On Asynchronous Multiparty Session Types for Federated Learning

Ivan Prokić[1][0000−0001−5420−1527], Simona Prokić[1][0000−0002−7161−3926], Silvia Ghilezan[1,2][0000−0003−2253−8285], Alceste Scalas[3][0000−0002−1153−6164], and Nobuko Yoshida[4][0000−0002−3925−8557]

[1] Faculty of Technical Sciences, University of Novi Sad, Serbia
[2] Mathematical Institute of the Serbian Academy of Sciences and Arts, Belgrade, Serbia, {prokic,simona.k,gsilvia}@uns.ac.rs
[3] Technical University of Denmark, DK, alcsc@dtu.dk
[4] University of Oxford, UK, nobuko.yoshida@cs.ox.ac.uk

Abstract. This paper improves the session typing theory to support the modelling and verification of processes that implement federated learning protocols. To this end, we build upon the asynchronous "bottom-up" session typing approach by adding support for input/output operations directed towards multiple participants at the same time. We further enhance the flexibility of our typing discipline and allow for safe process replacements by introducing a session subtyping relation tailored for this setting. We formally prove safety, deadlock-freedom, liveness, and session fidelity properties for our session typing system. Moreover, we highlight the nuances of our session typing system, which (compared to previous work) reveals interesting interplays and trade-offs between safety, liveness, and the flexibility of the subtyping relation.

Keywords: Multiparty session types · Federated learning · π-calculus · Type systems.

1 Introduction

Asynchronous multiparty session types [9] provide a formal approach to the verification of message-passing programs. The key idea is to express *protocols as types*, and use type checking to verify whether one or more communicating processes correctly implement some desired protocols. To enhance usability of session type theory in real-world applications, many extensions and variations of the approach have been proposed over the years [10,20,6,14,24,12,25]. However, these extensions remain insufficient for several important applications, including Federated learning (FL). FL is a distributed machine learning setting where clients train a model while keeping the training data decentralized [15,4,3,27,18,19]. In FL, communication protocols follow key patterns that must be expressed for proper modeling and verification. For instance, some initial investigations [18,19] note that using existing session type theories to model and verify FL protocols can be challenging due to presence of "arbitrary order of message arrivals". We now explain the nature of these challenges.

Modelling an asynchronous centralised federated learning protocol. Originally, FL considered centralised approach [15,4,18,19], where, in *phase 1*, a central *server* distributes the learning model to the *clients*. In *phase 2*, the clients receive the model, train it locally, and send the updated version back to the server. Finally, the server aggregates the updates and obtains an improved model. As a concrete example, consider a single round of a *generic centralised one-shot federated learning algorithm (FLA)* [18,19] having one server and two clients. We may model the processes for server p and two clients q and r with a value passing labeled π-calculus as follows — where ld and upd are message labels meaning "local data" and "update," $\sum$ represents a choice, and and $\mathsf{q}!ld\langle\ldots\rangle$ (resp. $\mathsf{q}?ld(\ldots)$) means "send (resp. receive) the message ld to (resp. from) participant q."

$$P = \mathsf{q}!ld\langle\mathsf{data}\rangle \,.\, \mathsf{r}!ld\langle\mathsf{data}\rangle . \sum\begin{Bmatrix}\mathsf{q}?upd(x) \,.\, \mathsf{r}?upd(y)\\ \mathsf{r}?upd(y) \,.\, \mathsf{q}?upd(x)\end{Bmatrix}$$

$$Q = R = \mathsf{p}?ld(x).\mathsf{p}!upd\langle\mathsf{data}\rangle$$

In phase 1, p's process P above sends its local data (i.e., a machine learning model) to q and then to r. In phase 2, server p receives the update upd from q and also r; this can happen in two possible orders depending on whether the data is received from q or r first (i.e., arbitrary order of message arrivals), and this is represented by the two branches of the choice $\sum$. Clients q and r receive server's local data and reply with update.

Furthermore, suppose we have implemented the above process specifications and verified that they behave as intended. Now, we want to upgrade the participant q to support for *multi-model FL* [4], in which a client can train multiple models, but only one per round (due to computational constraints). Assuming q can train two models ld and ld', we may model an updated process Q'.

$$Q' = \sum\begin{Bmatrix}\mathsf{p}?ld(x).\mathsf{p}!upd\langle\mathsf{data}\rangle\\ \mathsf{p}?ld'(y).\mathsf{p}!upd'\langle\mathsf{data}'\rangle\end{Bmatrix}$$

This rises a question: *Can we safely substitute Q with Q' in the protocol without needing to re-verify the entire protocol?*

Modelling an asynchronous decentralised federated learning protocol. Now let us consider a class of asynchronous decentralised federated learning (DFL) protocols that rely on a fully connected network topology (in which direct links exist between all pairs of nodes) [3,27,18,19]. Concretely, we consider a single round of a *generic decentralised one-shot federated learning algorithm (FLA)* [18,19] having three nodes, where there is no central point of control. Each participant in this algorithm follows the same behaviour divided in three phases. In *phase 1*, each participant sends its local data (i.e., a machine learning model) to all other participants. In *phase 2*, each participant receives the other participants' local data, trains the algorithm, and sends back the updated data. Finally, in *phase 3*, each participant receives the updated data from other participants and aggregates the updates.

Again, we may model the process for participant p with a value passing labeled π-calculus.

$$P_{12} = \mathsf{q}!\mathit{ld}\langle\mathsf{data}\rangle \,.\, \mathsf{r}!\mathit{ld}\langle\mathsf{data}\rangle . \sum\begin{Bmatrix}\mathsf{q}?\mathit{ld}(x) \,.\, \mathsf{q}!\mathit{upd}\langle\mathsf{data}\rangle \,.\, \mathsf{r}?\mathit{ld}(y) \,.\, \mathsf{r}!\mathit{upd}\langle\mathsf{data}\rangle . P_3 \\ \mathsf{r}?\mathit{ld}(y) \,.\, \mathsf{r}!\mathit{upd}\langle\mathsf{data}\rangle \,.\, \mathsf{q}?\mathit{ld}(x) \,.\, \mathsf{q}!\mathit{upd}\langle\mathsf{data}\rangle . P_3\end{Bmatrix}$$

$$P_3 = \sum\begin{Bmatrix}\mathsf{q}?\mathit{upd}(x) \,.\, \mathsf{r}?\mathit{upd}(y) \\ \mathsf{r}?\mathit{upd}(y) \,.\, \mathsf{q}?\mathit{upd}(x)\end{Bmatrix}$$

In phase 1, p's process P_{12} above sends its local data to q and then to r. In phase 2 (starting with the sum), participant p receives local data and replies the update *upd* to q and also r; this can happen in two possible orders depending on whether the data is received from q or r first. Finally, in phase 3 (in process P_3), p receives update data from q and r in any order (again, this is represented with two choice branches).

Notice that each participant here exhibits an arbitrary order of message arrivals, and that no single participant guides the protocol. This can be challenging to model using session type theories that rely on global types.

Contributions and outline of the paper. We present a novel "bottom-up" asynchronous session typing theory (in the style of [20], i.e., that does not require global types) that supports input/output operations directed towards multiple participants. This enables the modeling of arbitrary message arrival orders. We demonstrate that our approach enables the modelling and enhances verification of processes implementing federated learning protocols (both centralized and decentralized) by abstracting protocol behavior to the level of types. We enhance flexibility of our typing discipline and allow for safe process replacements by introducing a session subtyping relation tailored for this setting. Furthermore, we formalise and prove safety, deadlock-freedom, liveness, and session fidelity properties for well-typed processes, revealing interesting dependencies between these properties in the presence of a subtyping relation.

The paper is organized as follows: Section 2 introduces the asynchronous session calculus; Section 3 presents the types and subtyping relation; Section 4 details the type system, our main results, and the implementation of federated learning protocols; and finally, Section 5 concludes with a discussion of related and future work.

2 The Calculus

In this section we present the syntax and operational semantics of the value-passing labeled π-calculus used in this work (we omit session creation and delegation). The syntax of the calculus is defined in Table 1. Values can be either variables (x, y, z) or constants (positive integers or booleans).

