Dynamic Multirole Session Types

Pierre-Malo Deniélou  Nobuko Yoshida
Imperial College London

Abstract

Multiparty session types enforce structured safe communications between several participants, as long as their number is fixed when the session starts. In order to handle common distributed interaction patterns such as peer-to-peer protocols or cloud algorithms, we propose a new role-based multiparty session type theory where roles are defined as classes of local behaviours that an arbitrary number of participants can dynamically join and leave. We offer programmers a polling operation that gives access to the current set of a role’s participants in order to fork processes. Our type system with universal types for polling can handle this dynamism and retain type safety. A multiparty locking mechanism is introduced to provide communication safety, but also to ensure a stronger progress property for joining participants that has never been guaranteed in previous systems. Finally, we present some implementation mechanisms used in our prototype extension of ML.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.2 [Semantics of Programming Languages]: Process models

General Terms Theory, Types, Design

Keywords communications, roles, multiparty, session types, mobile processes, dynamic protocols, join-leave, peer-to-peer, auction.

1. Introduction

As a type foundation for structured distributed, communication-centred programming, session types [18, 31] have been studied over the last decade for a wide range of process calculi and programming languages. The original binary theory has been generalised to multiparty session types [19] in order to guarantee stronger conformance to stipulated session structures between cooperating multiple end-point participants. Since the first work [19] was proposed, the multiparty session type theory has been developed in process calculi [4, 10, 14, 22], and used in several different contexts such as distributed object communication optimisations [29], security [5, 9], design by contract [6], parallel and web service programming [25, 35, 36] and medical guidelines [23], some of which initiated industrial collaborations (see §6 and 7). While many interaction patterns can be captured in the existing multiparty sessions framework, there are significant limitations for describing and validating loosely-coupled, ungoverned, dynamic protocols, since the number of participants is required to be fixed both when the session is designed and when the session execution starts. This makes it unable to express interaction patterns frequently found in messaging oriented middleware and service-oriented computing.

The central underpinning of multiparty session types is that critical properties, such as communication safety (essentially a correspondence between send and receive) and deadlock-freedom, are guaranteed by the combination of two means: first, a static type-checking methodology based on the existence of a global type (a description of a multiparty protocol from a global viewpoint) and of its end-point projections — the global type is projected to end-point types against which processes can be efficiently type-checked; second, a synchronisation mechanism which ensures that all the well-behaved (i.e. well-typed) participants are actually present when the session starts. This paper introduces a new role-based multiparty session type system and synchronisation mechanism that, together, can specify, verify and govern dynamically evolving protocols.

In the rest of this section, we illustrate our motivation, approach and solutions through protocols of increasing complexity: (1) Map/Reduce (introduction of the notion of roles and universal quantification); (2) P2P chat (projection challenges) and (3) Auction (branching and communication safety).

(1) Map/Reduce We imagine a server that wants a task to be computed on a cluster made of three cluster clients: the server sends them jobs and they give back their answers. We give a picture illustrating this communication pattern and the corresponding global multiparty session type written in the original theory [19].

This session starts with the server sending asynchronously the messages Map to participants client1, client2 and client3. Each of them answers back with a message Reduce.1 Recursion μx denotes an unbounded number of repeated interactions.

The problem here is that such a session cannot start without one of the clients and, once running, is not able to handle a fourth client joining or one of the current clients leaving. In the original multiparty sessions, any of these scenarios requires ending the session, writing an appropriate global type for the new situation, and starting a fresh session again.

This paper proposes a theory of dynamic multirole session types that can describe global interactions between roles, which are classes of participants that share a common behaviour (e.g. the clients in the above example). Dynamism is disciplined by a simple universally quantified type that allows to spawn further interactions by polling the set of participants currently playing a given role.

In the above session, we notice that the three clients have the exact same behaviour (receiving a Map message and sending a Reduce message). We call this behaviour the client role and now expect a varying number of participants to inhabit it. On the other hand, the server role is as usual instantiated by exactly one participant and the session does not start without its presence. The follow-

1 Since the previous multiparty session types [4, 19] do not support explicit parallelism, we rely on asynchrony to express the desired behaviour.
ing picture illustrates this dynamic protocol. Its global type features the new universal type.

\[ G = \mu x \forall x: \text{client} \cdot \{ \text{server} \rightarrow x(Map); x \rightarrow \text{server}(\text{Reduce}) \}; x \]

The repeated interaction in the global type \( G \) involves a Map message to be sent by the server to every participant \( x \) of the client role; the server then expects a message \( \text{Reduce} \) in answer.

At the type level, such an operation is specified using a universal quantification:

\[ \forall x: r.G' \] polls the current participants \( p_1, \ldots, p_r \) of role \( r \) and, in parallel processes, binds \( x \) to each in the subsequent interaction, as in \( G' \{ p_1/x \} \cdots G' \{ p_r/x \} \)

In our example, \( G' = \text{server} \rightarrow x(\text{Map}); x \rightarrow \text{server}(\text{Reduce}) \) is executed in parallel for each client \( x \). Then, the recursion variable \( x \) points the interaction back to its beginning.

**Local types** Since the implementation, written here in a variant of the \( \pi \)-calculus, is distributed, the typing system first projects the global type to each end-point (local) type. For each role, the projection algorithm computes a local type that describes the behaviour of any participant that wants to play it. The local types for this session are the following:

\[ T_{\text{client}} = \mu x. \forall y. (\text{server}, \text{Map}) : x! (\text{server}, \text{Reduce}) : x \]
\[ T_{\text{server}} = \mu x. \forall y. (\text{server}, \text{Reduce}) : x \]

First, the client behaviour \( T_{\text{client}} \) is straightforward as it is only involved in two messages at each iteration with the server. The local type of the client expresses that it expects a message \( \text{Map} \) from the server (\( \forall y.(\text{server}, \text{Map}) \)) and that it sends a message \( \text{Reduce} \) as an answer (\( \forall y.(\text{server}, \text{Reduce}) \)). The server role is involved in all the messages of this session. We note the presence of the quantification over all \( x \) playing the client role.

**Processes** We write some process examples that would be well typed against the local types. The session identifier \( s \) denotes an active session:

\[ P_{\text{client}}(s) = a[z: \text{client}(s)]. \mu X. \forall y. (\text{server}, \text{Map}) : x! (\text{server}, \text{Reduce}) ; X \]
\[ P_{\text{server}}(s) = a[z: \text{client}(s)]. \mu X. \forall y. (\text{server}, \text{Reduce}) : x \]

A session starts through the join operation \( a[z: \text{client}(s)] \) which gets the session name \( s \) of a running session advertised on \( a \). A participant \( z \) playing the client with \( P_{\text{client}}(s) \) is simply exchanging messages \( \text{Map} \) and \( \text{Reduce} \) with the server through sending \( (s!) \) and receiving \( (s?) \) operations. The server needs to fork sub-processes for its interactions with each client. To this effect, the polling operation \( \forall y. (x: \text{client}), (s!: (\text{server}, \text{Map}), s?: (x, \text{Reduce}) \) creates as many processes \( s!: (\text{server}, \text{Map}), s?: (x, \text{Reduce}) \) as there are participants \( x \) playing the client role. Note that late joining client participants are incorporated in the session at each iteration: the repetition of the polling operation \( \forall y. (x: \text{client}) \) is able to ensure a safe interaction between all parties.

(2) **Peer-to-peer chat** In this session, there is only one role, the client, whose behaviour is to always broadcast its messages to all the other clients. We give the global type and a representation of the interaction when four clients are present.

\[ G = \mu x. (\forall x: \text{client}, y: \text{client}, u: \text{client}, v: \text{client}, w: \text{client} : x \rightarrow y\text{Msg}(\text{string}) \}; x \]

This type features a double quantification which specifies that each pair of clients \( x, y \) will interact in the form of a unique \text{Msg}. The explicit exclusion of \( x \) from the list of clients \( y \) prevents self-sent messages.

This second example shows the projection difficulties that arise from quantification.

**Local types** To illustrate the projection of nested quantifiers, we first rely on our intuition: each client should send a message \text{Msg} to every other client and, concurrently, should expect a message \text{Msg} from each of them.

\[ T_{\text{client}}(z) = \mu x. (\forall y: \text{client} \rightarrow \{ ! (x, \text{Msg}(\text{string})) \}); x \]
\[ T_{\text{server}}(z) = \mu x. (\forall y: \text{client} \rightarrow \{ ! (x, \text{Msg}(\text{string})) \}); x \]

Let us examine how the projection algorithm gives this local type. Suppose we project for a generic client \( z \). The first quantifier \( \forall x: \text{client} \) of the global type necessarily involves \( z \), meaning that among these parallel processes there is exactly one where \( z \) is \( x \). In the other parallel processes, although \( x \) is not \( z \), \( z \) can still be involved. The projection of the second nested quantifier \( \forall y: \text{client} \rightarrow \{ ! (x, \text{Msg}(\text{string})) \} \), which explicitly excludes the possible \( ! (z, \text{Msg}(\text{string})) \).

