Global Progress in Dynamically Interleaved Multiparty Sessions*

Lorenzo Bettini[1], Mario Coppo[1], Loris D'Antoni[1], Marco De Luca[1], Mariangiola Dezani-Ciancaglini[1], and Nobuko Yoshida[2]

[1] Dipartimento di Informatica, Università di Torino
[2] Department of Computing, Imperial College London

Abstract. A multiparty session forms a unit of structured interactions among many participants which follow a prescribed scenario specified as a global type signature. This paper develops, besides a more traditional *communication* type system, a novel static *interaction* type system for global progress in dynamically interleaved multiparty sessions.

1 Introduction

Widespread use of message-based communication for developing network applications to combine numerous distributed services has provoked urgent interest in structuring series of interactions to specify and implement program communication-safe software. The actual development of such applications still leaves to the programmer much of the responsibility in guaranteeing that communication will evolve as agreed by all the involved distributed peers. *Multiparty session type discipline* proposed in [12] offers a type-theoretic framework to validate a message-exchange among concurrently running multiple peers in the distributed environment, generalising the existing binary session types [10,11]; interaction sequences are abstracted as a global type signature, which precisely declares how multiple peers communicate and synchronise with each other.

The multiparty sessions aim to retain the powerful dynamic features from the original binary sessions, incorporating features such as recursion and choice of interactions. Among features, *session delegation* is a key operation which permits to rely on other parties for completing specific tasks transparently in a type safe manner. When this mechanism is extended to multiparty interactions engaged in two or more specifications simultaneously, further complex interactions can be modelled. Each multiparty session following a distinct global type can be dynamically *interleaved* by other sessions at runtime either implicitly via communications belonging to different sessions or explicitly via session delegation.

Previous work on multiparty session types [12] has provided a limited progress property ensured only within a single session, ignoring this dynamic nature. More precisely, although the previous system assures that the multiple participants respect the protocol, by checking the types of exchanged messages and the order of communications in a

* The work is partially supported by IST-3-016004-IP-09 SENSORIA, EPSRC GR/T03208, EPSRC EP/F003757 and IST2005-015905 MOBIUS.

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 418–433, 2008.

single session, it cannot guarantee a *global progress*, i.e, that a protocol which merges several global scenarios will not get stuck in the middle of a session. This limitation prohibits to ensure a successful termination of a transaction, making the framework practically inapplicable to a large size of dynamically reconfigured conversations.

This paper develops, besides a more traditional *communication* type system (§ 3), a novel static *interaction* type system (§ 4) for global progress in dynamically interleaved multiparty, asynchronous sessions. High-level session processes equipped with global signatures are translated into low-level processes which have explicit senders and receivers. Type-soundness of low-level processes is guaranteed against the local, compositional communication type system.

The new calculus for multiparty sessions offers three technical merits without sacrificing the original simplicity and expressivity in [12]. First it avoids the overhead of global linearity-check in [12]; secondly it provides a more liberal policy in the use of variables, both in delegation and in recursive definitions; finally it implicitly provides each participant of a service with a runtime channel indexed by its role with which he can communicate with all the other participants, permitting also broadcast in a natural way. The use of indexed channels, moreover, permits to define a light-weight interaction type system for global progress.

The interaction type system automatically infers causalities of channels for the low level processes, ensuring the entire protocol, starting from the high-level processes which consist of multiple sessions, does not get stuck at intermediate sessions also in the presence of implicit and explicit session interleaving.

Full definitions and the proofs are at http://www.di.unito.it/ dezani/papers/bcdddy.pdf

2 Syntax and Operational Semantics

Merging Two Conversations: Three-Buyer Protocol. We introduce our calculus through an example, the three-buyer protocol, extending the two-buyer protocol from [12], which includes the new features, session-multicasting and dynamically merging of two conversations. The overall scenario, involving a Seller (S), Alice (A), Bob (B) and Carol (C), proceeds as follows.

1. Alice sends a book title to Seller, then Seller sends back a quote to Alice and Bob. Then Alice tells Bob how much she can contribute.
2. If the price is within Bob's budget, Bob notifies both Seller and Alice he accepts, then sends his address, and Seller sends back the delivery date.
3. If the price exceeds the budget, Bob asks Carol to collaborate together by establishing a new session. Then Bob sends how much Carol must pay, then *delegates* the remaining interactions with Alice and Seller to Carol.
4. If the rest of the price is within Carol's budget, Carol accepts the quote and notifies Alice, Bob and Seller, and continues the rest of the protocol with Seller and Alice transparently, *as if she were Bob*. Otherwise she notifies Alice, Bob and Seller to quit the protocol.