Our **asynchronous multiparty sessions** (ranged over by $\mathcal{M}, \mathcal{M}', \ldots$) represent a parallel composition of **participants** (ranged over by $\mathsf{p}, \mathsf{q}, \ldots$) assigned with their **process** P and **output message queue** h (notation: $\mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h$). Here, $\mathsf{p} \triangleleft h$ denotes that h is the output message queue of participant p, and the **queued message** $(\mathsf{q}, \ell(\mathsf{v}))$ represents that participant p has sent message

$$
\begin{array}{rll}
\mathsf{v} ::= & x, y, z, \ldots \;\mid\; 1, 2, \ldots \;\mid\; \mathsf{true}, \mathsf{false} & \textit{(variables, values)} \\
\mathcal{M} ::= & \mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h \;\mid\; \mathcal{M} \mid \mathcal{M} & \textit{(participant, parallel)} \\
h ::= & \varnothing \;\mid\; (\mathsf{q}, \ell(\mathsf{v})) \;\mid\; h \cdot h & \textit{(empty, message, concatenation)} \\
P, Q ::= & \sum_{i \in I} \mathsf{p}_i?\ell_i(x_i).P_i \;\mid\; \sum_{i \in I} \mathsf{p}_i!\ell_i\langle \mathsf{v}_i \rangle.P_i & \textit{(external choice, internal choice)} \\
& \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q & \textit{(conditional)} \\
& X \;\mid\; \mu X.P \;\mid\; \mathbf{0} & \textit{(variable, recursion, inaction)}
\end{array}
$$

Fig. 1: Syntax of sessions, processes, and queues.

labeled ℓ with payload v to q. In the syntax of processes, the **external choice** $\sum_{i \in I} \mathsf{p}_i?\ell_i(x_i).P_i$ denotes receiving from participant p_i a message labelled ℓ_i with a value that should replace variable x_i in P_i, for any $i \in I$. The **internal choice** $\sum_{i \in I} \mathsf{p}_i!\ell_i\langle \mathsf{v}_i \rangle.P_i$ denotes sending to participant p_i a message labelled ℓ_i with value v_i, and then continuing as P_i, for any $i \in I$; we will see that, when the internal choice has more than one branch, then one of them is picked nondeterministically. In both external and internal choices, we assume $(\mathsf{p}_i, \ell_i) \neq (\mathsf{p}_j, \ell_j)$, for all $i, j \in I$, such that $i \neq j$. The **conditional** construct, process **variable**, **recursion**, and **inaction 0** are standard; we will sometimes omit writing $\mathbf{0}$. As usual, we assume that recursion is guarded, i.e., in the process $\mu X.P$, the variable X can occur in P only under an internal or external choice.

We show how this syntax can be used to compactly model two federated learning protocols [18,19]: a centralized (Example 1) and a decentralized (Example 2). First, we set up some notation.

We define a **concurrent input** macro $\|R_j\|_{j \in I}.Q$ to represent a process that awaits a series of incoming messages arriving in arbitrary order:

$$
R ::= \mathsf{p}?l(x) \;\mid\; \mathsf{p}?l(x).R^{?!} \qquad R^{?!} ::= \mathsf{p}?l(x) \;\mid\; \mathsf{p}!l\langle \mathsf{v} \rangle \;\mid\; \mathsf{p}?l(x).R^{?!} \;\mid\; \mathsf{p}!l\langle \mathsf{v} \rangle.R^{?!}
$$

$$
\|R_i\|_{i \in I}.Q \;=\; \textstyle\sum_{i \in I} R_i.\|R_j\|_{j \in I \setminus \{i\}}.Q \quad \text{and} \quad \|R_j\|_{j \in \{i\}}.Q \;=\; R_i.Q
$$

where R_i are all process prefixes starting with an input, possibly followed by other inputs or outputs. The concurrent input $\|R_i\|_{i \in I}.Q$ specifies an external choice of any input-prefixed R_i, for $i \in I$, followed by another choice of input-prefixed process with an index from $I \setminus \{i\}$; the composition continues until all $i \in I$ are covered, with process Q added at the end. In this way, concurrent input process $\|R_i\|_{i \in I}.Q$ can perform the actions specified by any R_i (for $i \in I$) in an arbitrary order, depending on which inputs become available first; afterwards, process Q is always executed. For instance, using our concurrent input macro, we may represent the FLA process P_3 from Section 1 as $P_3 = \|\mathsf{q}?\mathit{upd}(x),\ \mathsf{r}?\mathit{upd}(y)\|$.

Example 1 (Modelling a centralised FL protocol implementation). As specified in [18,19], the *generic centralized one-shot federated learning protocol* is implemented with one server and $n-1$ clients (Section 1 presented the processes of the protocol for $n = 3$). We may model an implementation of this protocol as a session $\mathcal{M} = \prod_{i \in I}(\mathsf{p}_i \triangleleft P_i \;\mid\; \mathsf{p}_i \triangleleft \varnothing)$, where $I = \{1, 2 \ldots, n\}$, and where p_1 plays

$$
\begin{array}{ll}
[\textsc{r-send}] & \mathsf{p} \triangleleft \sum_{i\in I} \mathsf{q}_i!\ell_i\langle \mathsf{v}_i\rangle.P_i \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathcal{M} \\
& \qquad \xrightarrow{\mathsf{p}:\mathsf{q}_k!\ell_k} \mathsf{p} \triangleleft P_k \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \cdot (\mathsf{q}_k, \ell_k(\mathsf{v}_k)) \mid \mathcal{M} \qquad (k \in I) \\
[\textsc{r-rcv}] & \mathsf{p} \triangleleft \sum_{i\in I} \mathsf{q}_i?\ell_i(x_i).P_i \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathsf{q} \triangleleft Q \mid \mathsf{q} \triangleleft (\mathsf{p}, \ell(\mathsf{v})) \cdot h \mid \mathcal{M} \\
& \quad \xrightarrow{\mathsf{p}:\mathsf{q}?\ell} \mathsf{p} \triangleleft P_k\{^{\mathsf{v}}/_{x_k}\} \mid \mathsf{p} \triangleleft h_{\mathsf{p}} \mid \mathsf{q} \triangleleft Q \mid \mathsf{q} \triangleleft h \mid \mathcal{M} \quad (\exists k{\in}I : (\mathsf{q}, \ell) = (\mathsf{q}_k, \ell_k)) \\
[\textsc{r-cond-T}] & \mathsf{p} \triangleleft \mathsf{if\ true\ then}\ P\ \mathsf{else}\ Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M} \xrightarrow{\mathsf{p}:\mathsf{if}} \mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h \mid \mathcal{M} \\
[\textsc{r-cond-F}] & \mathsf{p} \triangleleft \mathsf{if\ false\ then}\ P\ \mathsf{else}\ Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M} \xrightarrow{\mathsf{p}:\mathsf{if}} \mathsf{p} \triangleleft Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M} \\
[\textsc{r-struct}] & \mathcal{M}_1 \Rrightarrow \mathcal{M}_1' \ \text{ and }\ \mathcal{M}_1' \longrightarrow \mathcal{M}_2' \ \text{ and }\ \mathcal{M}_2' \Rrightarrow \mathcal{M}_2 \quad \Longrightarrow \quad \mathcal{M}_1 \longrightarrow \mathcal{M}_2
\end{array}
$$

Fig. 2: Reduction relation on sessions.

the role of the server, while the rest of participants are clients, defined with:

$$
\begin{aligned}
P_1 &= \mathsf{p}_2!ld\langle \mathsf{data}\rangle. \ldots \mathsf{p}_n!ld\langle \mathsf{data}\rangle. \| \mathsf{p}_2?upd(x_2), \ldots, \mathsf{p}_n?upd(x_n) \| \\
P_i &= \mathsf{p}_1?ld(x).\mathsf{p}_1!upd\langle \mathsf{data}\rangle \qquad \text{for } i = 2, \ldots, n
\end{aligned}
$$

Notice that, after sending the data to all the clients, the server P_1 then receives the updates from all of them in an arbitrary order. The clients first receive the data and then reply the update back to the server.

Example 2 (Modelling a decentralised FL protocol implementation). As specified in [18,19], an implementation of the *generic decentralized one-shot federated learning protocol* comprises n participant processes acting both as servers and clients (Section 1 presented one process of the protocol for $n = 3$). We may model an implementation of this protocol with a session $\mathcal{M} = \prod_{i\in I}(\mathsf{p}_i \triangleleft P_i \mid \mathsf{p}_i \triangleleft \varnothing)$, where $I = \{1, 2 \ldots, n\}$, and where process P_1 is defined as follows: (the rest of the processes are defined analogously)

$$
\begin{aligned}
P_1 = {} & \mathsf{p}_2!ld\langle \mathsf{data}\rangle. \ldots \mathsf{p}_n!ld\langle \mathsf{data}\rangle. \\
& \| \mathsf{p}_2?ld(x_2).\mathsf{p}_2!upd\langle \mathsf{data}\rangle, \ldots, \mathsf{p}_n?ld(x_n).\mathsf{p}_n!upd\langle \mathsf{data}\rangle \|. \\
& \| \mathsf{p}_2?upd(y_2), \ldots, \mathsf{p}_n?upd(y_n) \|
\end{aligned}
$$

Process P_1 specifies that participant p_1 first sends its local data to all other participants. Then, p_1 receives local data from all other participants and replies the update concurrently (i.e., in an arbitrary order). Finally, p_1 receives the updates from all other participants, again in an arbitrary order.