**Process** Once the local types are known, the client processes have a similar structure, including the explicit polling operator, written \( \forall y. (\forall z: \text{client} \rightarrow \{ ! (y, \text{Msg}(\text{string})) \} \)

\[ P_{\text{client}}(z) = a[z: \text{client}(s)]. \mu X. \forall y. (\forall z: \text{client} \rightarrow \{ ! (y, \text{Msg}(\text{string})) \} \]

(3) **Auction** We now illustrate the expressiveness of our universal types when combined with instantiation of participant identities and branching session. In this session, we have three roles: the multiple buyers (here participants \text{alice}, \text{alex}, \text{alan}) and sellers (here \text{bob}, \text{ben}) which all connect to a single broker. This broker will then form matching pairs \( (x,y) \) of buyers and sellers who will then continue their interaction \text{Price-Order} separately.

\[ G = \forall x: \text{buyer}, \forall y: \text{seller}, \{ \text{match}(y,x) \rightarrow \{ \text{notify}(y,x) \rightarrow \{ \text{price}(y,x) \rightarrow \{ \text{order}(y,x) \rightarrow \{ \text{stop}(x) \rightarrow \{ \end{array} \]

The quantifications \( \forall x: \text{buyer}, \forall y: \text{seller} \) specify that every possible association between buyers and sellers is considered by the broker when he makes his choices. For each pair \( (x,y) \) of buyer and seller, the broker selects to send to \( x \) either a message \text{match}(y) if he has found \( y \) to be a match for \( x \), or a message \text{quit}(y) otherwise. If the message \text{match} was sent, \( x \) notifies \( y \) and the interaction \text{Price-Order} proceeds. In the other branch, \( x \) needs to warn \( y \) by the message \text{stop} that the broker chose the second branch.

For this example, we just write a process for a buyer:

\[ P_{\text{buyer}}(s) = a[z: \text{buyer}(s)]. \forall y: \text{seller}. \{ \text{match}(y,x) \rightarrow \{ \text{notify}(y,x) \rightarrow \{ \text{price}(y,x) \rightarrow \{ \text{order}(y,x) \rightarrow \{ \text{stop}(x) \rightarrow \{ \end{array} \]

From the above process, we can see the importance of the communication of the participant identity \( y \) with the messages \text{match} and \text{Quit}. The adjunction of \( y \) to the messages is necessary for \( x \) to know to which \( y \) to send the \text{Notify} message. Note that the \( y \) in \text{match}(y) is not a regular payload as all the sellers \( y \) are already known by \( x \): at reception, \( x \) matches his known \( y \) against the one coming along \text{Match} or \text{Quit}.

This example presents a non-recursive session where all participants leave the session (through the expression \text{quit}(s)) at the end of their interaction. Since late joiners always start at the beginning of the session, they cannot safely interact with the participants that have already proceeded. To guarantee progress, we require that late

---

2 If we want our global type to include those self-sent message, it can be done explicitly by writing a global type: \( \mu x. (\forall x: \text{client} : x \rightarrow \{ y: \text{Msg}(\text{string}) \}; x \)
sessions wait for the current participants to end before joining themselves and beginning their actions. To provide consistent synchronisation, we introduce a multiparty locking mechanism to protect the global session executions.

Main contributions

2. Multirole session calculus

We describe here an extension of the multiparty session calculus (3) with a new role-based multiparty session type framework where participants can play several roles in a session. Its semantics allows participants to dynamically join and leave a running session, and create new parallel sessions.

3. Multirole session calculus

Introduction of a universal type for polling participants, along with explicit parallel compositions, and a type system that provides subject reduction (Theorem 4.2) and type safety (Corollary 4.3: no type error for values and labels). The endpoint projection and the well-formedness conditions of global types deal with the subtle interplay between universal quantifiers, parallel compositions, branching and instantiations of participant identities.

4. Semantics and type system with a simple locking mechanism by which communication safety (Theorem 5.4: every receiver has a corresponding sender with the right type), progress (Theorem 5.6: processes in a single multiparty session always progress) and join progress (Theorem 5.8: late joiners can always join to an existing session and progress) are established.

5. Practical implementation techniques used in our prototype implementation are available from [3].

The proofs, detailed definitions, additional examples and the prototype implementation are available from [3].

2. Multirole session calculus

We describe here an extension of the multiparty session calculus presented in [4]. Our new system handles roles and allows programs to participate in protocols that include multiple parallel interactions and dynamic role instantiation.

\[
\begin{align*}
\usec &::= x \mid a \mid b \mid \ldots & \text{Shared channel} \\
p &::= p: r \mid x: r & \text{Participant role} \\
\tilde{p} &::= p: \tilde{p} \mid x: \tilde{p} \mid \epsilon & \text{Participant list} \\
c &::= s(p) \mid y & \text{Session channel} \\
e &::= v \mid x \mid e \land e \mid \ldots & \text{Expression} \\
v &::= a \mid s[p: r] \mid \text{true} \mid \ldots & \text{Values} \\
P &::= & \text{Processes} \\
&= a(G) & \text{Session initialisation} \\
&= a[p](y).P \mid \text{Join} \\
&= c!(p.\{\tilde{p}(x)\}r_1,\ldots,p_r) & \text{Receive} \\
&= c!(x:r \setminus \tilde{p}).\{P\} & \text{Poll} \\
&= P \mid P \mid P, P & \text{Parallel, Sequential} \\
&= e & \text{Conditional} \\
&= \text{null} & \text{Recursion, Variable} \\
&= (v \cdot a : G)P & \text{Null, Restriction} \\
&= (v \cdot s)P \mid s:h & \text{Session restriction, Buffer} \\
&= a(s)[R] & \text{Registry} \\
R &::= r_1, p_1, \ldots, p_n & \text{Role set} \\
h &::= \epsilon \mid h.(p_0: r_0, p_1: r_1, l(\tilde{p})(v)) & \text{Buffer}
\end{align*}
\]

Figure 1. Multiparty session calculus

Syntax

We give in Figure 1 the syntax of the processes of our session variant of the π-calculus. A session is always initialised by a process of the form \(a(G)\) where \(G\) is a global type (formally defined in § 3). Session initialisation attributes a particular global interaction pattern \(G\) to a shared channel \(\epsilon\). Once the session has been initialised on channel \(\epsilon\), participants can join with \(a[p](y).P\) where \(p\) designates a participant identity \(p\) or \(x\) associated with a particular role name \(r\). Joining binds the variable \(y\) with the session channel that this particular participant can use when he plays the role \(r\). Leaving the session is done by \(\text{quit}(e)\), where \(e\) is the session channel corresponding to the participant and role.

The asynchronous emission \(c!(p.\{\tilde{p}(x)\}r_1,\ldots,p_r)\) allows to send to \(p\) a value \(e\) labelled by a constant \(l\) and participant names \(\tilde{p}\). The reception \(c?!(p.\{\tilde{p}(x)\}r_1,\ldots,p_r)\) expects from \(p\) a message with a label among the \(\{l\}_1,\ldots,l_r\) with participants \(\tilde{p}\). The message payload is then received in variable \(x_i\), which binds in \(P_i\). Messages are always labelled. The list of participants \(\tilde{p}\) enriches the label \(l_i\) in order for the receiver to be able to disambiguate messages that have the same sender and label, but different continuations.

The polling operation \(c?(x:r \setminus \tilde{p}).\{P\}\) is the main way to interact with the participants that instantiate a given role: \(P\) is replicated for each participant \(x\) playing role \(r\), with the exception of the participants mentioned in \(\tilde{p}\).

Parallel and sequential composition are standard, as are the conditional and recursion. The creation of a shared rendez-vous name is done by \((v \cdot a : G)P\). This fresh name can then be used as a reference for future instances of a session specified by \(G\).

Once a session is running, our semantics uses some artifacts that are not directly accessible to the programmer. First, session instances are represented by session restriction \((v \cdot s)P\). Second, the message buffer \(s:h\) stores the messages in transit for the session \(s\). Last, the session registry \(a(s)[R]\) records the current association between participants and roles in the running session \(s\).

For simplicity, we write \(c?!(p.\{\tilde{p}(x)\}r_1,\ldots,p_r)\) if there is a unique branch. Similarly, we omit the empty list of participant \((\epsilon)\) and unit payloads (e.g. \(c!(p.l)\)). We also do not write \(0\), and roles \(r\) (e.g. in \(x:r\)) if they are clear from the context.

We use syntactic sugar for the special roles that cannot be multiply instantiated. Polling is done implicitly for these roles. Their participants’ names (\(p\) or \(x\)) do not have to be explicitly mentioned: the mention of the role \(r\) is sufficient and unambiguous. In the Map/Reduce example from § 1, server is such a role.

We call a process which does not contain free variables and runtime syntax initial.

Semantics

Figure 2 lists the reduction rules. The \(\text{inst}\) rule proceeds to a session initialisation by reducing \(a(G)\). It creates a fresh session channel \(s\) and two processes. First, the session registry \(a(s)[R]\) is an entity that centralises the association between participants and roles in the particular instance \(s\) of a session. Initially, \(R\) does not record any participant for any of the roles of \(G\). The second process is the session’s message buffer \(s:h\), which is also initially empty.

