form \(s.p[q,m]\), \(s.p.q.m\), or \(t\). The typing environment transition \(\Rightarrow\) is inductively defined by the rules in Fig. 7 (top). We write \(\Gamma \Rightarrow \alpha \Gamma'\) iff \(\Gamma \Rightarrow_{\alpha} \Gamma'\) for some \(\Gamma'\). We define two reductions \(\Gamma \Rightarrow_{\alpha} \Gamma'\) (where \(s\) is a session) and \(\Gamma \Rightarrow_{\alpha} \Gamma'\) as follows:

\(\Gamma \Rightarrow_{\alpha} \Gamma'\) holds iff \(\Gamma \Rightarrow_{\alpha} \Gamma'\) with \(\alpha \in \{s.p[q,m], s.p.q.m, t | p, q \in R\}\) (where \(R\) is the set of all roles). We write \(\Gamma \Rightarrow s \Gamma'\) for some \(\Gamma'\), and \(\Rightarrow_{s}\) as the reflexive and transitive closure of \(\Rightarrow_{s}\);

\(\Gamma \Rightarrow_s \Gamma'\) holds iff \(\Gamma \Rightarrow_s \Gamma'\) for some \(s\). We write \(\Gamma \Rightarrow_{t} \Gamma'\) for some \(\Gamma'\), and \(\Rightarrow_{t}\) as the reflexive and transitive closure of \(\Rightarrow_{t}\).

The label \(s.p[q,m]\) indicates that \(p\) sends the message \(m\) to \(q\) on session \(s\), while \(s.p.q.m\) denotes the reception of \(m\) from \(q\) by \(p\) on \(s\). Additionally, the label \(t \in \{R_{\geq 0}\}\) 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 \([\Gamma-p]\) 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 \([\Gamma-c]\) 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 \([\Gamma-x]\) and \([\Gamma-r]\) pertain to untimed reductions, i.e. \(\alpha \neq t\), within a larger environment. Rule \([\Gamma-Tx]\) models time passing on an entry of timed session type by incrementing the associated clock valuation, while rule \([\Gamma-Tq]\) specifies that an entry of queue type is not affected with respect to time progression. Thus, rule \([\Gamma-Tc]\) captures the corresponding time behaviour for a timed-session/queue type entry. Additionally, rule \([\Gamma-T]\) ensures that time elapses uniformly across compatibly composed environments. Other rules are standard: \([\Gamma-\mu]\) is for recursion, and \([\Gamma-\text{struct}]\) ensures that reductions are closed under congruence.

The reduction \(\Gamma \Rightarrow_{\alpha} \Gamma'\) indicates that the typing environment \(\Gamma\) can advance on session \(s\), involving any roles, while \(\Gamma \Rightarrow_{t} \Gamma'\) signifies \(\Gamma\) progressing on any session. This distinction helps in illustrating properties of typed processes discussed in §4.5.
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 $⟨V,G⟩$, where $V$ is a clock valuation. Additionally, we specify the subject of an action $α$ as its responsible principal: $\text{subject}(α)$.

**Definition 10** (Timed Global Type Reduction). The timed global type transition $α →$ is inductively defined by the rules in Fig. 7 (bottom). We denote $⟨V,G⟩ → ⟨V',G'⟩$ if there exists $α$ such that $⟨V,G⟩ →^∗ ⟨V',G'⟩$, $⟨V,G⟩ →$ if there exists $⟨V',G'⟩$ such that $⟨V,G⟩ → ⟨V',G'⟩$, and $→^∗$ as the transitive and reflexive closure of $→$.

In Fig. 7 (bottom), the (highlighted) changes from the standard global type reduction rules [11] focus on time. Rule [GR-μ] accounts for the passage of time by incrementing the clock valuation. Rules [GR-δ] and [GR-κ] 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-ν] 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 $V₀$, i.e. all clocks in $V$ 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), 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 $\Gamma$ is associated with a timed global type $\langle \mathcal{V}; G \rangle$ for a multiparty session s, written $\langle \mathcal{V}; G \rangle \sqsubseteq_s \Gamma$, iff $\Gamma$ can be split into three (possibly empty) sub-environments $\Gamma = \Gamma_G, \Gamma_D, \Gamma_{\text{end}}$ where:

1. $\Gamma_G$ is associated with $\langle \mathcal{V}; G \rangle$ for $s$, provided as: (i) $\text{dom}(\Gamma_G) = \{ s[p] \mid p \in \text{roles}(G) \}$; (ii) $\forall s[p] \in \text{dom}(\Gamma_G) : \Gamma_G(s[p]) = \langle \mathcal{V}_p, T_p \rangle$; (iii) $\forall p \in \text{roles}(G) : G[p] \leq T_p$; and (iv) $\mathcal{V} = \bigcup_{p \in \text{roles}(G)} \mathcal{V}_p$ (recall that $\sqcup$ is an overriding union).

2. $\Gamma_D$ is associated with $G$ for $s$, given as follows:
   (i) $\text{dom}(\Gamma_D) = \{ s[p] \mid p \in \text{roles}(G) \}$;
   (ii) $\forall s[p] \in \text{dom}(\Gamma_D) : \Gamma_D(s[p]) = M_p$;
   (iii) if $G = \text{end}$ or $G = \mu t.G', \Gamma_D(s[p])$ associates with $G'$ for $s$;
   (iv) if $G = p \rightarrow q : \{ s_i(S_i) \delta_i \lambda_i \}, G_i \in I$, then (a1) $q \notin \text{receivers}(\Gamma_D(s[p]))$,
       and (a2) $\forall i \in I : \Gamma_D$ is associated with $G_i$ for $s$;
   (v) if $G = p \rightarrow q : \{ s_i(S_i) \delta_i \lambda_i \}, G_i \in I$, then (b1) $\Gamma_D(s[p]) = q ! \lambda_j(S_j) : M$
       with $S_j \leq S_i$, and (b2) $\Gamma_D(s[p] \rightarrow M) = \Gamma_D$ is associated with $G_j$ for $s$.

3. $\forall s[p] \in \text{dom}(\Gamma_{\text{end}}) : \Gamma_{\text{end}}(s[p]) = \emptyset$ ($\mathcal{V}, \text{end}$).

The association $\sqsubseteq_s$ is a binary relation over typed global types $\langle \mathcal{V}; G \rangle$ and typing environments $\Gamma$, parameterised by multiparty sessions $s$. There are three requirements for the association: (1) the typing environment $\Gamma$ 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 $\Gamma$ reflect $\langle \mathcal{V}; G \rangle$ by ensuring that: a. they align with the projections of $G$ via subtyping, and b. their clock valuations match $\mathcal{V}_s$; and (3) the queue type entries in $\Gamma$ correspond to the transmissions en route in $G$. Note that $\Gamma_{\text{end}}$ is specifically used to associate typing environments and end-types $\langle \mathcal{V}; \text{end} \rangle$, as in this case, both $\Gamma_G$ and $\Gamma_D$ are empty.

**Example 12.** Consider the timed global type $\{ \{ C_{\text{Sen}} = 0, C_{\text{Sat}} = 0, C_{\text{ser}} = 0 \}; G_{\text{data}} \}$, where $G_{\text{data}}$ is from Ex. 7, and a typing environment $\Gamma_{\text{data}} = \Gamma_{G_{\text{data}}}, \Gamma_{\text{data}}$, where:

- $\Gamma_{G_{\text{data}}} = s[\text{Sen}]: \{ s[\text{Sen}] = 0 \}; \Gamma_{\text{data}}$,
- $\text{Sat} : \{ \text{Sat} : s[\text{Sat}] = 0 \}; \Gamma_{\text{data}}$.

$\Gamma_{\text{data}}$ is associated with $\{ \{ C_{\text{Sen}} = 0, C_{\text{Sat}} = 0, C_{\text{ser}} = 0 \}; G_{\text{data}} \}$ for $s$, which can be formally verified by ensuring that $\Gamma_{\text{data}}$ 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 $\langle \mathcal{V}; G \rangle$ and typing environment $\Gamma : \langle \mathcal{V}; G \rangle \sqsubseteq_s \Gamma$. If $\Gamma \Rightarrow \Gamma'$, then there exists $\langle \mathcal{V}'; G' \rangle$ such that $\langle \mathcal{V}; G \rangle \Rightarrow \langle \mathcal{V}'; G' \rangle$ and $\langle \mathcal{V}'; G' \rangle \sqsubseteq_s \Gamma'$.