Session Reductions. The **reduction relation** for our process calculus is defined in Figure 2. Rule [r-send] specifies sending one of the messages from the internal choice, while the other choices are discarded; the message ℓ_k with payload v_k sent to participant q_k is appended to the participant p's output queue, and p's process becomes the continuation P_k. Dually, rule [r-rcv] defines how a participant p can receive a message, directed towards p with a supported label, from the head of the output queue of the sender q; after the reduction, the message is removed from the queue and the received message then substitutes the placeholder variable x_k in the continuation process P_k. Observe that, by rules

[R-SEND] and [R-RCV], output queues are used in FIFO order. Rules [R-COND-T] and [R-COND-F] define how to reduce a conditional process: the former when the condition is true, the latter when it is false. Rule [R-STRUCT] closes the reduction relation under a standard precongruence relation $\Rrightarrow$ (defined as expected) that allows reordering of messages in queues and unfolding recursive processes, combining the approach of [26] with [7].

Properties. In Definition 1 below we follow the approach of [7] and define how "good" sessions are expected to run by using behavioural properties. We define three behavioural properties. The first one is *safety*, ensuring that a session never has mismatches between the message labels supported by external choices and the labels of incoming messages. Since, in our sessions, one participant can choose to receive messages from multiple senders at once, our definition of safety requires external choices to support all possible message labels from all senders' queues. The *deadlock freedom* property requires that a session can get stuck (cannot reduce further) only in case it terminates. The *liveness* property ensures all pending inputs eventually receive a message and all queued messages are eventually received, if reduction steps are performed in a *fair* manner.

The fair path definition says that whenever a participant p is ready to perform an output in a session $\mathcal{M}_i$, then there is a session $\mathcal{M}_k$ (reached by reducing $\mathcal{M}_i$) where p actually performs an output. Dually, if a participant p in $\mathcal{M}_i$ is ready to receive a message which is already available at the top of the sender's output queue, then p will receive that message in $\mathcal{M}_k$. This avoids unfair paths, e.g., in which two participants recursively exchange messages, while a third participant forever waits to send a message. Our fair and live properties for session types with standard choices matches the liveness defined in [7, Definition 2.2]. Also, our fairness aligns with "fairness of components" according to [8] - where a "component" is a process in a session. Moreover, by the syntax and semantics of our processes, fairness of components coincides with justness [8].

Definition 1 (Session behavioral properties). *A* ***session path*** *is a (possibly infinite) sequence of sessions $(\mathcal{M}_i)_{i\in I}$, where $I = \{0,1,2,\ldots,n\}$ (or $I = \{0,1,2,\ldots\}$) is a set of consecutive natural numbers, and, $\forall i \in I\backslash\{n\}$ (or $\forall i \in I$), $\mathcal{M}_i \longrightarrow \mathcal{M}_{i+1}$. We say that a path $(\mathcal{M}_i)_{i\in I}$ is* ***safe*** *iff, $\forall i \in I$:*

(SS) *if $\mathcal{M}_i \Rrightarrow \mathsf{p}\triangleleft\sum_{j\in J}\mathsf{q}_j?\ell_j(x_j).P_j \mid \mathsf{p}\triangleleft h \mid \mathsf{q}\triangleleft Q \mid \mathsf{q}\triangleleft h_{\mathsf{q}} \mid \mathcal{M}'$ with $\mathsf{q} \in \{\mathsf{q}_j\}_{j\in J}$ and $h_{\mathsf{q}} \equiv (\mathsf{p}, \ell(\mathsf{v}))\cdot h'_{\mathsf{q}}$, then $(\mathsf{q},\ell) = (\mathsf{q}_j,\ell_j)$, for some $j \in J$*

We say that a path $(\mathcal{M}_i)_{i\in I}$ is ***deadlock-free*** *iff:*

(SD) *if $I = \{0,1,2,\ldots,n\}$ and $\mathcal{M}_n \not\longrightarrow$ then $\mathcal{M}_n \Rrightarrow \mathsf{p}\triangleleft\mathbf{0} \mid \mathsf{p}\triangleleft\varnothing$*

We say that a path $(\mathcal{M}_i)_{i\in I}$ is ***fair*** *iff, $\forall i \in I$:*

(SF1) *if $\mathcal{M}_i \xrightarrow{\mathsf{p:q!}\ell} \mathcal{M}'$, then $\exists k,j$ such that $I \ni k \geq i$ and $\mathcal{M}_k \xrightarrow{\mathsf{p:q}_j\mathsf{!}\ell_j} \mathcal{M}_{k+1}$*

(SF2) *if $\mathcal{M}_i \xrightarrow{\mathsf{p:q?}\ell} \mathcal{M}'$, $\exists k,j$ such that $I \ni k \geq i$ and $\mathcal{M}_k \xrightarrow{\mathsf{p:q}_j\mathsf{?}\ell_j} \mathcal{M}_{k+1}$*

(SF3) *if $\mathcal{M}_i \xrightarrow{\mathsf{p:if}} \mathcal{M}'$, $\exists k$ such that $I \ni k \geq i$ and $\mathcal{M}_k \xrightarrow{\mathsf{p:if}} \mathcal{M}_{k+1}$*

We say that a session path $(\mathcal{M}_i)_{i\in I}$ is ***live*** *iff, $\forall i \in I$:*

(SL1) *if* $\mathcal{M}_i \Rrightarrow \mathsf{p} \triangleleft \mathsf{if}\ \mathsf{v}\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M}'$*, then* $\exists k$ *such that* $I \ni k \geq i$ *and, for some* $\mathcal{M}''$*, we have either* $\mathcal{M}_k \Rrightarrow \mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft h \mid \mathcal{M}''$ *or* $\mathcal{M}_k \Rrightarrow \mathsf{p} \triangleleft Q \mid \mathsf{p} \triangleleft h \mid \mathcal{M}''$

(SL2) *if* $\mathcal{M}_i \Rrightarrow \mathsf{p} \triangleleft \sum_{j \in J} \mathsf{q}_j!\ell_j\langle \mathsf{v}_j \rangle.P_j \mid \mathsf{p} \triangleleft h \mid \mathcal{M}'$*, then* $\exists k$ *such that* $I \ni k \geq i$ *and* $\mathcal{M}_k \xrightarrow{\mathsf{p}:\mathsf{q}_j!\ell_j} \mathcal{M}_{k+1}$*, for some* $j \in J$

(SL3) *if* $\mathcal{M}_i \Rrightarrow \mathsf{p} \triangleleft P \mid \mathsf{p} \triangleleft (\mathsf{q}, \ell(\mathsf{v})) \cdot h \mid \mathcal{M}'$*, then* $\exists k$ *such that* $I \ni k \geq i$ *and* $\mathcal{M}_k \xrightarrow{\mathsf{q}:\mathsf{p}?\ell} \mathcal{M}_{k+1}$

(SL4) *if* $\mathcal{M}_i \Rrightarrow \mathsf{p} \triangleleft \sum_{j \in J} \mathsf{q}_j?\ell_j(x_j).P_j \mid \mathsf{p} \triangleleft h \mid \mathcal{M}'$*, then* $\exists k$ *such that* $I \ni k \geq i$*, and* $\mathcal{M}_k \xrightarrow{\mathsf{p}:\mathsf{q}?\ell} \mathcal{M}_{k+1}$*, where* $(\mathsf{q}, \ell) \in \{(\mathsf{q}_j, \ell_j)\}_{j \in J}$

We say that a session $\mathcal{M}$ *is* ***safe/deadlock-free*** *iff all paths beginning with* $\mathcal{M}$ *are safe/deadlock-free. We say that a session* $\mathcal{M}$ *is* ***live*** *iff all fair paths beginning with* $\mathcal{M}$ *are live.*

Like in standard session types [20], in our calculus safety does not imply liveness nor deadlock-freedom (Example 3). Also, liveness implies deadlock-freedom, since liveness requires that session cannot get stuck before all external choices are performed and queued messages are eventually received. The converse is not true: e.g., a session in which two participants recursively exchange messages is deadlock free (as it never gets stuck), even if a third participant waits forever to receive a message that nobody will send. This session is not live, since in any fair path the third participant never fires its actions.

However, *unlike* standard session types, in our calculus liveness does *not* imply safety: this is illustrated in Example 4 below.

Example 3 (A safe but non-live session). Consider session

$$\begin{aligned}\mathcal{M} = \mathsf{p} \triangleleft \sum\{&\mathsf{q}?\ell_1(x).\mathsf{r}?\ell_2(y),\ \mathsf{r}?\ell_2(x),\ \mathsf{r}?\ell_3(x)\} \mid \mathsf{p} \triangleleft \varnothing \\ &\mid \mathsf{q} \triangleleft \mathbf{0} \mid \mathsf{q} \triangleleft (\mathsf{p}, \ell_1(\mathsf{v}_1)) \mid \mathsf{r} \triangleleft \mathbf{0} \mid \mathsf{r} \triangleleft (\mathsf{p}, \ell_2(\mathsf{v}_2))\end{aligned}$$

In the session, participant p is ready to receive an input either from q or r, which, in turn, both have enqueued messages for p. The labels of enqueued messages are safely supported in p's receive. Session $\mathcal{M}$ has two possible reductions, where p receives either from q, and $\mathcal{M} \longrightarrow \mathsf{p} \triangleleft \mathsf{r}?\ell_2(y) \mid \mathsf{p} \triangleleft \varnothing \mid \mathsf{r} \triangleleft \mathbf{0} \mid \mathsf{r} \triangleleft (\mathsf{p}, \ell_2(\mathsf{v}_2)) \longrightarrow \mathsf{p} \triangleleft \mathbf{0} \mid \mathsf{p} \triangleleft \varnothing$; or from r, in which case $\mathcal{M} \longrightarrow \mathsf{q} \triangleleft \mathbf{0} \mid \mathsf{q} \triangleleft (\mathsf{p}, \ell_1(\mathsf{v}_1))$.