The rule \(\text{join}\) governs the registration of a participant to a running session. The participant asks with \(a[p : r](s).P\) to join the session advertised on channel \(a\) and specifies his identity \(p\) and which role \(r\) he wants to play. This information is added to the session registry \(a(s)[R \cdot r : P \cup \{p\}]\) and the session channel \(s[p : r]\) is communicated. The rule \(\text{quit}\) manages the departure of a participant from a session: \(\text{quit}(a[p](y).P)\) forces the deletion of \(p : r\) from the registry.

The rule \(\text{send}\) describes asynchronous sending, which appends its labelled message to the buffer \(s:h\). In rule \(\text{recv}\), the reception takes from the session buffer the first message \((p' : r', p : r, l_1(p)(v))\) that has a proper address, label and participant list and selects the matching continuation \(P_k\). The rule \(\text{poll}\) details the reduction of the polling process \(s[p : r']r[x : r \setminus \tilde{p}].\{P\}\). The set
of participants \( \{p_1, \ldots, p_k\} \) that play role \( r \) (once the ones in the \( \overline{p} \) are removed) is received from the session registry and the process \( P \) is forked accordingly, with \( x \) appropriately substituted. In [PAR], bound names in \( e \) and free names in \( Q \) are disjoint.

The reduction is defined modulo the standard structural equivalence \( \equiv \). We just mention here the session garbage collection rule (\( v \in G, s\)) \((a(s) \mid x \mid e) \equiv 0 \) when \( \forall r_t \in G, R(r_t) = \emptyset \) and the permutation rule \( s : (q, p, l(p_1) \langle q \rangle) \cdot (q, p', l(p_2) \langle r \rangle) \cdot s : \equiv s : (q, p', l(p_2) \langle r \rangle) \cdot (q, p, l(p_1) \langle q \rangle) \cdot s : - h \) which allows to put forward in the session buffer the messages that have different senders, recipients, labels or participants lists. Others are standard.

**Reduction example** We take the process \( P_{\text{client}}(z) \) from the peer-to-peer chat mentioned in the introduction (§1.2), Figure 3 gives reduction steps of a situation where we have two client processes \( P_{\text{client}}(p_1) \) and \( P_{\text{client}}(p_2) \) that want to interact on session channel \( a \). We call \( Q(z) \) the process \( \mu X.(z)_x(\forall y:\text{client}\cdot z).\{z\}_x(y, Msg(m))\cdot [z]_x(\forall x:\text{client}\cdot z).\{z\}_x(x, Msg(w))\}; X \) and abbreviate the registry \( a(s)\cdot[\text{client}: \{p_1, p_2\}] \) by \( R \).

```latex
\begin{align*}
\begin{array}{ll}
\vdash (v a)(a(G)) & \rightarrow (v a)\{p_1(p_1)|p_2(p_2)\} \quad \text{[INIT]} \\
\vdash (v a)((v s)(a(s)\cdot[\text{client}: \emptyset]) \mid x \mid e) & \rightarrow (p_1(p_1)|p_2(p_2)) \quad \text{[JOIN]} \\
\vdash (v a, a(s)\cdot[\text{client}: \{p_1, p_2\}] \mid x \mid e) & \rightarrow (p_1(p_1)|p_2(p_2)) \quad \text{[JOIN]} \\
\vdash (v a, a(s)\cdot[\text{client}: \{p_1, p_2\}] \mid x \mid e) & \rightarrow (p_1(p_1)|p_2(p_2)) \quad \text{[JOIN]} \\
\vdash (v a, s)\cdot[\text{client}: \{p_1, p_2\}] \mid x \mid e) & \rightarrow (p_1(p_1)|p_2(p_2)) \quad \text{[JOIN]} \\
\vdash (v a, s)\cdot[\text{client}: \{p_1, p_2\}] \mid x \mid e) & \rightarrow (p_1(p_1)|p_2(p_2)) \quad \text{[JOIN]} \\
\vdash (v a, s)\cdot[\text{client}: \{p_1, p_2\}] \mid x \mid e) & \rightarrow (p_1(p_1)|p_2(p_2)) \quad \text{[JOIN]} \\
\end{array}
\end{align*}
```

**Figure 3.** Reduction for the peer-to-peer chat example

**3. Multirole session types**

In this section, we present the multirole session types which specify the communication patterns that are to be enforced. We start with the definition of global and local types and follow with projection and well-formedness properties.

### 3.1 Global and local types

**Global types** \( G \) describe role-based global scenarios between multiple participants as a type signature. When a participant agrees with a global type \( G \), his behaviour is defined by a local protocol (called local type \( T_i \)) that is generated by the projection of \( G \) to the role he wants to play. If each of the local programs \( P_1, \ldots, P_n \) can be type-checked against the corresponding local type \( T_1, \ldots, T_n \), then they are automatically guaranteed to interact properly, following the intended scenario. The grammar of global types \( (G, G', \ldots) \) and local types \( (T, T', \ldots) \) is given in figure 4. There are four key extensions from the standard multiparty session types [4]: (1) association of each participant to a role; (2) universal quantifiers to bind participants identities; (3) parallel compositions for local types; and (4) labels that can be extended by lists of participants.

In the global types \( (G, G', \ldots) \), a global interaction can be a labelled message exchange \( (p \rightarrow p' \langle l_i(p_i) | U_i \rangle_{G_i} \mid \{G\}) \) where \( p \) and \( p' \) denote the sending and receiving participants with roles (recall that \( p \) denotes either \( p : r \) or \( x : r \)). \( p_i \) is a list of participants, \( U_i \) is the payload type of the message and \( G_i \) the interaction that follows the choice of label \( l_i \) (\( l_i \) is a finite set of integers). Value types \( S \) include channel types \( G \) or base types (bool, unit, ...). Message types \( U \) are either value types \( S \) or local types \( T \) (which correspond to the behaviour of one of the session participants) for delegation.

Parallel composition is written as \( G \sqcup G' \). Universal quantification is written \( \forall x : r \ \overline{p}.G \) where the participants of role \( r \) bind...
free occurrences $x$ in $G$. It corresponds to the operational semantics of $s \triangleright p : r \triangleright \{G, \triangleright p : r \triangleright \{P\}$ (see § 2), i.e., a parallel composition $G[p_1/x] \ldots G[p_k/x]$ for some list of participants $\{p_1, \ldots, p_k\}$ playing the role $r$ (which is decided at runtime), from which the list of participants $p$ has been excluded.

In local types $T$, selection expression the transmission to $p$ to a label $l$ taken from a set $\{l_1, \ldots, l_n\}$ with a list of participants $\vec{p}$ and a message type $U_l$ followed by $T$. Branching is its dual counterpart.

The other local types are similar to their global versions.

We consider global and local types modulo the following equalities. For local types, we define: $(T | \triangleright e \cdot T) = (e; T)$, $(T | \triangleright e \cdot T; \triangleright e)$ and $(T | \triangleright e \cdot T; \triangleright e)$. Similar equalities are applied to global types. We also use similar abbreviations for local and global types as the ones for processes (mentioned in § 2). In particular, we write $p \rightarrow q \triangleright \{G\}$ or $\{p, l, \{G\}; T \}$ for a single branching, and $p \rightarrow q \triangleright \{G\}$ if the list of participant is empty and the payload type is unit. end is also often eluded.

**Example 3.1** (Global types). To give some additional clarity to the semantics of global session types, we give here several variations on an additional example. We imagine a chat protocol (similar in spirit to the peer-to-peer chat session) where the clients must interact through a single server. We have thus two roles: the unique server and the multiple clients. Each client’s behaviour is to send a message to the server who will then broadcast it to all the others. In the following picture, we only represent the Msg that one client sends to the server and that is followed by the server broadcasting its content (in message Spread) to all the other clients.

![Chat protocol diagram](chat-diagram.png)

The global type for this session relies on the sequentiality that links each Msg to its following Spread. We write it as:

$$G_1 = \mu x. \forall y : \text{client}. \{x \rightarrow \text{server}(\text{Msg}); y \rightarrow x. \text{server} \rightarrow y \text{(Spread)}\}; x$$

It starts with a quantification over all clients $x$. Upon reception by the server of a message from $x$, the global type specifies that Spread should be sent to all the other clients: $\forall y : \text{client}; x \rightarrow \text{server}(\text{server} \rightarrow y \text{(Spread)}).

An alternate chat server could be one where the server collects all incoming messages and then sends a digest to all clients. In that case, the global type would be written:

$$G_2 = \mu x. \forall y : \text{client}. \{x \rightarrow \text{server}(\text{Msg}); y \rightarrow x. \text{server} \rightarrow y \text{(Spread)}\}; x$$

The central synchronisation between the two quantified types is important in our model. The semantics is radically different if this synchronisation is removed.

$$G_3 = \mu x. \forall y : \text{client}. \{x \rightarrow \text{server}(\text{Msg}); y \rightarrow x. \text{Spread}\}; x$$

The global type $G_3$ means that, independently for each client, the server first collects a message Msg and then immediately sends back to the same client a message Spread.

### 3.2 Projection from multirole global types to local types

We now define the projection operation, which, for any participant $z$ playing a role $r$ in a session $G$, computes the local type it has to conform to. We say an end-point projection of $G$ onto $z : r$, written $G | z : r$, is the local type that the participant $z$ should respect to play the role $r$ in session $G$.