Then multiparty session programming consists of two steps: specifying the intended communication protocols using global types, and implementing these protocols using

processes. The specifications of the three-buyer protocol are given as two separated global types: one is G_a among Alice, Bob and Seller and the other is G_b between Bob and Carol. We write principals with legible symbols though they will actually be coded by numbers: in G_a we have $\mathtt{S} = 3$, $\mathtt{A} = 1$ and $\mathtt{B} = 2$, while in G_b we have $\mathtt{B} = 2$, $\mathtt{C} = 1$.

$G_a =$

1. $\mathtt{A} \longrightarrow \mathtt{S} : \quad \langle \mathsf{string} \rangle.$
2. $\mathtt{S} \longrightarrow \{\mathtt{A},\mathtt{B}\} : \langle \mathsf{int} \rangle.$
3. $\mathtt{A} \longrightarrow \mathtt{B} : \quad \langle \mathsf{int} \rangle.$
4. $\mathtt{B} \longrightarrow \{\mathtt{S},\mathtt{A}\} : \{\mathsf{ok} : \mathtt{B} \longrightarrow \mathtt{S} : \langle \mathsf{string} \rangle.$
5. $\qquad\qquad\qquad\qquad\quad \mathtt{S} \longrightarrow \mathtt{B} : \langle \mathsf{date} \rangle ; \mathsf{end}$
6. $\qquad\qquad\quad \mathsf{quit} : \mathsf{end}\}$

$G_b =$

1. $\mathtt{B} \longrightarrow \mathtt{C} : \langle \mathsf{int} \rangle.$
2. $\mathtt{B} \longrightarrow \mathtt{C} : \langle T \rangle.$
3. $\mathtt{C} \longrightarrow \mathtt{B} : \{\mathsf{ok} : \mathsf{end}, \quad \mathsf{quit} : \mathsf{end}\}.$

$T =$

$\oplus(\{\mathtt{S},\mathtt{A}\},$
$\{\mathsf{ok} : !\langle \mathtt{S}, \mathsf{string} \rangle ; ?\langle \mathtt{S}, \mathsf{date} \rangle ; \mathsf{end},$
$\mathsf{quit} : \mathsf{end}\})$

The types give a global view of the two conversations, directly abstracting the scenario given by the diagram. In G_a, line 1 denotes A sends a string value to S. Line 2 says S multicasts the same integer value to A and B and line 3 says that A sends an integer to B. In lines 4-6 B sends either ok or quit to S and A. In the first case B sends a string to S and receives a date from S, in the second case there are no further communications.

Line 2 in G_b represents the delegation of the capability specified by the action type T of channels (formally defined later) from B to C (note that S and A in T concern the session on a).

We now give the code, associated to G_a and G_b, for S, A, B and C in a "user" syntax formally defined in the following section:

$$\mathrm{S} = \bar{a}[3](y_3).y_3?(\mathit{title});y_3!\langle \mathit{quote} \rangle;y_3\&\{\mathsf{ok} : y_3?(\mathit{address});y_3!\langle \mathit{date} \rangle;\mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$$

$$\mathrm{A} = a[1](y_1).y_1!\langle \texttt{"Title"} \rangle;y_1?(\mathit{quote});y_1!\langle \mathit{quote}\ \mathsf{div}\ 2 \rangle;y_1\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$$

$$\begin{aligned}\mathrm{B} = {} & a[2](y_2).y_2?(\mathit{quote});y_2?(\mathit{contrib});\\ & \text{if } (\mathit{quote} \text{ - } \mathit{contrib} < 100) \text{ then } \ y_2 \oplus \mathsf{ok};y_2!\langle \texttt{"Address"} \rangle;y_2?(\mathit{date});\mathbf{0}\\ & \text{else } \bar{b}[2](z_2).z_2!\langle \mathit{quote} \text{ - } \mathit{contrib} \text{ - } 99 \rangle;z_2!\langle\langle y_2 \rangle\rangle;z_2\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\}\end{aligned}$$

$$\begin{aligned}\mathrm{C} = {} & b[1](z_1).z_1?(x);z_1?((t));\\ & \text{if } (x < 100) \text{ then } z_1 \oplus \mathsf{ok};t \oplus \mathsf{ok};t!\langle \texttt{"Address"} \rangle;t?(\mathit{date});\mathbf{0}\\ & \text{else } z_1 \oplus \mathsf{quit};t \oplus \mathsf{quit};\mathbf{0}\end{aligned}$$

Session name a establishes the session corresponding to G_a. S initiates a session involving three bodies as third participant by $\bar{a}[3](y_3)$: A and B participate as first and second participants by $a[1](y_1)$ and $a[2](y_2)$, respectively. Then S, A and B communicate using the channels y_3, y_1 and y_2, respectively. Each channel $y_\mathtt{p}$ can be seen as a port connecting participant p with all other ones; the receivers of the data sent on $y_\mathtt{p}$ are specified by the global type (this information will be included in the runtime code). The first line of G_a is implemented by the input and output actions $y_3?(\mathit{title})$ and $y_1!\langle \texttt{"Title"} \rangle$. The last line of G_b is implemented by the branching and selection actions $z_2\&\{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\}$ and $z_1 \oplus \mathsf{ok}$, $z_1 \oplus \mathsf{quit}$.