**Theorem 14 (Soundness of Association).** Given associated timed global type $\langle \mathcal{V}; G \rangle$ and typing environment $\Gamma : \langle \mathcal{V}; G \rangle \sqsubseteq_s \Gamma$. If $\langle \mathcal{V}; G \rangle \Rightarrow \Gamma$, then there exists $\alpha$, $\langle \mathcal{V}''; G'' \rangle$, $\Gamma'$, and $\Gamma''$, such that $\langle \mathcal{V}'; G \rangle \sqsubseteq_s \Gamma'$, $\langle \mathcal{V}''; G'' \rangle \Rightarrow \langle \mathcal{V}''; G'' \rangle$, $\Gamma' \Rightarrow \Gamma''$, and $\langle \mathcal{V}''; G'' \rangle \sqsubseteq_s \Gamma''$. 
We now present a typing system for ATMP, which relies on typing judgments of the form:

\[ \Theta \cdot \Gamma \vdash P \]  

(with \( \Theta \) omitted when empty)

This judgement indicates that the process \( P \) adheres to the usage of its variables and channels as specified in \( \Gamma \) (Def. 8), guided by the process types in \( \Theta \) (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 \( c : (V, T) \). Rule \([T-end]\) introduces a predicate \( \text{end}() \) for typing environments, signifying the termination of all endpoints. This predicate is used in \([T-o]\) to type an inactive process \( \bot \). Rules \([T-def]\) and \([T-X-Call]\) deal with recursive processes declarations and calls, respectively. Rule \([T-\neg]\) partitions the typing environment into two, each dedicated to typing one sub-process.

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

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

**Timed Branching and Selection**: Rules \([T-\&]\) and \([T-\neg]\) are for timed branching and selection, respectively. We elaborate on \([T-\&]\), as \([T-\neg]\) is its dual. The first premise in \([T-\&]\) specifies a time interval \([V, V + n]\) within which the message must be received, in accordance with each \( \delta_i \). 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 \([V, V + n]\). Here, the clock valuation \( V \) is reset based on each \( \lambda_i \). The remaining premises stipulate that the clock valuation \( V' \) of each delegated receiving session must satisfy \( \delta'_i \), 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. \([T-Kill]\)
\[\begin{align*}
\Theta(X) = (V_1, T_1), \ldots, (V_n, T_n) \\
\Theta \vdash X : (V_1, T_1), \ldots, (V_n, T_n) & \quad [T-X] \\
\forall i \in 1..n \quad c_i : (V_i, T_i) \vdash c_i : (V_i, \text{end}) & \quad [T-end] \\
\Theta, X : (V_1, T_1), \ldots, (V_n, T_n) \vdash P & \quad [T-ref] \\
\Theta, X : (V_1, T_1), \ldots, (V_n, T_n), \Gamma \vdash Q & \quad [T-def] \\
(V, T) \in (V', T') & \quad [T-Sub] \\
\Theta, \Gamma \vdash 0 & \quad [\text{T-0}] \\
\Theta, \Gamma \vdash \text{delay}(t) & \quad [T-\delta] \\
\Theta, \Gamma \vdash \text{delay}(\delta) & \quad [T-t] \\
\Theta, \Gamma \vdash P & \quad [T-\bullet] \\
\Theta, \Gamma \vdash \text{try} \quad P \quad \text{catch} \quad Q & \quad [T-Try] \\
\Theta, \Gamma \vdash \text{timeout}[P] & \quad [T-Failed] \\
\Theta, \Gamma \vdash s[p] \quad \Theta \vdash s[p] & \quad [T-\varepsilon] \\
\Theta, \Gamma \vdash s[p] \quad \Theta \vdash s[p] & \quad [T-\tau] \\
\Theta, \Gamma \vdash s[p] \quad \Theta \vdash s[p] & \quad [T-Cancel] \\
\Theta, \Gamma \vdash s[p] & \quad [T-Kill] \\
\Theta, \Gamma \vdash s[p] \quad \Theta \vdash s[p] & \quad [T-\mu-G] \\
\end{align*}\]

\textbf{Figure 8:} ATMP typing rules.