As mentioned in § 1, the main difficulty lies in the projection of the quantifiers. Let us first consider informally the global type $\forall x : r : G$. This global type has the same semantics as $G[p_1/x] \ldots G[p_k/x]$ for some $p_1, \ldots, p_k$ playing the role $r$. If we write the projection of $\forall x : r : G$ for a participant $p_k$ playing role $r$ (written as $\forall x : r : G \triangleright p_k : r$), we can single out the instance corresponding to $p_k$:

$$(G[p_1/x] \triangleright p_1 : r) \ldots (G[p_k/x] \triangleright p_k : r) = (G[p_1/x] \triangleright p_1 : r) \mid \forall x : r : G \triangleright p_k : r$$

Based on this intuition behind the projection of quantifiers, we give the projection definition in figure 5. Projection is role-based, i.e., for each role $r$ of a session $G$, a local type $T = G \triangleright p = p : r$ is computed with $p = z : r$. The case $p = r : p$ is defined by replacing $z$ by $p$.

$$p \rightarrow p \triangleright \{p, l, \{G\}; T \} = p \rightarrow p \triangleright \{p, l, \{G\}; T \}$$

The projection of communication leads to a case analysis: if the participant projected to (i.e., $p$) is the sender, then the projection is a selection sent to $p'$; if $p$ is the receiver then the projection is an input from $p'$; if participant $p$ is both sender and receiver then the projection is an output followed by an input; otherwise, the communication is not observed locally and is skipped. The operator $\triangleright$ then merges the different remote branches (this operation was introduced in [35, § 4]). Roughly speaking, it makes sure that the locally observable behaviours are either independent of the remotely chosen branch or can be properly identified through their labels. It is defined by $T \triangleright T = T$ and the following equality:

$$\forall z : r : G \triangleright z : r = \forall z : r : G \triangleright z : r$$

Note that the merging operation may not return a result if the session uses labels ambiguously. An example can be found in § 3.3. Finally, the most critical rules define the projection of a quantified global type $\forall x : r : \{G\}$ to $p$. The first rule applies only when the quantification acts on the same role $r$ as the projection, and when $p$ is not in the exclusion list $\vec{p}$. In that case, as explained above, the local type is the parallel composition of $G$ where $x : r$ is substituted by $p$, projected for $p$, and a quantification excluding $p$. The second rule sees the projection acting homomorphically through the quantification on a different role, or if $p$ is in $\vec{p}$. Other rules are homomorphic as well. We say that $G$ is projectable if $G$ can be projected (i.e. projection gives a result) for each of its roles.

**Example 3.2** (Projection). We give an example of projection for the peer-to-peer chat session from § 1, which features nested quantifiers. The local type $T(z : \text{client})$ is calculated in the following way ($p$ is $z : \text{client}$):

$$\mu x. \forall y : \text{client}. \{x \rightarrow \text{server}(\text{Msg}); y \rightarrow x. \text{server} \rightarrow y \text{(Spread)}\}; x$$

(1) Peer-to-peer chat example. We give an example of projection for the peer-to-peer chat session from § 1, which features nested quantifiers. The local type $T(z : \text{client})$ is calculated in the following way ($p$ is $z : \text{client}$):

$$\mu x. \forall y : \text{client}. \{x \rightarrow \text{server}(\text{Msg}); y \rightarrow x. \text{server} \rightarrow y \text{(Spread)}\}; x$$

(2) Chat-server from example 3.1. We give the projections for each of the three global types. The projection of $G_1$ for the server and client roles gives:

$$T_1(\text{server}) = \mu x. \forall y : \text{client}. \{x \rightarrow \text{client}(\text{Msg}); y \rightarrow x. \text{client} \rightarrow y \text{(Spread)}\}; x$$

$$T_1(\text{client}) = \mu x. \forall y : \text{server}(\text{Msg}); x \rightarrow x. \text{client} \rightarrow z \text{?(server, \text{Spread})}; x$$
Note that the sequentiality between $\text{Msg}$ and $\text{Spread}$ is rightly present in the server’s local type. The projection of $G_2$ results in:

$$T_2(z:\text{client}) = \mu x.\forall y.\exists t(x,\text{client, Msg}); x$$

$$T_2(z:\text{server}) = \mu x.\forall y.\exists t(x,\text{server, Msg}, z, (?y, \text{client, Spread})); x$$

We note that the server’s local type represents a behaviour which first collects all incoming messages and then sends a digest to all clients. On the other hand, the client behaviour is the same as in session $G_1$. The projection of $G_3$ is given as:

$$T_3(z:\text{client}) = \mu x.\forall y.\exists t(x,\text{client, Msg}, ! (y, \text{client, Spread})); x$$

$$T_3(z:\text{server}) = \mu x.\forall y.\exists t(x,\text{server, Msg}, ? (server, Spread)); x$$

In the above types, for each client, the server first collects a message $\text{Msg}$ and then immediately sends back a message $\text{Spread}$ to this client.

### 3.3 Well-formedness

For type-checking to work, global types need to follow a set of rules that will ensure a reliable and unambiguous session behaviour.

**Syntax correctness** We apply kinding rules [3] to construct syntactically correct types. A first point that is verified is that every participant variable $x$ is bound by a quantifier and that it is consistently used with the same role. Then, we check that recursion variables do not appear under quantification or explicit parallel composition. Formally, if a global type is of the form $\forall x.\exists y. G \parallel G'$, then $G$ and $G'$ are required not to contain any free recursion variables. This condition prevents any race condition between different iterations of the same loop. We give a few examples of correct and incorrect global session types.

- $\forall x.\exists y. G_1 = \mu x.\exists y.\forall z.\exists t(z, \text{server, \text{msg}}); x \parallel \text{server, \text{broker, Notify}}; x$
- $G_2 = \mu x.\exists y.\forall z.\exists t(z, \text{server, \text{msg}}); x \parallel \text{server, \text{broker, Notify}}; x$
- $G_3 = \mu x.\exists y.\forall z.\exists t(z, \text{server, \text{msg}}); x \parallel \text{server, \text{broker, Notify}}; y$

Other checks include the verification that the position of `end` is indeed correct. For example, $(G; \text{end}); (G'; \text{end})$ is not well-formed.

**Projectability** As seen in § 3.2, projection does not always return a local type, due to the verification made when branches are merged. The merging operation verifies that each branch is properly labelled and that no local process can be confused about which branch to follow. We thus conclude that any global session type $G$ should be projectable.

- $G_4 = \text{broker, buyer, seller, np, Pay, Price, Order, \text{Stop}}$
- $G_5 = \text{broker, buyer, seller, np, Pay, Price, Order, \text{Stop}}$

The seller in $G_4$ cannot distinguish the two $\text{Msg}$ sent by the buyer. In $G_5$, the seller knows which branch has been taken by the broker since the upper one is labelled by $\text{Price}$ and the lower one by $\text{Stop}$.

**Linearity** The concept of linearity is introduced in [19] but, in our case, we use a relaxed version to allow flexible parallel compositions (explicit or through quantification) and branching. It makes sure that messages are always labelled in a way that prevents communication mix-ups.

To verify the linearity of a global type $G$, we first need to transform the quantifiers into explicit parallel compositions. To this effect, we associate to each role $r$ of $G$ a (big enough) list of participant names $p_0, p_1, \ldots$. Then, we compute for each role $r$ the local type $T_r = G \parallel \rho$ and homomorphically replace every subterm of $T_r$ of the form $\forall x.\exists y. z$, by $T_0(p_1/x) \parallel T_0(p_2/x)$ with $p_1, p_2$ the first two participant names for role $r$ that do not appear in $\rho$. This transformation is called dequantification.

---

1. We leave the implicit quantifiers of the singly instantiated roles untouched.

**Definition 3.3 (Linearity).** We say that a well-labelled global type $G$ is *linear* if, for all roles $r$ of $G$, the dequantification $T'_r$ of $T_r = G \parallel p_0$ satisfies:

- $\forall p \in \{ p_1, p_2 \} (U_{i=1} \cup J)$ and $\forall p' \in \{ p_1, p_2 \} (U'_{j=1} \cup J)$ are both subterms of $T'_r$, then, $\forall i, j \in J, l_i = l'_j \Rightarrow (l'_{i=1} \cup J) \wedge l_i = l'_j \Rightarrow (p_i = p'_j \Rightarrow T_i = T'_j)$.

This definition checks that if two receptions exist in the local type of a role $r$, then either they share no label (and thus cannot be confused), or they share exactly the same set of message labels, with identical payload types, in which case they should only differ by the distinguishing lists of participants: these lists allow to reliably target different continuation types even when concurrent threads expect messages with the same labels.

$$G_6 = \forall x, y.\exists \text{broker, } y.\forall z.\exists t(z, \text{broker, \text{msg}}); x \parallel \text{broker, \text{msg}}; y$$

The dequantification of $G_6 \parallel p_0$ is $\exists \text{broker, } y.\forall z.\exists t(z, \text{broker, \text{msg}}); x \parallel \text{broker, \text{msg}}; y$(buyer are $p_0, p_1$), sellers are $q_0, q_1$):

- $\forall \text{broker, } y.\exists \text{msg} \exists q_0, q_1.\forall t(z, \text{broker, \text{msg}}); x \parallel \text{broker, \text{msg}}; y$(buyer are $p_0, p_1$), sellers are $q_0, q_1$):

These two concurrent threads have identical guards but different continuations. The dequantification of $G_7 \parallel p_0$ is $\exists \text{broker, } y.\forall z.\exists t(z, \text{broker, \text{msg}}); x \parallel \text{broker, \text{msg}}; y$(buyer are $p_0, p_1$), sellers are $q_0, q_1$):

In that case, the participant identity $(y)$ is added to the label $\text{Msg}$ and is able to disambiguate the concurrent receptions.