Hence, session $\mathcal{M}$ is safe, since in all reductions inputs and matching enqueued messages have matching labels. However, $\mathcal{M}$ is not live, since the second path above starting with $\mathcal{M}$ is not live, for q's output queue contains an orphan message that cannot be received. Notice also that $\mathcal{M}$ is not deadlock-free since the final session in the second path cannot reduce further but is not equivalent to a terminated session.

Example 4 (A live but unsafe session). Consider the following session:

$$\begin{aligned}\mathcal{M}' = \mathsf{p} \triangleleft \sum\{&\mathsf{q}?\ell_1(x).\mathsf{r}?\ell_2(y),\ \mathsf{r}?\ell_3(x)\} \mid \mathsf{p} \triangleleft \varnothing \\ &\mid \mathsf{q} \triangleleft \mathbf{0} \mid \mathsf{q} \triangleleft (\mathsf{p}, \ell_1(\mathsf{v}_1)) \mid \mathsf{r} \triangleleft \mathbf{0} \mid \mathsf{r} \triangleleft (\mathsf{p}, \ell_2(\mathsf{v}_2))\end{aligned}$$

The session $\mathcal{M}'$ is not safe since the message in r's output queue has label ℓ_2 which is not supported in p's external choice (that only supports label ℓ_3 for receiving from r). However, $\mathcal{M}'$ has a single reduction path where p receives from q, and then p receives from r; then, the session ends with all processes being $\mathbf{0}$ and all queues empty, which implies $\mathcal{M}'$ is live (and thus, also deadlock-free).

3 Types and Typing Environments

We now introduce our (local) session types, typing contexts and their properties, and subtyping. Our types are a blend of asynchronous multiparty session types [7] and the separated choice multiparty sessions (SCMP) types from [16].

Definition 2. *The* ***sorts*** S *are defined as* $\mathsf{S} ::= \mathtt{nat} \mid \mathtt{bool}$*, and* ***(local) session types*** T *are defined as:*

$$\mathsf{T} \;::=\; \sum_{i\in I} \mathsf{p}_i!\ell_i\langle \mathsf{S}_i\rangle.\mathsf{T}_i \;\mid\; \sum_{i\in I} \mathsf{p}_i?\ell_i(\mathsf{S}_i).\mathsf{T}_i \;\mid\; \mathtt{end} \;\mid\; \mu \boldsymbol{t}.\mathsf{T} \;\mid\; \boldsymbol{t}$$

where $(\mathsf{p}_i, \ell_i) \neq (\mathsf{p}_j, \ell_j)$*, for all* $i, j \in I$ *such that* $i \neq j$*. The* ***queue types*** σ *and* ***typing environments*** Γ *are defined as:*

$$\sigma \;::=\; \epsilon \;\mid\; \mathsf{p}!\ell\langle \mathsf{S}\rangle \;\mid\; \sigma\cdot\sigma \qquad\qquad \Gamma \;::=\; \emptyset \;\mid\; \Gamma, \mathsf{p} : (\sigma, \mathsf{T})$$

Sorts are the types of values, which can be natural numbers ($\mathtt{nat}$) or booleans ($\mathtt{bool}$). The *internal choice* session type $\sum_{i\in I} \mathsf{p}_i!\ell_i\langle \mathsf{S}_i\rangle.\mathsf{T}_i$ describes an output of message ℓ_i with sort S_i towards participant p_i and then evolving to type T_i, for some $i \in I$. Similarly, $\sum_{i\in I} \mathsf{p}_i?\ell_i(\mathsf{S}_i).\mathsf{T}_i$ stands for *external choice*, i.e., receiving a message ℓ_i with sort S_i from participant p_i, for some $i \in I$. The type $\mathtt{end}$ denotes the terminated session type, $\mu\mathbf{t}.\mathsf{T}$ is a recursive type, and $\mathbf{t}$ is a recursive type variable. We assume that all recursions are guarded and follow a form of Barendregt convention: every $\mu\mathbf{t}.T$ binds a syntactically distinct $\mathbf{t}$.

The *queue type* represents the type of the messages contained in an output queue; it can be empty (ϵ), or it can contain message ℓ with sort S for participant p ($\mathsf{p}!\ell\langle\mathsf{S}\rangle$), or it can be a concatenation of two queue types ($\sigma\cdot\sigma$). The *typing environment* assigns a pair of queue/session type to a participant ($\mathsf{p} : (\sigma, \mathsf{T})$). We use $\Gamma(\mathsf{p})$ to denote the type that Γ assigns to p.

Typing Environment Reductions. Now we define typing environment reductions, which relies on a structural congruence relation $\equiv$ over session types, queue types, and typing environments (defined as expected).

Definition 3. *The typing environment reduction* $\xrightarrow{\alpha}$*, with* α *being either* $\mathsf{p}{:}\mathsf{q}?\ell$ *or* $\mathsf{p}{:}\mathsf{q}!\ell$ *(for some* $\mathsf{p}, \mathsf{q}, \ell$*), is inductively defined as follows:*

$$[\text{E-RCV}]\quad \mathsf{p}_k{:}\,(\mathsf{q}!\ell_k\langle\mathsf{S}_k\rangle\cdot\sigma, \mathsf{T}_\mathsf{p}), \mathsf{q}{:}\,(\sigma_\mathsf{q}, \sum_{i\in I}\mathsf{p}_i?\ell_i(\mathsf{S}_i).\mathsf{T}_i), \Gamma \xrightarrow{\mathsf{q}:\mathsf{p}_k?\ell_k} \mathsf{p}_k{:}\,(\sigma, \mathsf{T}_\mathsf{p}), \mathsf{q}{:}\,(\sigma_\mathsf{q}, \mathsf{T}_k), \Gamma \qquad (k\in I)$$

$$[\text{E-SEND}]\quad \mathsf{p} : (\sigma, \sum_{i\in I}\mathsf{q}_i!\ell_i\langle\mathsf{S}_i\rangle.\mathsf{T}_i), \Gamma \xrightarrow{\mathsf{p}:\mathsf{q}_k!\ell_k} \mathsf{p} : (\sigma\cdot\mathsf{q}_k!\ell_k\langle\mathsf{S}_k\rangle, \mathsf{T}_k), \Gamma \qquad (k \in I)$$

$$[\text{E-STRUCT}]\quad \Gamma \equiv \Gamma_1 \xrightarrow{\alpha} \Gamma_1' \equiv \Gamma' \implies \Gamma \xrightarrow{\alpha} \Gamma'$$

We use $\Gamma \longrightarrow \Gamma'$ *instead of* $\Gamma \xrightarrow{\alpha} \Gamma'$ *when* α *is not relevant, and* $\longrightarrow^*$ *for the reflexive and transitive closure of* $\longrightarrow$*.* $\Gamma \longrightarrow$ *denotes* $\Gamma \longrightarrow \Gamma'$*, for some* Γ'*.*

In rule [E-RCV] an environment has a reduction step labeled $\mathsf{q}{:}\mathsf{p}_k?\ell_k$ if participant p_k has at the head of its queue a message for q with label ℓ_k and payload sort S_k, and q has an external choice type that includes participant p_k with label ℓ_k and a corresponding sort S_k; the environment evolves by consuming p_k's message and activating the continuation T_k in q's type. Rule [E-SEND] specifies reduction if p has an internal choice type where p sends a message toward q_k, for some $k \in I$; the reduction is labelled $\mathsf{p}{:}\mathsf{q}_k!\ell_k$ and the message is placed at the end of p's queue. Rule [E-STRUCT] is standard closure of the reduction relation under structural congruence.

Properties. Similarly to [20], we define behavioral properties also for typing environments (and their reductions). As for processes, we use three properties for typing environments. *Safety* ensures that the typing environment never has label or sort mismatches. In our setting, where one participant can receive from more than one queue, this property ensures safe receptions of queued messages. The second property, *deadlock freedom*, ensures that typing environment which can not reduce further must be terminated. The third property is *liveness*, which, for the case of standard session types (as in [7]), matches the liveness defined in [7, Definition 4.7]: it ensures all pending inputs eventually receive a message and all queued messages are eventually received, if reduction steps are performed *fairly*.

Notably, our definition of deadlock-freedom and liveness also require safety. The reason for this is that liveness and deadlock-freedom properties for typing environments, if defined without assuming safety, are not preserved by the subtyping (introduced in the next section, see Example 9). We use the predicate over typing environments $\mathsf{end}(\Gamma)$ (read "Γ **is terminated**") that holds iff, for all $p \in dom(\Gamma)$, we have $\Gamma(\mathsf{p}) \equiv (\epsilon, \mathsf{end})$.