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

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

\textbf{Queue:} Rules [T-\varepsilon] and [T-\tau] concern the typing of queues. [T-\varepsilon] types an empty queue under an ended typing environment, while [T-\tau] types a non-empty queue by inserting a message type into \(\Gamma\). This insertion may either prepend the message to an existing queue type in \(\Gamma\) or add a queue-typed entry to \(\Gamma\) if not present.

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

\subsection{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 $\Gamma$ 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 $[G\nu\cdot G]$ in Fig. 8.

**Theorem 17 (Subject Reduction).** Assume $\Theta \cdot \Gamma \vdash P$ where $\forall s \in \Gamma : \exists (\forall; G) : (\forall; G) \sqsubseteq \Gamma_s$. If $P \rightarrow P'$, then $\exists \Gamma'$ such that $\Gamma \rightarrow^* \Gamma'$, $\Theta \cdot \Gamma' \vdash P'$, and $\forall s \in \Gamma' : \exists (\forall'; G') : (\forall'; G') \sqsubseteq \Gamma'_s$.

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

**Example 19.** Take the typed process $Q_{\text{sen}} | Q_{\text{sat}} | Q_{\text{ser}}$ and the typing environment $\Gamma_{\text{data}}$ from Exs. 4, 12, and 16. After a reduction using $[\text{r-delay}]$, $Q_{\text{sen}} | Q_{\text{sat}} | Q_{\text{ser}}$ transitions to $\text{delay}(6.5) : Q_{\text{sen}} | \text{delay}(6) : Q_{\text{sat}} | \text{delay}(6) : Q_{\text{ser}}$. This remains typable by $\Gamma_{\text{data}}$. Then, applying $[\text{r-tee}]$, $Q_{\text{2}}$ evolves to $\Psi_{0.5}(Q_{\text{2}})$, typed as $\Gamma_{\text{data}}$. Further reduction through $[\text{r-fan}]$ leads $\Psi_{0.5}(Q_{\text{2}}) \rightarrow Q_{\text{ser}} | \text{delay}(6.5) : Q_{\text{sat}} | \text{delay}(6.5) : Q_{\text{ser}}$. Later, via $[\text{r-fan}]$, $Q_{\text{3}}$ reduces to $\text{cancel}(s) | \text{delay}(6.5) : Q_{\text{ser}} | \text{delay}(6.5) : Q_{\text{sat}}$. This can be typed by $\Gamma_{\text{data}}$, obtained from $\Gamma_{\text{data}}$. This establishes 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 $\text{def } X(x_1; (V_1, T_1), \ldots, x_n; (V_n, T_n)) = Q$ in $P'$, for all $i \in 1..n$, $T_i \not\equiv \text{end}$ implies that a call $Y(..., x_i, ...)$ can only occur in $Q$ as a subterm of $x_i^n[q] \sum_{j \in J} m_j(y_j)P_j$ or $x_i^n[q] \oplus m(d).P''$ (i.e. after using $x_i$ for input or output);

2. only plays role $p$ in $s$, by $\Gamma$, if and only if (i) $P$ has guarded definitions; (ii) $\nu s(P) = \emptyset$; (iii) $\Gamma = \Gamma_0, s[p] ; \tau$ with $\tau \not\equiv (\forall', \text{end})$ and $\text{end}(\Gamma_0)$; (iv) in all subterms $(\nu's ; \Gamma')P'$ of $P$, we have $\Gamma' \leftrightarrow s'[p] ; \tau ; (\forall', \text{end})$ or $\Gamma' \leftrightarrow s'[p] ; (\forall', \text{end})$ (for some $p', \forall'$).