**Well-formedness** We now give the formal version of the well-formedness condition. Note that it is decidable.

**Definition 3.4 (Well-formed global types).** We say that a global type $G$ is well-formed if the following conditions hold:

1. **(Syntactically correct)** $G$ is syntactically correct [3].
2. **(Projectability)** $G \parallel r$ is defined for each role $r$ of $G$.
3. **(Linearity)** $G$ is linear (Definition 3.3).

**Example 3.5 (Well-formedness).** We test the well-formedness of the auction example from § 1 (the numbers below correspond to the well-formedness conditions). Recall the global type $G$:

$$G = \forall x.\exists y.\text{buyer, broker, np, Pay, Price, Order, \text{Stop}}$$

The syntax correctness (1) is checked easily: there is no recursion, participant variables are bound and used for a unique role, and end is well-positioned. $G$ is projectable (2) since the two branches (Match, Quit) do not forget to use different labels (Notify, Stop) to propagate to the seller $y$ the choice that the broker makes. Concerning linearity (3), the potential problem is in the first message: when a buyer $x$ receives a message Match or Quit from the broker, $x$ should know which parallel instance it concerns among the ones the quantification $\forall y$; seller creates. We only give below the verification details for the buyer. With buyers $p_0, p_1$ and sellers $q_0, q_1$, the result of the dequantification of $G \parallel p_0$; buyer is:

$$\exists \text{broker, } y.\forall z.\exists \text{msg} \exists q_0, q_1.\forall t(z, \text{broker, \text{msg}}); x \parallel \text{broker, \text{msg}}; y$$

We check linearity by looking at the different occurrences of the same label (for example Match in $G \parallel p_0$; buyer) being received from the same participant (e.g. broker): we verify that the lists of participant identities are different whenever the continuations are different. Linearity is thus only achieved here thanks to the communication of the disambiguating $y$ in messages Match and Quit, as it can be seen in the buyer’s case. The seller’s and broker’s (here omitted) linearity verifications are trivial.
4. Multirole session typing system

This section introduces the typing system and proves subject reduction (Theorem 4.2) and type safety (Corollary 4.3). There are three main differences with previous session systems. First, a participant \( x \) can appear free in environments, types and processes, and is necessarily bound by universal quantifiers. Second, previous systems did not allow any parallel composition of types which use common channels. Since the projection of a universal quantified type generates parallel compositions, we relax this restriction. Thanks to the well-formedness of the global types (Definition 3.4), the typing system for initial processes is kept simple. Third, our runtime typing system needs to track parallel behaviours by forks and joins.

4.1 Typing systems

Environments

We start with the grammar of environments.

\[
\Gamma ::= \emptyset \mid \Gamma, u : s \mid \Gamma, y : r \mid \Gamma, X : \Delta \\
\Delta ::= \emptyset \mid \Delta, c : T
\]

\( \Gamma \) is the standard environment which associates variables to sort types or roles, shared names to global types, and process variables to session types. \( \Delta \) is the session environment which associates channels to session types. We write \( \Gamma, u : s \) only if \( u \notin \text{dom}(\Gamma) \).

Similarly for other variables. We define the sequential \( ; \) and parallel \( \circ \) compositions for types as follows:

\[
\Delta \circ \Delta' = \Delta \circ \text{dom}(\Delta) \cup \Delta' \circ \text{dom}(\Delta) \\
\cup \{ c : \Delta(c) \mid c \in \text{dom}(\Delta) \cap \text{dom}(\Delta') \}
\]

where \( \emptyset \in \{ \_ \} \) and \( \Delta(c) \circ \Delta'(c) \) is syntactically well-formed.

Typing systems for initial processes

We detail the typing system for expressions and processes in figure 6. The judgement for expression typing is given as \( \Gamma \vdash e : S \). The judgement for process typing is given as \( \Gamma \vdash P : \Delta \) which can be read as: “under the environment \( \Gamma \), process \( P \) has session type \( \Delta \)”.

Rules \([\text{BOOLOp}10]\) are standard. \( \Gamma \vdash \text{Env} \) means that \( \Gamma \) is well-formed, and \( \Gamma \vdash S : T \) means \( S \) is well-formed under \( \Gamma \). Since a participant variable with role can appear both in types and environments, we need to use kinding techniques to make sure that types with free variables do not appear before the variables’ declarations and ensure well-formedness (see Definition 3.4). Rules \([\text{RL}, \text{RLV}]\) are introduction rules for participants associated with roles.

Rules \([\text{INIT}]\) type the initialisation of a session with a global type \( G \). The judgement \( \Gamma \vdash \Delta : \text{end} \) means that \( \Delta \) only contains end or \( \emptyset \) [18, 19]. The rule ensures the initialisation is not followed by the prefix. Rule \([\text{JOIN}]\) types a joining process that follows the projection to \( p \). A leaving process is typed if the remaining session type is completed (i.e. end).

Rule \([\text{Sele}]\) is for the selection of label \( l_i \), participants \( p_i \) and payload \( e \). We first infer the destination \( p \) from \( \Gamma \). If \( e \) is an atomic type (e.g. bool) or a shared channel type, then it is typed as in standard selection rules [4, 19] for the expression by recording participants \( p_i \) in the resulting type. This way, we can preserve the dependency between the participants during polling and session communications. Rule \([\text{SELS}]\) is a session delegation rule [4, 19]. Rule \([\text{DqA}]\) is the dual of the selection rules. Note that the participants \( p \), \( p_i \in \text{SEL}(p, \{ l_i(p_i(y_i), P_i) \}_{i \in I}) \) are free so that they are bound by the polling and dynamically instantiated by reductions. Rules \([\text{PAR,SEQ}]\) assume \( \Delta \circ \Delta' \) and \( \Delta, \Delta' \) are defined. Rule \([\text{POLLING}]\) is the introduction rule for the universal quantification. It only concerns a single session (otherwise other sessions are copied after forking).

The other rules are standard [4, 19].

Proposition 4.1. Assuming the bound names and variables in \( P \) are annotated (i.e. processes whose bound variables are annotated by types), type-checking of \( \Gamma \vdash P : \emptyset \) terminates.

Typing runtime processes

While the session typing systems for initial processes are simple, typing runtime (which keeps tracking intermediate invariants to prove the theorems) is not trivial due to parallel processes and participant instantiations generated by polling. We first extend the syntax of types \( T \) to include message selection type \( \Gamma \vdash (p : r, l, \{ \bar{p} \}) (T) \), which is an intermediate type for labelled values stored in the message buffer.

To type runtime processes, we need to extend judgements to \( \Gamma \vdash \Sigma P : \Delta \), which means that \( P \) contains the message buffers whose session names are in \( \Sigma \). We only show the most interesting typing rule for the register:

\[
\Gamma \vdash a : (G) \quad \{ p \}_{p \in \text{dom}(G)} G \vdash x : r = T_i \\
\Gamma \vdash a : (G)[R] \vdash \{ \{ p \}_{p \in \text{dom}(G)}(T_p / x_i) \}_{i \in I} \text{[RGST]} \\
\Gamma \vdash \Sigma P : \Delta (i = 1, 2) \\
\Gamma \vdash \Sigma, p : \Delta_1 : \Delta_2 \text{[GPAR]}
\]

\([\text{RGST}]\) assigns to the registry a type which holds a set of projected local types for all roles with participants which are not recorded in \( \Sigma \). Session typing \( s[r : p] : T_p / x_i \) is erased once it interacts with the initialisation process \( a[p : r] : \{ \} \) (see rule \([\text{GSEL}]\) in figure 2), and the resulting \( P \{ p[ r : p] / x_j \} \) holds \( s[r : p] : T_p / x_i \) (see the proof of Subject reduction theorem in [3]).

When two runtime processes are put in parallel (rule \([\text{GPAR}]\) a queue associated to the same session does not appear twice (\( \Sigma_1 \cap \Sigma_2 = \emptyset \)). For composing the two session environments, either (1) we sequence a message type \( T \) and a local type \( T' \) for the same session typing \( s[p : r] : T_p / x_i \) and \( s[p : r] : T' \) can be parallel composed as \( s[p : r] : (T \parallel T') \) by checking the linearity condition for \( T' \parallel T' \) following Definition 3.3; and otherwise (3) undefined. Then we define \( \Delta \circ \Delta' \) replacing \( \emptyset \) by \( \uparrow + \Delta' \) in the definition of \( \Delta \circ \Delta' \).

4.2 Subject reduction

As session participants join, interact and leave, runtime session types need to follow. This dynamism is formalised by a type reduction relation \( \Rightarrow \) on session environments as follows.