Definition 4 (Typing environment properties). *A* typing environment path *is a (possibly infinite) sequence of typing environments $(\Gamma_i)_{i\in I}$, where $I = \{0, 1, 2, \ldots, n\}$ (or $I = \{0, 1, 2, \ldots\}$) is a set of consecutive natural numbers, and, $\forall i \in I\backslash\{n\}$ (or $\forall i \in I$), $\Gamma_i \longrightarrow \Gamma_{i+1}$. We say a path $(\Gamma_i)_{i\in I}$ is **safe** iff, $\forall i \in I$:*

(TS) *if $\Gamma_i(\mathsf{p}) \equiv (\sigma_\mathsf{p}, \sum_{j\in J} \mathsf{q}_j?\ell_j(\mathsf{S}_j).\mathsf{T}_j)$ and $\Gamma_i(\mathsf{q}_k) \equiv (\mathsf{p}!\ell_k\langle\mathsf{S}_k\rangle\cdot\sigma_\mathsf{q}, \mathsf{T}_\mathsf{q})$, with $\mathsf{q}_k \in \{\mathsf{q}_j\}_{j\in J}$, then $\exists j \in J : (\mathsf{q}_j, \ell_j, \mathsf{S}_j) = (\mathsf{q}_k, \ell_k, \mathsf{S}_k)$*

*We say that a path $(\Gamma_i)_{i\in I}$ is **deadlock-free** iff:*

(TD) *if $I = \{0, 1, \ldots, n\}$ and $\Gamma_n \not\longrightarrow$ then $\mathsf{end}(\Gamma_n)$*

*We say that a path $(\Gamma_i)_{i\in I}$ is **fair** iff, $\forall i \in I$:*

(TF1) *if $\Gamma_i \xrightarrow{\mathsf{p:q!}\ell} \Gamma'$, then $\exists k, \mathsf{q}', \ell'$ such that $I \ni k \geq i$ and $\Gamma_k \xrightarrow{\mathsf{p:q'!}\ell'} \Gamma_{k+1}$*

(TF2) *if $\Gamma_i \xrightarrow{\mathsf{p:q?}\ell} \Gamma'$, then $\exists k, \mathsf{q}', \ell'$ such that $I \ni k \geq i$ and $\Gamma_k \xrightarrow{\mathsf{p:q'?}\ell'} \Gamma_{k+1}$*

*We say that a path $(\Gamma_i)_{i\in I}$ is **live** iff, $\forall i \in I$:*

(TL1) *if $\Gamma_i(\mathsf{p}) \equiv (\mathsf{q}!\ell\langle\mathsf{S}\rangle \cdot \sigma\, ,\, \mathsf{T})$, then $\exists k$ such that $I \ni k \geq i$ and $\Gamma_k \xrightarrow{\mathsf{q:p?}\ell} \Gamma_{k+1}$*

(TL2) *if $\Gamma_i(\mathsf{p}) \equiv \left(\sigma_\mathsf{p}\, ,\, \sum_{j\in J} \mathsf{q}_j?\ell_j(\mathsf{S}_j).\mathsf{T}_j\right)$, then $\exists k, \mathsf{q}, \ell$ such that $I \ni k \geq i$, $(\mathsf{q}, \ell) \in \{(\mathsf{q}_j, \ell_j)\}_{j\in J}$, and $\Gamma_k \xrightarrow{\mathsf{p:q?}\ell} \Gamma_{k+1}$*

A typing environment Γ is ***safe*** *iff all paths starting with Γ are safe. A typing environment Γ is* ***deadlock-free*** *iff all paths starting with Γ are safe and deadlock-free. We say that a typing environment Γ is* ***live*** *iff it is safe and all fair paths beginning with Γ are live.*

Since our deadlock-freedom and liveness for typing environments subsumes safety, we do not have the situation in which typing environment is deadlock-free and/or live but not safe. Still, we can have typing environments that are safe but not deadlock-free or live.

Example 5. Consider typing environment

$$\Gamma = \{\mathsf{p}\colon (\epsilon, \sum\{\mathsf{q}?\ell_1(\mathsf{S}_1).\mathsf{r}?\ell_2(\mathsf{S}_2),\ \mathsf{r}?\ell_2(\mathsf{S}_2),\ \mathsf{r}?\ell_3(\mathsf{S}_3)\},$$
$$\mathsf{q}\colon (\mathsf{p}!\ell_1\langle\mathsf{S}_1\rangle, \mathsf{end}), \mathsf{r}\colon (\mathsf{p}!\ell_2\langle\mathsf{S}_2\rangle, \mathsf{end})\}$$

Here, Γ is safe by Definition 4, but is not live nor deadlock-free, because the path in which p receives from r message ℓ_2 leads to a typing environment that cannot reduce further but is not terminated. Now consider typing environment

$$\Gamma' = \{\mathsf{p}\colon (\epsilon, \sum\{\mathsf{q}?\ell_1(\mathsf{S}_1).\mathsf{r}?\ell_2(\mathsf{S}_2),\ \mathsf{r}?\ell_3(\mathsf{S}_3)\}, \mathsf{q}\colon (\mathsf{p}!\ell_1\langle\mathsf{S}_1\rangle, \mathsf{end}), \mathsf{r}\colon (\mathsf{p}!\ell_2\langle\mathsf{S}_2\rangle, \mathsf{end})\}$$

Γ' is not safe by Definition 4 (hence, also not deadlock-free nor live) since there is a mismatch between r sending ℓ_2 to p, while p can only receive ℓ_3 from r.

We now prove that all three behavioral properties are closed under structural congruence and reductions.

Proposition 1. *If Γ is safe/deadlock-free/live and $\Gamma \equiv \Gamma'$ or $\Gamma \longrightarrow \Gamma'$, then Γ' is safe/deadlock-free/live.*

3.1 Subtyping

In Definition 5 below we formalise a standard "synchronous" subtyping relation as in [16], not dealing with the possible reorderings of outputs and inputs allowed by asynchronous subtyping [7].

Definition 5. *The* subtyping $\leqslant$ *is coinductively defined as:*

$$\frac{\forall i \in I \quad \mathsf{T}_i \leqslant \mathsf{T}'_i \quad \{\mathsf{p}_i\}_{i\in I} = \{\mathsf{p}_i\}_{i\in I\cup J}}{\sum_{i\in I} \mathsf{p}_i!\ell_i\langle\mathsf{S}_i\rangle.\mathsf{T}_i \leqslant \sum_{i\in I\cup J} \mathsf{p}_i!\ell_i\langle\mathsf{S}_i\rangle.\mathsf{T}'_i}\ [\text{S-OUT}]$$

$$\frac{\forall i \in I \quad \mathsf{T}_i \leqslant \mathsf{T}'_i \quad \{\mathsf{p}_i\}_{i\in I} = \{\mathsf{p}_i\}_{i\in I\cup J}}{\sum_{i\in I\cup J} \mathsf{p}_i?\ell_i(\mathsf{S}_i).\mathsf{T}_i \leqslant \sum_{i\in I} \mathsf{p}_i?\ell_i(\mathsf{S}_i).\mathsf{T}'_i}\ [\text{S-IN}]$$

$$\frac{\mathsf{T}_1\{\mu\mathbf{t}.\mathsf{T}_1/\mathbf{t}\} \leqslant \mathsf{T}_2}{\mu\mathbf{t}.\mathsf{T}_1 \leqslant \mathsf{T}_2}\ [\text{S-MUL}] \qquad \frac{\mathsf{T}_1 \leqslant \mathsf{T}_2\{\mu\mathbf{t}.\mathsf{T}_2/\mathbf{t}\}}{\mathsf{T}_1 \leqslant \mu\mathbf{t}.\mathsf{T}_2}\ [\text{S-MUR}] \qquad \frac{}{\mathsf{end} \leqslant \mathsf{end}}\ [\text{S-END}]$$

Pair of queue/session types are related via subtyping $(\sigma_1, \mathsf{T}_1) \leqslant (\sigma_2, \mathsf{T}_2)$ iff $\sigma_1 = \sigma_2$ and $\mathsf{T}_1 \leqslant \mathsf{T}_2$. We define $\Gamma \leqslant \Gamma'$ iff $dom(\Gamma) = dom(\Gamma')$ and $\forall \mathsf{p} \in dom(\Gamma) : \Gamma(\mathsf{p}) \leqslant \Gamma'(\mathsf{p})$.

The subtyping rules allow less branches for the subtype in the internal choice (in [s-out]), and more branches in the external choice (in [s-in]). The side condition in both rules ensures the set of participants specified in the subtype and supertype choices is the same. Hence, subtyping allows flexibility in the set of labels, not in the set of participants. Subtyping holds up to unfolding (by [s-muL] and [s-muR]), as usual for coinductive subtyping [17, Chapter 21]. By rule [s-end], end is related via subtyping to itself.