We say “$P$ only plays role $p$ 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 $Q\hat{\xi}$ 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 \( \langle \mathcal{V} ; G \rangle \subseteq_s \Gamma \), \( P \equiv \Pi_{p \in I} P_p \), \( Q^\delta \), and \( \Gamma = \bigcup_{p \in I} \Gamma_p \cup \Gamma_0 \), such that \( \emptyset \cdot \Gamma_0 \vdash Q^\gamma \), and for each \( P_p \) : (1) \( \emptyset \cdot \Gamma_p \vdash P_p \), and (2) either \( P_p \equiv 0 \), or \( P_p \) only plays role \( p \) in \( s \), by \( \Gamma_p \). Then, \( \Gamma \rightarrow_s \) implies \( \exists \Gamma', \langle \mathcal{V}' ; G' \rangle \), \( P' \) such that \( \Gamma \rightarrow_s \Gamma' \), \( P \rightarrow^\delta P' \), and \( \emptyset \cdot \Gamma' \vdash P' \), with \( \langle \mathcal{V}' ; G' \rangle \subseteq_s \Gamma' \), \( P' \equiv \Pi_{p \in I} P_p' \), \( Q'^\delta \), and \( \Gamma' = \bigcup_{p \in I} \Gamma_p' \cup \Gamma_0' \) such that \( \emptyset \cdot \Gamma_0' \vdash Q'^\gamma \), and for each \( P_p' \) : (1) \( \emptyset \cdot \Gamma_p' \vdash P_p' \), and (2) either \( P_p' \equiv 0 \), or \( P_p' \) only plays role \( p \) in \( s \), by \( \Gamma_p' \).

**Example 22.** Consider the processes \( Q_{\text{Sen}}, Q_{\text{Sat}}, Q_{\text{Ser}} \) from Ex. 4, the process \( Q_3 \) from Ex. 19, and the typing environment \( \Gamma_{\text{data}} \) from Ex. 12. \( Q_{\text{Sen}}, Q_{\text{Sat}}, \) and \( Q_{\text{Ser}} \) only play roles \( \text{Sen}, \text{Sat}, \) and \( \text{Ser} \), 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 \( \Gamma_{\text{data}} + 6.5 \xrightarrow{s\text{SenSatData}} \Gamma_{\text{data}}' \), there exists \( Q_4 \), resulting from \( Q_3 \rightarrow Q_4 \) via \( \text{R-Out} \), such that \( \Gamma_{\text{data}}' \) can type \( Q_4 \), with \( \Gamma_{\text{data}}' \) 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 \( \Gamma \) associated with a timed global type \( \langle \mathcal{V}^0 ; G \rangle \), is deadlock-free.

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

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

**Example 25.** Given the processes \( Q_{\text{Sen}}, Q_{\text{Sat}}, \) and \( Q_{\text{Ser}} \) from Ex. 4, along with the typing environment \( \Gamma_{\text{data}} \) from Ex. 12, \( (\nu s) \Gamma_{\text{data}}' \) \( Q_{\text{Sen}} \mid Q_{\text{Sat}} \mid Q_{\text{Ser}} \) 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 \( (\delta \text{ in ATMP}) \) 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\textsuperscript{T} involves using two libraries: the `crossbeam_channel` RUST library\textsuperscript{[9]} for asynchronous messaging, and the RUST standard library time\textsuperscript{[39]} for handling and comparing virtual clocks.

**5.1 Time Bounds in MultiCrusty\textsuperscript{T}**

**Implementing Time Bounds** To demonstrate the integration of time bounds in MultiCrusty\textsuperscript{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 \( C_{Sat2} \), which is not reset afterward.

In MultiCrusty\textsuperscript{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 \( C_{Sat2} \) 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,’ ‘,End>`. 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\textsuperscript{[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\textsuperscript{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\textsuperscript{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.
29:22  Fearless Asynchronous Communications with Timed Multiparty Session Protocols

- **Table 1** Primitives available in MultiCrusty\(^T\).

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>let s = s.send(p, clocks)?;</code></td>
</tr>
<tr>
<td>2</td>
<td>If allowed by the time constraint compared to the given clock in <code>clocks</code>, sends a payload <code>p</code> on a channel <code>s</code> and assigns the continuation of the session to a new variable <code>s</code></td>
</tr>
<tr>
<td>3</td>
<td><code>let (p, a) = s.recv(clocks)?;</code></td>
</tr>
<tr>
<td>4</td>
<td>If allowed by the time constraint compared to the given clock in <code>clocks</code>, receives a payload <code>p</code> on channel <code>s</code> and assigns the continuation of the session to a new variable <code>s</code></td>
</tr>
<tr>
<td>5</td>
<td><code>s.close()</code></td>
</tr>
<tr>
<td>6</td>
<td>Closes the channel <code>s</code> and returns a unit type</td>
</tr>
<tr>
<td>7</td>
<td><code>fn endpoint_data_ser( </code></td>
</tr>
<tr>
<td>8</td>
<td><code>s: EndpointSerData, clocks: </code>&lt;</td>
</tr>
</tbody>
</table>
| 9    | `clocks: &mut HashMap<char, Instant>>> ( 
| 10   | `let s = s.send(GetData {}, clocks)?;` |
| 11   | `let (data, a) = 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 $\nu$Scr$^T$.