1. \( \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \} \Rightarrow \{ p[q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \} \)
2. \( \{ p[ q : r] : (q : r, l_i(p_i(U_i), T_{i,x}) \} \Rightarrow \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \}
3. \( \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \} \Rightarrow \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \}
4. \( \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \} \Rightarrow \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \}
5. \( \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \} \Rightarrow \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \}
6. \( \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \} \Rightarrow \{ p[ q : r] : (p[q : r], l_i(p_i(U_i), T_{i,x}) \}

In the above type reduction rules, message selection types are considered modulo the type equivalence relation \( \equiv \) and \( \Delta' \) is a type evaluation context (i.e. \( \Delta' = \emptyset \cup \{ I \} \parallel T \mid \Delta \cup \Delta' \mid \Delta' ; T \parallel T \mid \Delta' \cup \Delta' ; T \parallel T' \parallel T' \).

Rule (1) corresponds to the choice of label \( l \). Rule (2) corresponds to the exchange of a labelled value from participant \( p : r \) to participant \( q : r' \). Rule (3) is about self-sending and receiving. Rule (4) governs universal quantification and forks types with respect to the participants which are not in the exclusion list \( \bar{p} \). Rules (5,6) are congruent rules.

Hereafter we assume all processes are derived from the initial processes (§ 2) (i.e. subterms of those who are reduced from initials). Using the above definitions.

Theorem 4.2 (Subject reduction). Suppose \( \Gamma \vdash \Sigma P : \Delta \) and \( P \Rightarrow^* P' \). Then, \( \Gamma \vdash \Sigma P : \Delta \Rightarrow^* P' \).
subject reduction theorem and the well-formedness of global types (Definition 3.4), we can prove:

**Corollary 4.3 (Type safety).** Suppose \( \Gamma \vdash P \triangleright \Delta \). For any \( P' \) such that \( P \rightarrow P' \), \( P' \) has no type error.

5. Communication safety and progress

This section discusses the difficulties that a distributed session semantics creates when participants can dynamically join, leave and poll. We illustrate two limitations of the semantics and typing system presented so far and propose a solution based on multiparty locking that allows more flexibility for leaving a session and guarantees communication safety. We give two progress properties, one of which goes beyond existing achievements.

5.1 Limitations

**Leaving a session** While our operational semantics ( \([\text{QUIT}]\) in figure 2) allows a participant to leave a session at any time, the typing rule (\([\text{LEAVE}]\) in figure 6) only allows a participant to leave when its local type is \( \text{end} \).

Recall the peer-to-peer chat example from \( \S 1 \) (2), \( G \) is of the form \( \mu x.G_0 x.\text{end} \) with \( G_0 = \forall x.\text{client}.y y.\text{client} x \{x \leftarrow y\text{Msg} \{\text{string}\}\} \). The recursive type prevents any participant from ever leaving since a process will never reach type \( \text{end} \). We however remark that a client can play just one interaction round (i.e. \( G_0 \)) and leave safely before another session iteration occurs. If the starting and ending points of global types are known, some participants are able to leave a session safely while others stay.

**Communication safety and progress** In traditional multiparty sessions, the subject reduction theorem immediately brings communication safety and progress (in a single session) [19]. The reason is that standard multiparty session initiation ensures that all parties are eventually present (it waits for the expected fixed number of participants to join), while the typing system guarantees the safety of the communications when they start. This does not hold in our system due to the interplay between joining, leaving and polling.

We illustrate this point with the peer-to-peer chat example from \( \S 1 \). In that global type, every client is broadcasting \( \text{Msg} \) to all the others. Recall the client process \( P_{\text{client}}(x) \) from \( \S 1 \).

\[
\alpha[z;\text{client}](x).\mu X. (\alpha'[y;\text{client} \{z\}.\{x!\{y,\text{Msg} \{m\}\}] \mid \alpha'[(x;\text{client} \{z\}.\{x\}\text{Msg}(w)\}];X)
\]

At each iteration, every client does exactly two polling operations. Now suppose that a client does the first polling operation (to send \( \text{Msg} \)) before another client joins. It means that this new client will not receive the message it expects. More generally, the polls that correspond to the emissions need to always give the exact same result as the reception polls. This suggests that some mechanism to synchronise distributed polling processes is required to guarantee consistent polling results.

5.2 Multiparty locking for polling synchronisation

This subsection shows that a simple locking policy that can be automatically computed from the global type is able to ensure a safe synchronisation to allow flexible session departure and consistent polling results. The key point is to temporarily \( \text{lock} \) late participants from joining in the middle of a session execution in order to prevent any interference with polling. This is simply done by automatically surrounding global types by \( \text{locks} \): \( \left[\text{lock}\{G\}\right] \) means that the interactions specified by \( G \) are protected from late joiners and is called a locked global type. This condition is easily implementable using a standard two phase commitment protocol which minimises the necessary synchronisation between processes (figure 7) and is easily implementable in ML (\( \S 5.5 \)).

**Syntax** We first extend the syntax of processes (figure 1) as:

\[
P ::= \ldots \mid \text{c}\left[\text{lock}\{\forall x.\text{client}.y y.\text{client} x \{x \leftarrow y\text{Msg} \{\text{string}\}\}\};x\right]
\]

\[
\Lambda ::= \emptyset \mid \Lambda \cup \{p;r\}
\]

The process syntax is extended to locking and unlocking operations. The registry has two new states: \( \alpha^R[\Lambda,\Lambda] \) represents a registry that is in the process of being locked (so far by participants \( \Lambda \)), while \( \alpha^R[\Lambda,\Lambda] \) represents a registry that is locked (and where participants \( \Lambda \) are still involved).

**Semantics** The operational semantics with multiparty locking is given in figure 7. It defines the relations between the three states of the registry and is based on a standard two phase locking protocol commonly found in distributed applications.

The first phase is the \textit{registration state}: if the registry is of the form \( \alpha^R[\Lambda] \), participants can join and leave the session through \([\text{Join}]\) and \([\text{QUIT}]\). The only other reduction rule that can be applied is \( \text{lock} \), which puts the registry in its second state, the \textit{locking state} \( \alpha^R[\Lambda,\ell;\Lambda] \). Then, the session can only wait for all the current participants in \( \Lambda \) to activate their locks by the rules \([\text{Up}\} \) and \([\text{Top}\}\). A new process can asynchronously join by \([\text{Join2}]\), and a current process can finally decide to leave \([\text{QUIT2}]\) from the active session.

The difference between \([\text{Up}\} \) and \([\text{Top}\}\) lies in the side condition:
Other rules are from 2.

\[ a(G) \rightarrow (v s)(a(s) R \mid s : e) \quad (\forall r \in G, R(r) = \emptyset) \quad [\text{INT}] \]

\[ a[p;r](y).P \mid a(s) R \mid r : P \rightarrow P[s[p;r]/y] \mid a(s) R \mid r : P \uplus [p] \quad [\text{JOIN}] \]

\[ a[p;r](y).P \mid a^*(s) R \mid r : P.A \rightarrow P[s[p;r]/y] \mid a^*(s) R \mid r : P \uplus [p], A \quad (p : r \not\in \Lambda) \quad [\text{JOIN2}] \]

\[ \text{quit}(s[p;r]) \mid a(s) R \mid r : P \rightarrow a(s) R \mid r : P \setminus [p] \quad [\text{QUIT}] \]

\[ \text{quit}(s[p;r]) \mid a^*(s) R \mid r : P.A \rightarrow a^*(s) R \mid r : P \setminus [p], A \quad (p : r \not\in \Lambda) \quad [\text{QUIT2}] \]

\[ s[p;r] \mid \text{lock} \mid a(s) R \rightarrow a^*(s) R \mid [p;r] \quad [\text{LOCK}] \]

\[ s[p;r] \mid \text{unlock} \mid a^*(s) R \mid r : P \rightarrow a^*(s) R \mid [p;r] \quad (\Lambda \neq \emptyset) \quad [\text{DOWN}] \]

\[ s[p;r]\{\langle p', r', l(P) \rangle | a^*(s) R, \Lambda | s : h - a^*(s) R, \Lambda | s : h - (p, r, p' : r', l(P)(v)) \} \mid a^*(s) R | (p \in R(r) \land P \in R(r')) \quad (p \in R(r) \land P \in R(r')) \quad [\text{SEND}] \]

\[ s[p;r]\{\langle p', r', [l_1(P_1)]_i \rangle | a^*(s) R | s : (p', r', p : r, l_2(P_2)(v')) : h \rightarrow P_1[v'/v_1] \mid a^*(s) R | s : h \} \quad (p \in R(r) \land k \in I) \quad [\text{RECV}] \]

\[ s[p;r] \uparrow (x : r \setminus \beta).\{ P \} \mid a^*(s) R \mid r : P.A \rightarrow P[p_{1}/x] \mid ... \mid P[p_{k}/x] \mid a^*(s) R \mid r : P.A \quad (R(r) \setminus \beta = \{ p_1, \ldots, p_k \} \uplus \{ P \in R(r') \}) \quad [\text{POLL}] \]

Other rules are from 2.