Example 6. Consider the typing environments Γ and Γ' from Example 5. Since $\sum\{\mathsf{q}?\ell_1(\mathsf{S}_1).\mathsf{r}?\ell_2(\mathsf{S}_2),\ \mathsf{r}?\ell_2(\mathsf{S}_2),\ \mathsf{r}?\ell_3(\mathsf{S}_3)\} \leqslant \sum\{\mathsf{q}?\ell_1(\mathsf{S}_1).\mathsf{r}?\ell_2(\mathsf{S}_2),\ \mathsf{r}?\ell_3(\mathsf{S}_3)\}$ holds by [s-in], we also have that $\Gamma \leqslant \Gamma'$ holds.

The following two lemmas show that subtyping of safe typing environments is a simulation and that subtyping preserves all typing environment properties.

Lemma 1. *If $\Gamma' \leqslant \Gamma$, Γ is safe, and $\Gamma' \xrightarrow{\alpha} \Gamma_1'$, then there is Γ_1 such that $\Gamma_1' \leqslant \Gamma_1$ and $\Gamma \xrightarrow{\alpha} \Gamma_1$.*

Lemma 2. *If Γ is safe/deadlock-free/live and $\Gamma' \leqslant \Gamma$, then Γ' is safe/deadlock-free/live.*

Remark 1. If in Definition 4 safety is not assumed in deadlock-freedom and liveness, then Lemma 2 does not hold (see Example 6). The same observation also holds for deadlock-freedom in the synchronous case [16, Remark 3.2].

4 The Typing System and Its Properties

This section introduces the type system that assigns types to processes, queues, and sessions. Our typing system is an extension and adaptation of the one in [7].

Definition 6 (Typing system). *A* **shared typing environment** *Θ, which assigns sorts to expression variables, and (recursive) session types to process variables, is defined as $\Theta \ ::= \ \emptyset \ \mid \ \Theta, X : \mathsf{T} \ \mid \ \Theta, x : \mathsf{S}$.*

Our type system is inductively defined by the rules in Figure 3, with 4 judgements — which cover respectively values and variables v, processes P, message queues h, and sessions $\mathcal{M}$:

$$\Theta \vdash \mathsf{v} : \mathsf{S} \qquad \Theta \vdash P : \mathsf{T} \qquad \vdash h : \sigma \qquad \Gamma \vdash \mathcal{M}$$

By Definition 6, the typing judgment $\Theta \vdash \mathsf{v} : \mathsf{S}$ means that value v is of sort S under environment Θ. The judgement $\Theta \vdash P : \mathsf{T}$ says that process P behaves as prescribed with type T and uses its variables as given in Θ. The judgement $\vdash h : \sigma$ says that the messages in the queue h correspond to the queue type σ. Finally, $\Gamma \vdash \mathcal{M}$ means that all participant processes in session $\mathcal{M}$ behave as prescribed by the session types in Γ.

We now illustrate the typing rules in Figure 3. The first row gives rules for typing natural and boolean values and expression variables. The second row

$$\frac{}{\Theta \vdash 1, 2, \ldots : \mathtt{nat}}\,[\text{T-NAT}] \qquad \frac{}{\Theta \vdash \mathsf{true}, \mathsf{false} : \mathtt{bool}}\,[\text{T-Bool}] \qquad \frac{}{\Theta, x : \mathsf{S} \vdash x : \mathsf{S}}\,[\text{T-VAR}]$$

$$\frac{}{\vdash \varnothing : \epsilon}\,[\text{T-NUL}] \qquad \frac{\vdash \mathsf{v} : \mathsf{S}}{\vdash (\mathsf{q}, \ell(\mathsf{v})) : \mathsf{q}!\ell\langle \mathsf{S}\rangle}\,[\text{T-ELM}] \qquad \frac{\vdash h_1 : \sigma_1 \quad \vdash h_2 : \sigma_2}{\vdash h_1 \cdot h_2 : \sigma_1 \cdot \sigma_2}\,[\text{T-QUEUE}]$$

$$\frac{}{\Theta \vdash \mathbf{0} : \mathsf{end}}\,[\text{T-}\mathbf{0}] \qquad \frac{\forall i \in I \quad \Theta \vdash \mathsf{v}_i : \mathsf{S}_i \quad \Theta \vdash P_i : \mathsf{T}_i}{\Theta \vdash \sum_{i\in I} \mathsf{p}_i!\ell_i\langle \mathsf{v}_i\rangle.P_i : \sum_{i\in I} \mathsf{p}_i!\ell_i\langle \mathsf{S}_i\rangle.\mathsf{T}_i}\,[\text{T-OUT}]$$

$$\frac{\forall i \in I \quad \Theta, x_i : \mathsf{S}_i \vdash P_i : \mathsf{T}_i}{\Theta \vdash \sum_{i\in I} \mathsf{p}_i?\ell_i(x_i).P_i : \sum_{i\in I} \mathsf{p}_i?\ell_i(\mathsf{S}_i).\mathsf{T}_i}\,[\text{T-IN}]$$

$$\frac{\Theta \vdash \mathsf{v} : \mathtt{bool} \quad \Theta \vdash P_i : \mathsf{T} \ (i = 1, 2)}{\Theta \vdash \mathsf{if}\ \mathsf{v}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \mathsf{T}}\,[\text{T-COND}]$$

$$\frac{\Theta, X : \mathsf{T} \vdash P : \mathsf{T}}{\Theta \vdash \mu X.P : \mathsf{T}}\,[\text{T-REC}] \qquad \frac{}{\Theta, X : \mathsf{T} \vdash X : \mathsf{T}}\,[\text{T-VAR}] \qquad \frac{\Theta \vdash P : \mathsf{T} \quad \mathsf{T} \leqslant \mathsf{T}'}{\Theta \vdash P : \mathsf{T}'}\,[\text{T-SUB}]$$

$$\frac{\Gamma = \{\mathsf{p}_i : (\sigma_i, \mathsf{T}_i) \mid i \in I\} \quad \forall i \in I \quad \emptyset \vdash P_i : \mathsf{T}_i \quad \vdash h_i : \sigma_i}{\Gamma \vdash \prod_{i\in I}(\mathsf{p}_i \triangleleft P_i \mid \mathsf{p}_i \triangleleft h_i)}\,[\text{T-SESS}]$$

Fig. 3: Typing rules for values, queues, and for processes and sessions.

provides rules for typing message queues: an empty queue has the empty queue type (by [T-NUL]) while a queued message is typed if its payload has the correct sort (by [T-ELM]), and a queue obtained with concatenating two message queues is typed by concatenating their queue types (by [T-QUEUE]).

Rule [T-0] types a terminated process $\mathbf{0}$ with end. Rules [T-OUT]/[T-IN] type internal/external choice processes with the corresponding internal/external choice types: in each branch the payload v_i/x_i and the continuation process P_i have the corresponding sort S_i and continuation type T_i; observe that in [T-IN], each continuation process P_i is typed under environment Θ extended with the input-bound variable x_i having sort S_i. Rule [T-COND] types conditional: both branches must have the same type, and the condition must be boolean. Rule [T-REC] types a recursive process $\mu X.P$ with T if P has type T in an environment where that X has type T. Rule [T-VAR] types recursive process variables. Finally, [T-SUB] is the *subsumption rule*: a process P of type T can be typed with any supertype T'. This rule makes our type system equi-recursive [17], as $\leqslant$ relates types up-to unfolding (by Definition 5). Rule [T-SESS] types a session under environment Γ if each participant's queue and process have corresponding queue and process types in an empty Θ (which forces processes to be closed).

Example 7 (Types for a centralised FL protocol implementation). Consider session $\mathcal{M}$ from Example 1. Take that value data and variables $x, x_2, \ldots, x_n$ are of sort S. We may derive $\vdash P_1 : \mathsf{T}_1$ and $\vdash P_i : \mathsf{T}_2$ (for $i = 1, 2, \ldots, n$), where (with a slight abuse of notation using the concurrent input macro for types):

$$\mathsf{T}_1 = \mathsf{p}_2!ld\langle \mathsf{S}\rangle. \ldots \mathsf{p}_n!ld\langle \mathsf{S}\rangle.\|\mathsf{p}_2?upd(\mathsf{S}), \ldots, \mathsf{p}_n?upd(\mathsf{S})\|$$
$$\mathsf{T}_2 = \mathsf{p}_1?ld(\mathsf{S}).\mathsf{p}_1!upd\langle \mathsf{S}\rangle$$

Hence, with $\Gamma = \{\mathsf{p}_1 : (\epsilon, \mathsf{T}_1)\} \cup \{\mathsf{p}_i : (\epsilon, \mathsf{T}_2) \mid i = 2, \ldots, n\}$ we obtain $\Gamma \vdash \mathcal{M}$. Observe that Γ is deadlock-free and live: after p_1 sends the data to all other participants, each participant receives and replies with an update, that p_1 finally receives in arbitrary order.