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 $\text{Ok}()$, 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 $\nu$Scr$^T$, a multiparty protocol description language, with time features, resulting in $\nu$Scr$^T$. Additional keywords such as within, using, and and resetting are incorporated in $\nu$Scr to support the specification of time windows, clocks, and resets, respectively. In Fig. 10, we illustrate the $\nu$Scr$^T$ protocol for remote data, showcasing the application of these enhancements. $\nu$Scr$^T$ 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
message through roles, and the \textit{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 \textit{ring} protocol, compile-time benchmarks (Fig. 11b) indicate that \textsf{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), \textsf{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 \textit{mesh} protocol, where all roles send and receive messages (compile-time benchmarks in Fig. 11a and runtime benchmarks in Fig. 11c), \textsf{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 \textsf{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, \textsf{MultiCrusty}T demonstrates a growing overhead, mainly attributed to the incorporation of additional time constraint checks.

\subsection{6.2 Expressivity and Feasibility with Case Studies}

We implement a variety of protocols to showcase the expressivity, feasibility, and capabilities of \textsf{MultiCrusty}T, conducting benchmarking using both \textsf{MultiCrusty}T and \textsf{MultiCrusty}. The send and recv operations in both libraries are ordered, directed, and involve the same set of participants. Additionally, when implemented with \textsf{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 \textsf{MultiCrusty}T (\textcolor{blue}{
\textbullet
}) and \textsf{MultiCrusty} (\textcolor{red}{
\textbullet
}), 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 \textsf{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 \cite{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\textsuperscript{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\textsuperscript{T}, is demonstrated in Fig. 11 (bottom). The results indicate that MultiCrusty\textsuperscript{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.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|c|}
\hline
Protocol & Generated Types & Implemented Lines of Code & MP & Branching & Rec & Time Constraints \\
\hline
Calculator \cite{20} & 52 & 51 & ✓ & ✓ & ✓ & 11 \\
Online wallet \cite{33} & 142 & 160 & ✓ & ✓ & ✓ & 24 \\
SMTP \cite{36} & 331 & 475 & ✓ & ✓ & ✓ & 98 \\
Simple voting \cite{20} & 73 & 96 & ✓ & ✓ & ✓ & 16 \\
Three buyers \cite{24} & 108 & 78 & ✓ & ✓ & ✓ & 22 \\
Travel agency \cite{21} & 118 & 128 & ✓ & ✓ & ✓ & 30 \\
oAuth \cite{53} & 199 & 89 & ✓ & ✓ & ✓ & 30 \\
HTTP \cite{13} & 648 & 610 & ✓ & ✓ & ✓ & 126 \\
Remote data \cite{7} & 100 & 119 & ✓ & ✓ & ✓ & 16 \\
Servo \cite{38} & 74 & 48 & ✓ & ✓ & ✓ & 10 \\
Gravity sensor \cite{2} & 61 & 95 & ✓ & ✓ & ✓ & 9 \\
PineTime Heart Rate \cite{35} & 101 & 111 & ✓ & ✓ & ✓ & 17 \\
Proximity Based Car Key \cite{41} & 70 & 134 & ✓ & ✓ & ✓ & 22 \\
\hline
\end{tabular}
\caption{Metrics for protocols implemented in MultiCrusty\textsuperscript{T}.}
\end{table}

7 Related Work and Conclusion

\textbf{Time in Session Types} Bocchi et al. \cite{4} propose a timed extension of MPST to model real-time choreographic interactions, while Bocchi et al. \cite{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. \cite{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. \cite{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 ($\odot$) 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 SILL.

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


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