In B, if the quote minus A's contribution exceeds 100€ (i.e. $\mathit{quote} \text{ - } \mathit{contrib} \geq 100$), another session between B and C is established dynamically through shared name b. The delegation is performed by passing the channel y_2 from B to C (actions $z_2!\langle\langle y_2 \rangle\rangle$ and $z_1?((t))$), and so the rest of the session is carried out by C with S and A. We can

Table 1. Syntax for user-defined processes

$P ::= \bar{u}[n](y).P$		Multicast Request
$\mid u[\mathtt{p}](y).P$		Accept
$\mid y!\langle e\rangle;P$		Value sending
$\mid y?(x);P$		Value reception
$\mid y!\langle\langle z\rangle\rangle;P$		Session delegation
$\mid y?((z));P$		Session reception
$\mid y \oplus l;P$		Selection
$\mid y\&\{l_i : P_i\}_{i\in I}$		Branching
$u ::= x \mid a$		Identifier
$v ::= a \mid \mathsf{true} \mid \mathsf{false}$		Value
$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$		Conditional
$\mid P \mid Q$		Parallel
$\mid \mathbf{0}$		Inaction
$\mid (\nu a)P$		Hiding
$\mid \mathsf{def}\ D\ \mathsf{in}\ P$		Recursion
$\mid X\langle e,y\rangle$		Process call
$e ::= v \mid x$		
$\mid e\ \mathsf{and}\ e' \mid \mathsf{not}\ e \ldots$		Expression
$D ::= X(x,y) = P$		Declaration

further enrich this protocol with recursive-branching behaviours in interleaved sessions (for example, C can repeatedly negotiate the quote with S as if she were B). What we want to guarantee by static type-checking is that the whole conversation between the four parties preserves progress as if it were a single conversation.

Syntax for Multiparty Sessions. The syntax for processes initially written by the user, called *user-defined processes*, is based on [12]. We start from the following sets: *service names*, ranged over by $a,b,\ldots$ (representing public names of endpoints), *value variables*, ranged over by $x,x',\ldots$, *identifiers* , i.e., service names and variables, ranged over by $u,w,\ldots$, *channel variables*, ranged over by $y,z,t\ldots$, *labels*, ranged over by $l,l',\ldots$ (functioning like method names or labels in labelled records); *process variables*, ranged over by $X,Y,\ldots$ (used for representing recursive behaviour). Then *processes*, ranged over by $P,Q\ldots$, and *expressions*, ranged over by $e,e',\ldots$, are given by the grammar in Table 1.

For the primitives for session initiation, $\bar{u}[n](y).P$ initiates a new session through an identifier u (which represents a shared interaction point) with the other multiple participants, each of shape $u[\mathtt{p}](y).Q_{\mathtt{p}}$ where $1 \leq \mathtt{p} \leq n-1$. The (bound) variable y is the channel used to do the communications. We call p, q,... (ranging over natural numbers) the *participants* of a session. Session communications (communications that take place inside an established session) are performed using the next three pairs of primitives: the sending and receiving of a value; the session delegation and reception (where the former delegates to the latter the capability to participate in a session by passing a channel associated with the session); and the selection and branching (where the former chooses one of the branches offered by the latter). The rest of the syntax is standard from [11].

Global Types. A *global type*, ranged over by $G,G',..$ describes the whole conversation scenario of a multiparty session as a type signature. Its grammar is given below:

Global	$G ::= \mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \langle U\rangle.G'$	Exchange	$U ::= S \mid T$
	$\mid \mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \{l_i : G_i\}_{i\in I}$	Sorts	$S ::= \mathsf{bool} \mid \ldots \mid G$
	$\mid \mu\mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end}$		

We simplify the syntax in [12] by eliminating channels and parallel compositions, while preserving the original expressivity (see § 5).