Furthermore, considering again the upgrade process Q' from Section 1 that allows multi-model FL. We may derive $\vdash Q' : \mathsf{T}'_2$, for

$$\mathsf{T}'_2 = \sum\{\mathsf{p}_1?ld(\mathsf{S}).\mathsf{p}_1!upd\langle\mathsf{S}\rangle,\ \mathsf{p}_1?ld'(\mathsf{S}).\mathsf{p}_1!upd'\langle\mathsf{S}\rangle\}$$

By our subtyping relation we have $\mathsf{T}'_2 \leqslant \mathsf{T}_2$. If we denote with Γ' typing environment where one of the clients has the upgraded type T'_2, and the rest is same as in Γ, we also have $\Gamma' \leqslant \Gamma$. Hence, by Lemma 2 we have that Γ' is also deadlock-free and live, confirming that Q can safely be replaced with Q' in this, but also in any other, session.

Example 8 (Types for a decentralised FL protocol implementation). Consider session $\mathcal{M}$ from Example 2. Take that value data and variables $x_2, \ldots, x_n, y_2, \ldots, y_n$ are of sort S. We may derive $\vdash P_1 : \mathsf{T}_1$ where (again with a slight abuse of notation using the concurrent input macro for types):

$$\begin{aligned}\mathsf{T}_1 = \mathsf{p}_2!ld\langle\mathsf{S}\rangle.\ldots\mathsf{p}_n!ld\langle\mathsf{S}\rangle.\|\mathsf{p}_2?ld(\mathsf{S}).\mathsf{p}_2!upd\langle\mathsf{S}\rangle, \ldots, \mathsf{p}_n?ld(\mathsf{S}).\mathsf{p}_n!upd\langle\mathsf{S}\rangle\|.\\ \|\mathsf{p}_2?upd(\mathsf{S}), \ldots, \mathsf{p}_n?upd(\mathsf{S})\|\end{aligned}$$

and similarly for other process we may derive the corresponding types so that $\vdash P_i : \mathsf{T}_i$, for $i = 1, 2, \ldots, n$. Thus, with $\Gamma = \{\mathsf{p}_i : (\epsilon, \mathsf{T}_i) \mid i = 1, \ldots, n\}$ we obtain $\Gamma \vdash \mathcal{M}$. Notice that Γ is safe, i.e., has no label or sort mismatches: participant p_i always receives from p_j message ld before up (as they are enqueued in this order). This also implies Γ is deadlock-free and live: a participant first asynchronously sends all its ld messages (phase 1), then awaits to receive ld messages from queues of all other participants (phase 2), and only then awaits to receive up messages (phase 3). Hence, all $n(n-1)$ of ld and also of up messages are sent/received, and a reduction path always ends with a terminated typing environment.

Remark 2 (On the decidability of typing environment properties). Under the "bottom-up approach" to session typing, typing environment properties such as deadlock freedom and liveness are generally undecidable, since two session types with unbounded message queues are sufficient to encode a Turing machine [1, Theorem 2.5]. Still, many practical protocols have bounded buffer sizes — and in particular, the buffer sizes for the FL protocols in [19] are given in the same paper. Therefore, Examples 7 and 8 yield finite-state typing environment whose behavioural properties are decidable and easily verified, e.g., via model checking.

Properties of our typing system. We now illustrate and prove the properties of our typing system in Definition 6. We aim to prove that if a session $\mathcal{M}$ is typed with safe/deadlock-free/live typing environment Γ, then so is $\mathcal{M}$. Along the way, we illustrate the subtle interplay between our deadlock freedom, liveness, and safety properties (Definition 4) and subtyping (Definition 5).

First, we introduce the subtyping-related Lemma 3 below, which holds directly by rule [T-SUB] and is important for proving Subject Reduction later on (Theorem 1). Based on this, in Example 9 we show why our definitions of deadlock-free and live typing environments (Definition 4) also require safety.

Lemma 3. *If $\Gamma \vdash \mathcal{M}$ and $\Gamma \leqslant \Gamma'$, then $\Gamma' \vdash \mathcal{M}$.*

Example 9. Consider the safe but not deadlock-free nor live session $\mathcal{M}$ from Example 3, and the typing environments Γ and Γ' from Example 5. It is straightforward to show that $\Gamma \vdash \mathcal{M}$, by Definition 6. Observe that, as shown in Example 6, we have $\Gamma \leqslant \Gamma'$; therefore, by Lemma 3 we also have $\Gamma' \vdash \mathcal{M}$. As noted in Example 5, the typing environment Γ' is not safe. Now, suppose that in Definition 4, deadlock-freedom and liveness of typing environments were defined without requiring safety (as for deadlock-freedom and liveness of sessions in Definition 1); also notice that Γ' has a single path that is deadlock-free and live, but not safe. Therefore, this change in Definition 4 would cause Γ', an unsafe but deadlock-free and live typing environment, to type $\mathcal{M}$, a session that is not deadlock-free nor live. This would hamper our goal of showing that if a session is typed by a deadlock-free/live typing environment, then the session is deadlock free/live.

Next we show our main theoretical results: subject reduction (a reduction of a typed session can be followed by its safe/deadlock-free/live typing environment); session fidelity (if the typing environemnt reduces so can the session); and type safety, deadlock-freedom, and liveness (if typing environment is safe/deadlock-free/live, then so is the session).

Theorem 1 (Subject Reduction). *Assume $\Gamma \vdash \mathcal{M}$ with Γ safe/deadlock-free/live and $\mathcal{M} \longrightarrow \mathcal{M}'$. Then, there is safe/deadlock-free/live type environment Γ' such that $\Gamma \longrightarrow^* \Gamma'$ and $\Gamma' \vdash \mathcal{M}'$.*

Theorem 2 (Type Safety). *If $\Gamma \vdash \mathcal{M}$ and Γ is safe, then $\mathcal{M}$ is safe.*

Theorem 3 (Session Fidelity). *Let $\Gamma \vdash \mathcal{M}$. If $\Gamma \longrightarrow$, then $\exists \Gamma', \mathcal{M}'$ such that $\Gamma \longrightarrow \Gamma'$ and $\mathcal{M} \longrightarrow^+ \mathcal{M}'$ and $\Gamma' \vdash \mathcal{M}'$.*

Theorem 4 (Deadlock freedom). *If $\Gamma \vdash \mathcal{M}$ and Γ is deadlock-free, then $\mathcal{M}$ is deadlock-free.*

Theorem 5 (Liveness). *If $\Gamma \vdash \mathcal{M}$ and Γ is live, then $\mathcal{M}$ is live.*

5 Related and Future Work

The original asynchronous multiparty session types [9] is "top-down," in the sense that it begins by specifying a *global type*, i.e., a choreographic formalisation of all the communications expected between the *participants* in the protocol; then, the global type is *projected* into a set of (local) session types, which can be used to type-check processes. There are many extensions to this "top-down" approach. [10,6] extend multiparty sessions to support protocols with optional and dynamic participants, allowing sender-driven choices directed toward multiple participants. This is used by other work [23,21,22] to express communication protocols utilized in distributed cloud and edge computing. The models in [14,24,12,13] generalize the notion of projection in asynchronous multiparty session types. Building on the global types of [10], they allow local types with internal and external choices directed at different participants, as we do in this paper. Nevertheless, these approaches are constrained by their reliance on global types and a projection function: the global type couples both sending and receiving in a single construct and enforces a protocol structure where a single participant drives the interaction. This results in at least one projected local type beginning with a receive action — unlike the local types for our decentralised FL protocol implementation (see Example 8).

An alternative "bottom-up" approach to session typing [20] removes the requirement of global types: instead, local session types are specified directly, and the properties of their composition (e.g. deadlock freedom and liveness) are checked and transferred to well-typed processes. The approach in [16] extends the "bottom-up" synchronous multiparty session model of [20], not only by introducing more flexible choices but also by supporting mixed choices — i.e., choices that combine inputs and outputs. In contrast to [16], which assumes synchronous communication, our theory supports asynchronous communication, which is essential for our motivating scenarios and running examples; moreover, [16] does not prove process liveness nor session fidelity.

The recent work [25] presents an automata-based approach for checking multiparty protocols. Unlike ours, [25] presents a top-down approach to session typing, providing a soundness and completeness result on its projection; moreover, [25] shows that its typing system is also usable in "bottom-up" fashion, like ours. Their global types are specified as Protocol State Machines (PSMs), featuring decoupled send-receive operations, allowing for a wider range of protocols to be specified. Their local type specifications are specified as Communicating State Machines (CSMs), whereas ours are the extended session types. Their sessions are represented, as ours, as π-calculus processes (theirs with delegation), and the type system relates sessions to CSMs, and has the following distinguishing points w.r.t. our paper. First, we provide a coinductively defined subtyping relation with a standard subsumption rule [T-SUB], while [25] has no subtyping relation: specifically, it embeds output subtyping (i.e., the selection of one among multiple possible outputs) within the typing rules, but does not support input subtyping. [25] explicitly notes that "there are subtleties for subtyping as one cannot simply add receives". This makes the interplay of safety, deadlock-freedom/liveness, and subtyping we pointed out in our paper not reproducible in their setting. Second, [25] proves a safety property (no label mismatches and terminated sessions not leaving orphan messages) and the progress (deadlock-freedom) property (if the type of the session is not final, the process can take a step). Their progress property is proven for the processes containing only one session (like ours) — but their results do not include a proof of liveness, unlike ours. Finally, [25] specifies "...unsafe communication: a process is stuck because all the queues it is waiting to receive from are not empty, but the labels of the first messages do not match any of the cases the process is expecting" — while in our case it is sufficient for one queue to have an unmatching message. This is a reason why [25] does not allow for subtyping, unlike our paper.