**Figure 7.** Operational semantics with multiparty lock

R \not\equiv \Lambda holds when \( \forall p : r : (\Lambda = \Lambda \uplus [p : r] \Rightarrow R = R[r : p \uplus [p]]) \). Consequently, \([\text{TOP}]\) is only triggered when the set \( \Lambda \) contains the exact same combinations of participants and role as the set \( R \), meaning that all participants have activated their locks.

The application of rule \([\text{TOP}]\) marks the beginning of the interaction state, with a registry of the form \( a^*(s) [\text{R}, \text{\Lambda}] \). Only in this state can the rules \([\text{SEND}, \text{RECV}, \text{POLL}]\) be safely applied. The registry goes back to its registration state by the application of rule \([\text{DOWN}]\) which can occur only when everyone besides one participant has activated the unlock operation by rule \([\text{DOWN}]\).

**Types and typing**

The syntax of global and local types are extended from section 4 as follows:

\[ G ::= ... \mid \text{lock} [G] \quad T ::= ... \mid \text{lock} \mid \text{unlock} \]

We say that a global type \( G \) is **terminable** if there exists at least one finite path (whose leaf is \( e \)) up to the unfolding of \( G \). A terminable type can be easily defined by a kinding system: \( E \) is terminable, \( p \vdash p' \{ l_1[p_1], U, G_i \}_{i \in I} \) is terminable if for some \( k \in I \), \( G_0 \) is terminable; and others are defined homomorphically (see [3]). For example, \( \mu x.p \vdash p' \{ l_1 U_1, x, l_2 U_2 \}.x \) is terminable, but \( \mu x.p \vdash p' \{ l_1 U_1, x, l_2 U_2 \}.x \) is not. We define the condition for global types and environments.

**Definition 5.1** (Well-locked and persistently well-locked).

We say that a global type \( G \) is **well-locked** if \( G \) is closed (i.e. no free participant and recursive type variables) and of the form \( \text{lock} [G_0] \mid \text{end} \), and \( G_0 \) does not include any \( \text{lock} \). We say that a closed global type \( G \) is **persistently well-locked** if \( G \) is of the form \( \mu x.\text{lock} [G_0] x \mid \text{end} \), with \( \text{lock} [G_0] x \) well-locked and \( G_0 \) is terminable. We call \( G \) **well-locked** if for all \( U(x) = (G) \), \( G \) is either well-locked or persistently well-locked. We call \( G \) **persistently well-locked** if for all \( U(x) = (G) \), \( G \) is persistently well-locked.

Type \( \text{lock} [G_0] \) means that a single multiparty session is locked. Type \( \mu x.\text{lock} [G_0] x \) states a multiparty session is **persistently** (repeatedly) locked. The persistent lock ensures if a new participant \( p \) wants to join, it can join at the beginning of the interaction \( G_0 \), and if one wishes to quit, it can quit at the end of the session. Consequently, it requires the global type to be of the form \( \mu x.\text{lock} [G_0] x \) with a well-locked \( G_0 \) that does not contain any infinite loop which would prevent from reaching a new iteration (unlock). The persistent condition is needed for the final strong join progress discussed later.
The property can be generalised to \( a : (G) \vdash P \triangleright \Delta \) if \( P \) is typable and with a type derivation where the session typing in the premise and the conclusion of each prefix rule is restricted to be at most a singleton (more precisely, \( \Delta = \varnothing \) in \([\text{JOIN}, \text{LEAVE}, \text{SERVER}, \text{BAR}]\) and \( \Delta \) contains at most one element in \( \Delta \neq \varnothing \) in \([\text{SEQ}], \Delta \neq \Delta' \) in \([\text{PAR}, \text{SEQ}], \Delta \in \{ \text{IF}, \text{NEW}, \text{REC}, \text{VAR} \} \) in figure 6, deleting \([\text{SLS}]\). We say \( Q = a[p](y).Q' \) is a single-session join if \( a : (G) \vdash P \triangleright \varnothing \) and \( Q' \) does not contain shared name restriction and any join process.

\( \Gamma \vdash^* P \triangleright \Delta \) ensures that \( P \) contains (several) join processes each of which holds a single session, while single-session join \( a[p](y).Q' \) has only one active point \( a \), and once the session initiated at \( a \), \( Q' \) can only perform session communication at that initiated session.

We prove the communication safety in a single multi-party session.

**Theorem 5.4 (Communication safety).** Suppose \( a : (G) \vdash P \triangleright \varnothing \) and \( P \) is initial. Assume \( a : (G) \) is well-locked\(^4\), and \( P \) does not contain any shared name restriction. For any \( P' \) such that \( P \vdash^* P' \), \( P' \) is communication safe.

The proof starts by a definition of coherent environments (a certain kind of duality relation over multiple participants \([4, \S 3]\)). Then, we prove a stronger subject reduction theorem that shows the reduction of well-locked processes preserves the coherency of the resulting environment. We note that session fidelity \([19, \Corollary 5.6]\) comes also as a corollary.

Now we prove the progress property in a single multi-party session as in \([19, \Theorem 5.12]\), i.e. if a program \( P \) starts from one session, the reductions at session channels do not get stuck.

**Definition 5.5 (Progress property).** We say \( \Gamma \vdash^* P \triangleright \varnothing \) can progress, or satisfies the progress property, if, whenever \( P \vdash^* P' \), then either \( P' \equiv 0 \), \( P' \triangleright R \) or for some single-session join \( a : (G) \vdash Q \) with \( a : (G) \in \Gamma \) such that \( P' \triangleright Q \triangleright R \) and \( R \) can progress.

The above definition means that a process satisfies the progress property if it can never reach a deadlock state, i.e., if it never reduces to a process which contains active sessions (this amounts to containing waiting process at some session channel) and which is irreducible in any inactive context with single-session join \( Q \) running in parallel.

**Theorem 5.6 (Progress).** Suppose \( \Gamma \vdash^* P \triangleright \varnothing \) and \( P \) is initial. Assume \( \Gamma \) is well-locked and \( P \) does not contain any shared name restriction. Then \( P \) can progress.

### 5.4 Join progress

The above standard progress property is not strong enough, since all late joiners cannot participate to existing sessions. This subsection states a new progress property, not found in the literature.

Recall the (1) map-reduce example from \( \S 1 \), and the position of the recursion in the global type to:

\[ G_0 = \forall x \cdot \text{client-server} \rightarrow x(	ext{Map}; \mu x \cdot x \rightarrow 	ext{server}(\text{Reduce}); x) \]

From \( G_0 \), we have the following well-formed processes:

\[ \begin{align*}
P_0(s, z; \text{client}) &= s!(\text{server, Map}; \mu x.s! x)(\text{server, Reduce}); x \\
P_0(s, z; \text{server}) &= x!(s; \text{client}, x!(s, \text{Map}; \mu x.s! x, \text{Reduce}); x) 
\end{align*} \]

While the interaction between them is communication safe, the problem is that a late client will never be listened to by the \textit{existing server} because the server’s polling operation is not repeated to include him. In other words, the late client cannot join an existing, already running session. Persistent locking ensures this situation does not happen.

\(^4\)The property can be generalised to \( \Gamma \) from \( \{ a : (G) \} \) if we compose a parallel composition of single-session processes to \( E[G] \) in Definition 5.2 as Definition 5.5. A similar generalisation is possible for Definition 5.7.

\[ \begin{align*}
P &\rightarrow^* Q \quad \text{if} \quad P \rightarrow Q \quad \text{and} \quad P \rightarrow Q \quad \text{is derived using} \quad [\text{QUIT}, \text{QUIT2}, \text{SEND}, \text{RECV}] \quad \text{or} \quad [\text{POLL}] \quad \text{at} \quad s[p]; r \\
\quad \text{with} \quad [\text{PAR,CTX,CONG}], \text{i.e.} \quad P \quad \text{interacts with a queue or registry through} \quad s[p]; r. 
\end{align*} \]

**Definition 5.7 (Join progress property).** We say that \( a : (G) \vdash P \triangleright \varnothing \) satisfies the \textit{join progress property} if:

- \( P \) can progress; and
- if \( P \rightarrow^* (v \ s)[P | a(s)[R]] \) then, for any single-session join \( a : (G) \vdash a[p]; r(y).Q \triangleright \varnothing \) with \( P | P' \) and, for any \( R \) such that \( P' | a(s)[R] | a[p]; r(y).Q \rightarrow^* a(s)[R'] \), \( R = Q' \),
- if \( s[p]; r \in R \), then there exists \( Q' \rightarrow^* a[p]; r \); and
- \( (v \ s)Q' \) satisfies the join progress property.

The above definitions says that a fresh joiner \( a[p]; r(y).Q \) cannot always join the existing (unlocked) session \( s \) in \( P' \). In addition, it can always progress at the created session channel \( s \) by interacting with \( P' \). More intuitively, once some participants under any role start a session, the late joiner can still join that session and interact with earlier joiners, progressing further. Note that we can consider any single-session join \( a[p]; r(y).Q \) to make a progress process, which contrasts with the definition of the progress property (Definition 5.5) where \( P \) is only composed of single-session joining processes.