The correctness of the decentralized FL algorithm has been formally verified for deadlock-freedom and termination in [19,5] by using the Communicating Sequential Processes calculus (CSP) and the Process Analysis Toolkit (PAT) model checker. Our asynchronous multiparty session typing model abstracts protocol behavior at the type level, paving the way for more scalable and efficient techniques — not only for model checking, but also for broader verification and analysis applications.

Conclusions and future work. We present the first "bottom-up" asynchronous multiparty session typing theory that supports internal and external choices targeting multiple distinct participants. We introduce a process and type calculus, which we relate through a type system. We formally prove safety, deadlock-freedom, liveness, and session fidelity, highlighting interesting dependencies between these properties in the presence of a subtyping relation. Finally, we demonstrate how our model can represent a wide range of communication protocols, including those used in asynchronous decentralized federated learning.

For future work, we identify two main research directions. The first focuses on the fundamental properties of our approach, including: verifying typing environment properties via model checking, in the style of [20]; investigating decidable approximations of deadlock freedom and liveness of typing contexts (see Remark 2), e.g. with the approach of [11]; and investigating the expressiveness of our model based on the framework in [16]. The second direction explores the application of our model as a foundation for reasoning about federated-learning-specific properties, e.g.: handling crashes of arbitrary participants [2]; supporting optional participation [10,6]; ensuring that participants only receive data they are able to process; statically enforcing that only model parameters—not raw data—are exchanged to preserve privacy; guaranteeing sufficiently large server buffers to receive messages from all clients; and ensuring that all clients contribute equally to the algorithm.

Acknowledgments. This work was supported by Horizon EU project 101093006 *TaRDIS*, MSTDI (Contract No. 451-03-137/2025-03/200156) and FTS-UNS through the project *Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at FTS-UNS 2025* (No. 01-50/295), COST CA20111 *EuroProofNet*, and EPSRC grants EP/T006544/2, EP/N027833/2,EP/T014709/2, EP/V000462/1, EP/X015955/1, EP/Y005244/1 and ARIA.

References

1. Bartoletti, M., Scalas, A., Tuosto, E., Zunino, R.: Honesty by typing. Log. Methods Comput. Sci. **12**(4) (2016)
2. Barwell, A.D., Hou, P., Yoshida, N., Zhou, F.: Designing asynchronous multiparty protocols with crash-stop failures. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 1:1–1:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023), `https://doi.org/10.4230/LIPIcs.ECOOP.2023.1`

3. Beltrán, E.T.M., Pérez, M.Q., Sánchez, P.M.S., Bernal, S.L., Bovet, G., Pérez, M.G., Pérez, G.M., Celdrán, A.H.: Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges. IEEE Commun. Surv. Tutorials **25**(4), 2983–3013 (2023), `https://doi.org/10.1109/COMST.2023.3315746`
4. Bhuyan, N., Moharir, S.: Multi-model federated learning. In: 14th International Conference on COMmunication Systems & NETworkS, COMSNETS 2022, Bangalore, India, January 4-8, 2022. pp. 779–783. IEEE (2022), `https://doi.org/10.1109/COMSNETS53615.2022.9668435`
5. Djukic, M., Prokic, I., Popovic, M., Ghilezan, S., Popovic, M., Prokic, S.: Correct orchestration of federated learning generic algorithms: Python translation to CSP and verification by PAT. Int. J. Softw. Tools Technol. Transf. **27**(1), 21–34 (2025), `https://doi.org/10.1007/s10009-025-00795-0`
6. Gheri, L., Lanese, I., Sayers, N., Tuosto, E., Yoshida, N.: Design-by-contract for flexible multiparty session protocols. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 8:1–8:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), `https://doi.org/10.4230/LIPIcs.ECOOP.2022.8`
7. Ghilezan, S., Pantovic, J., Prokic, I., Scalas, A., Yoshida, N.: Precise subtyping for asynchronous multiparty sessions. ACM Trans. Comput. Log. **24**(2), 14:1–14:73 (2023), `https://doi.org/10.1145/3568422`
8. van Glabbeek, R., Höfner, P.: Progress, justness, and fairness. ACM Comput. Surv. **52**(4), 69:1–69:38 (2019), `https://doi.org/10.1145/3329125`
9. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016), `http://doi.acm.org/10.1145/2827695`
10. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: Huisman, M., Rubin, J. (eds.) Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10202, pp. 116–133. Springer (2017), `https://doi.org/10.1007/978-3-662-54494-5_7`
11. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. pp. 97–117 (2019). `https://doi.org/10.1007/978-3-030-25540-4_6`
12. Li, E., Stutz, F., Wies, T., Zufferey, D.: Complete multiparty session type projection with automata. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13966, pp. 350–373. Springer (2023), `https://doi.org/10.1007/978-3-031-37709-9_17`
13. Li, E., Stutz, F., Wies, T., Zufferey, D.: Characterizing implementability of global protocols with infinite states and data. CoRR **abs/2411.05722** (2024), `https://doi.org/10.48550/arXiv.2411.05722`
14. Majumdar, R., Mukund, M., Stutz, F., Zufferey, D.: Generalising projection in asynchronous multiparty session types. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 35:1–35:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021), `https://doi.org/10.4230/LIPIcs.CONCUR.2021.35`
15. McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In: Singh, A., Zhu, X.J. (eds.) Proceedings of the 20th International Conference on

Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA. Proceedings of Machine Learning Research, vol. 54, pp. 1273–1282. PMLR (2017), http://proceedings.mlr.press/v54/mcmahan17a.html

16. Peters, K., Yoshida, N.: Separation and encodability in mixed choice multiparty sessions. In: Sobocinski, P., Lago, U.D., Esparza, J. (eds.) Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2024, Tallinn, Estonia, July 8-11, 2024. pp. 62:1–62:15. ACM (2024), https://doi.org/10.1145/3661814.3662085
17. Pierce, B.C.: Types and programming languages. MIT Press (2002)
18. Popovic, M., Popovic, M., Kastelan, I., Djukic, M., Ghilezan, S.: A simple Python testbed for federated learning algorithms. In: 2023 Zooming Innovation in Consumer Technologies Conference (ZINC). pp. 148–153. IEEE (2023)
19. Prokic, I., Ghilezan, S., Kasterovic, S., Popovic, M., Popovic, M., Kastelan, I.: Correct orchestration of federated learning generic algorithms: Formalisation and verification in CSP. In: Kofron, J., Margaria, T., Seceleanu, C. (eds.) Engineering of Computer-Based Systems - 8th International Conference, ECBS 2023, Västerås, Sweden, October 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14390, pp. 274–288. Springer (2023), https://doi.org/10.1007/978-3-031-49252-5_25
20. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. PACMPL **3**(POPL), 30:1–30:29 (2019), https://doi.org/10.1145/3290343
21. Simic, M., Dedeic, J., Stojkov, M., Prokic, I.: A hierarchical namespace approach for multi-tenancy in distributed clouds. IEEE Access **12**, 32597–32617 (2024), https://doi.org/10.1109/ACCESS.2024.3369031
22. Simic, M., Dedeic, J., Stojkov, M., Prokic, I.: Data overlay mesh in distributed clouds allowing collaborative applications. IEEE Access **13**, 6180–6203 (2025), https://doi.org/10.1109/ACCESS.2024.3525336
23. Simic, M., Prokic, I., Dedeic, J., Sladic, G., Milosavljevic, B.: Towards edge computing as a service: Dynamic formation of the micro data-centers. IEEE Access **9**, 114468–114484 (2021), https://doi.org/10.1109/ACCESS.2021.3104475
24. Stutz, F.: Asynchronous multiparty session type implementability is decidable - lessons learned from message sequence charts. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 32:1–32:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023), https://doi.org/10.4230/LIPIcs.ECOOP.2023.32
25. Stutz, F., D'Osualdo, E.: An automata-theoretic basis for specification and type checking of multiparty protocols. In: Vafeiadis, V. (ed.) Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II. Lecture Notes in Computer Science, vol. 15695, pp. 314–346. Springer (2025), https://doi.org/10.1007/978-3-031-91121-7_13
26. Udomsrirungruang, T., Yoshida, N.: Top-down or bottom-up? complexity analyses of synchronous multiparty session types. Proc. ACM Program. Lang. **9**(POPL), 1040–1071 (2025), https://doi.org/10.1145/3704872
27. Yuan, L., Wang, Z., Sun, L., Yu, P.S., Brinton, C.G.: Decentralized federated learning: A survey and perspective. IEEE Internet Things J. **11**(21), 34617–34638 (2024), https://doi.org/10.1109/JIOT.2024.3407584