**Theorem 5.8 (Join progress).** Suppose \( a : (G) \vdash P \triangleright \varnothing \) and \( P \) is initial. Assume \( a : (G) \) is persistently well-locked and \( P \) does not contain any shared name restriction. Then \( P \) satisfies the join progress property.

We have for our examples:

**Proposition 5.9 (Properties of the examples).** Assume that each global type \( G \) in the protocols (1–3) of \( \S 1 \) is replaced by \( \text{lock}(G) \). Then all examples are type/communication safe and can always progress. Moreover if each global type in the protocols (1,2) of \( \S 1 \) inside the recursive type, i.e. \( \mu x.G \) is replaced by \( \mu x. \text{lock}(G) \), they then additionally satisfy the join progress property.

### 5.5 Implementation

**Prototype implementation** The multirule calculus has been implemented as an extension of ML. Following the technique used in \([5, 13]\), the global types that the programmer writes are compiled into an end-point function for each role. This choice allows to replace the implementation of the typing system by an automated generation of well-typed processes that can be used, through an API, by the programmer. The session semantics is thus entirely generated and implemented by communication libraries.

**A distributed implementation** The main issue for our compiler is to distribute as much as possible the centralised aspects of the semantics of figures 2 and 7. First concerned, the message buffers are completely distributed and implemented on the sender side: a thread is spawned to asynchronously make sure that the message gets across the TCP channel. Second, the registries can be partially distributed with one registry per role that deals with the corresponding joining, leaving and polling activities. These distributed registries however need to stay in contact to synchronise the global locking events (rules \([\text{LOCK, TOP, UNLOCK}]\)). Registries are attributed to participants by age: the first joiner for a role plays the registry as well, until he quits, in which case the registry is transmitted to the second older.

**Extension** Singly instantiated roles, like the server or broker from examples in \( \S 1 \), are modelled through an inefficient implicit quantification. Our implementation gives a special status to these roles. We use the fact that they play their own registry. As a consequence,
no separate polling is necessary to send them messages and the extra messages required by the quantification can be avoided.

**Efficiency** To gain performances, we propose an implementation with optimised messaging and improved asynchrony. First, since the slowest operation is communication, our implementation tries to minimise the number of messages that are exchanged. The main illustration is that if two messages are specified to be sent in a row between the same participants, they are automatically concatenated.

A more radical change is to do the polling only once for every participant, at the beginning of each locked part of the session execution. The advantages are to limit the number of sent messages and to remove in effect the global synchronisation point of rule [\texttt{UNLOCK}]. As soon as all polling operations are done, the distributed session execution can safely proceed until the list of participants of the next iteration is synchronised (rule [\texttt{Top}]).

6. Related work

The first motivation for the present work is a strong need to extend session type theory with dynamic reconfiguration of multiparty sessions and role-based abstraction to support a wider range of communication protocols found in practice.

The inspiration for multiparty session types comes from the design of high-level global protocol signatures for Web Services Choreography Description Language [1]. In CDL, types of participants (participantType) are declared as instances of types of roles (roleType) which represent collections of interaction behaviours. Later, some of the members of the W3C CDL working group have started developing a language called Scribble [17, 28] based on the theory of multiparty session types [4, 19]. Scribble is currently being experimented with for several different application domains in distributed systems [2, 24, 27] including business and financial protocols [32]. Our auction example in § 1(3) was extracted from the Scribble specification document.

The need for roles in session programming is also substantiated by our experiences in implementing web service usecases [1] and parallel algorithms for high-performance clusters [25] using Session Java (SJ) [20, 21]. In this work, we first describe communications between processes in a global topology (e.g. a multidimensional mesh or a ring) in the form of parameterised multiparty session types [35]. The compatibility between [35] and our present work is yet to be investigated as complex topologies with dynamic features need sophisticated distributed synchronisation algorithms (see also § 7).

The second motivation for the present work is the incorporation of dynamic features most suited to and compatible with existing multiparty session types [4–6, 9, 14, 19, 22, 29, 35]. The Conversation Calculus [8, 33] models distributed behaviours among “places” using new primitives such as conversation contexts (i.e. shared interaction points) and up (\texttt{U}) communication (similar to [7, 12]). A conversation models the interactions between a client and various services, with dynamic joining into a conversation, for a possibly unknown number of processes. While both their work and ours aim to support dynamic natures for sessions, the two join mechanisms are quite different. Their join is encoded by base primitives for late joining into a point of conversation, which more closely resembles the late asynchronous session initiation in [20, 35]. On the other hand, our join mechanism is role-based, and articulated at the level of global types, by declaring a single type construct which binds participants to a role. In contrast to [8], the process which controls joining might be a sender or listener, depending on the result of the projection (i.e. the position of polling). This flexibility enables direct modelling and clear articulation (i.e. without encoding) of different patterns of dynamic parallel protocols including symmetric peer-to-peer chats (§ 1 and Examples 3.1 and 3.2) by types. In [8], they proposed a sophisticated typing system that builds a well-founded order on events (similar to the line of [34]), to guarantee progress for processes under the assumption that all communications are matched with sufficient joiners. They do not, however, explore type inference for progress (decidability of a generation of well-formed ordering) [33]. Our progress can be, on the other hand, guaranteed by well-formedness of global types, with an automatic insertion of locks (which means a typing system with progress is decidable with Proposition 4.1). This leads to a simple but practical prototype implementation as discussed in § 5.5. A strong joining property has not been studied in [8].

Contracts [11] record abstract interaction behaviours of processes, and typable processes themselves may not always satisfy the properties of session types such as progress: it is proved later by checking whether a whole contract conforms to a certain form. Proving properties with contracts requires an exploration of all possible interleaved or non-deterministic paths of a protocol, see [35, § 5].

The first suggestion to use roles to model dynamic conversations in the context of session types was made in [16]. This idea is further developed in a master’s thesis [26] which formalises a Java-like core calculus for role-based session interactions. A session structure is described as a collection of binary session types for broadcast channels (used to send messages to role participants). New participants can only join a conversation before it starts. Type structures for global protocols and their induced properties (in particular progress) are not studied in [26].

For further comparisons of session types with other service-oriented calculi and behaviour typing systems, see [15] for a wide ranging survey of the related literature.

7. Conclusion and future work

This work introduced a multirole session type discipline for validating dynamic behaviours among an unspecified number of participants, answering a well-known open problem of multiparty session types [4, 6, 9, 14, 19, 22, 29, 35]. Dynamism is formalised through a powerful universal type construct which can represent many collective communications protocols, ranging over parallel computations, P2P networking, chat protocols and e-commerce auctions. The greater expressiveness, projection and type checking are decidable. Global types offer a practical guideline for a correct multiparty synchronisation mechanism, by which the theorems (properties) are articulated as: (1) \( \forall x . G \) (subject reduction and type safety with dynamic join and leave semantics), (2) well-locked \( \text{Lock}(G) \) (communication safety and progress); and (3) persistently well-locked \( \mu x . \text{Lock}(G) ; x \) (join progress). Our prototype implementation demonstrates the direct applicability of the present theory.

To realise the full potential of the multirole session type theory, several challenges need to be addressed. First, the theory can be integrated with the multiparty session exceptions developed in [10] in order to handle system failure and fault-tolerance in a larger class of distributed protocols, preserving type safety. It is especially useful to directly express more complex and dynamic topologies, in combination with the parameterised type theory from [35].

One extension that comes immediately to mind is the addition of an explicit existential \( \exists x : r . G \). It however raises many semantic issues. Consider \( G' = \forall x . \text{client} . \exists y : \text{server} . x = y (\text{Msg}...) \). In that example, every client contacts a server (the intuition is that each x chooses its y). The question is: how can we ensure by local typing that servers will be listening to the right number of requests? The difficulty is that a server y can be potentially chosen by every client x or by none, and that this choice is distributed (and thus very hard to locally type check). Consequently, the global existential quantification rather abstracts complex distributed election al-
algorithm. A different solution is an extension to subtyping between roles $r_1 \ll r_2$ by which we can represent a protocol with memberships, e.g. a client sends a message to a subset of subscribers.

Second, type-based approaches for correct locking has been widely studied, including [30] in a framework of linear program analysis and types. Our aim in § 5.2 is to propose a simple way to realise synchronisation, articulated by global types, suggesting another use of global descriptions for different purposes. One such instance is studied in [14], where multiparty session types lead to an efficient buffer analysis, along with automatically guaranteed communication and buffer safety. A benefit of using global types (i.e. a choreography framework [1]) is that the analysis can be done solely based on global types, without directly analysing (possibly distributed) end-point types or processes since we can assume all processes agree with that global specification. An integration with global and local locking [30] is, however, an interesting future topic from the viewpoint of local refinements [22].

Third, we are currently collaborating with several industry partners working on open standardisations for financial protocols [32] and messaging middleware [2], governance architectures [27] and cyberinfrastructures [24] to attest the practical use and expressive power of local refinements for our next task.

Acknowledgement We are grateful to Luís Caires, Marco Carbone, Søren Debois, Kohei Honda, Raymond Hu, Kohei Suenaga, Hugo Vieira and Vasco Vasconcelos for their helpful comments. We thank the anonymous reviewers for their precise questions and suggestions. This work was supported by EPSRC EP/F003757/01 and GO15635/01.

References