Table 2. Runtime syntax: the other syntactic forms are as in Table 1

$P ::= c!\langle\{\mathtt{p}_k\}_{k\in K}, e\rangle; P$		Value sending	$\mid c \oplus \langle\{\mathtt{p}_k\}_{k\in K}, l\rangle; P$		Selection
$\mid c?(\mathtt{p}, x); P$		Value reception	$\mid c\&(\mathtt{p}, \{l_i : P_i\}_{i\in I})$		Branching
$\mid c!\langle\langle \mathtt{p}, c'\rangle\rangle; P$		Session delegation	$\mid (\nu s)P$		Hiding session
$\mid c?((\mathtt{q}, y)); P$		Session reception	$\mid s : h$		Named queue
			$\mid \ldots$		
$c ::= y \mid s[\mathtt{p}]$					Channel
$m ::= (\mathtt{q}, \{\mathtt{p}_k\}_{k\in K}, v) \mid (\mathtt{q}, \mathtt{p}, s[\mathtt{p}']) \mid (\mathtt{q}, \{\mathtt{p}_k\}_{k\in K}, l)$					Message in transit
$h ::= m \cdot h \mid \varnothing$					Queue

The global type $\mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \langle U\rangle.G'$ says that participant $\mathtt{p}$ multicasts a message of type U to participants $\mathtt{p}_k$ ($k \in K$) and then interactions described in G' take place. *Exchange types* $U, U', \ldots$ consist of *sorts* types $S, S', \ldots$ for values (either base types or global types), and *action* types $T, T', \ldots$ for channels (discussed in §3). Type $\mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \{l_i : G_i\}_{i\in I}$ says participant $\mathtt{p}$ multicasts one of the labels l_i to participants $\mathtt{p}_k$ ($k \in K$). If l_j is sent, interactions described in G_j take place. Type $\mu\mathbf{t}.G$ is a recursive type, assuming type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way, i.e. type variables only appear under some prefix. We take an *equi-recursive* view of recursive types, not distinguishing between $\mu\mathbf{t}.G$ and its unfolding $G\{\mu\mathbf{t}.G/\mathbf{t}\}$ [18] (§21.8). We assume that G in the grammar of sorts is closed, i.e., without free type variables. Type $\mathtt{end}$ represents the termination of the session. We often write $\mathtt{p} \to \mathtt{p}'$ for $\mathtt{p} \to \{\mathtt{p}'\}$.

Runtime Syntax. User defined processes equipped with global types are executed through a translation into runtime processes. The runtime syntax (Table 2) differs from the syntax of Table 1 since the input/output operations (including the delegation ones) specify the sender and the receiver, respectively. Thus, $c!\langle\{\mathtt{p}_k\}_{k\in K}, e\rangle$ sends a value to all the participants in $\{\mathtt{p}_k\}_{k\in K}$; accordingly, $c?(\mathtt{p}, x)$ denotes the intention of receiving a value from the participant $\mathtt{p}$. The same holds for delegation/reception (but the receiver is only one) and selection/branching.

We call $s[\mathtt{p}]$ a *channel with role*: it represents the channel of the participant $\mathtt{p}$ in the session s. We use c to range over variables and channels with roles. As in [12], in order to model TCP-like asynchronous communications (message order preservation and sender-non-blocking), we use the queues of messages in a session, denoted by h; a message in a queue can be a value message, $(\mathtt{q}, \{\mathtt{p}_k\}_{k\in K}, v)$, indicating that the value v was sent by the participant $\mathtt{q}$ and the recipients are all the participants in $\{\mathtt{p}_k\}_{k\in K}$; a channel message (delegation), $(\mathtt{q}, \mathtt{p}', s[\mathtt{p}])$, indicating that $\mathtt{q}$ delegates to $\mathtt{p}'$ the role of $\mathtt{p}$ on the session s (represented by the channel with role $s[\mathtt{p}]$); and a label message, $(\mathtt{q}, \{\mathtt{p}_k\}_{k\in K}, l)$ (similar to a value message). The empty queue is denoted by $\varnothing$. With some abuse of notation we will write $h \cdot m$ to denote that m is the last element included in h and $m \cdot h$ to denote that m is the head of h. By $s : h$ we denote the queue h of the session s. In $(\nu s)P$ all occurrences of $s[\mathtt{p}]$ and the queue s are bound. Queues and the channel with role are generated by the operational semantics (described later).

We present the translation of Bob (B) in the three-buyer protocol with the runtime syntax: the only difference is that all input/output operations specify also the sender and the receiver, respectively.

$$\begin{aligned}\mathrm{B} = a[2](y_2).y_2?(3,\mathit{quote});y_2?(1,\mathit{contrib});\\ \quad \mathsf{if}\ (\mathit{quote}\text{ - }\mathit{contrib} < 100)\ \mathsf{then}\ y_2 \oplus \langle\{1,3\},\mathtt{ok}\rangle;y_2!\langle\{3\},\mathtt{"Address"}\rangle;y_2?(3,\mathit{date});\mathbf{0}\\ \quad \mathsf{else}\ \bar{b}[2](z_2).z_2!\langle\{1\},\mathit{quote}\text{ - }\mathit{contrib}\text{ - }99\rangle;z_2!\langle\langle 1,y_2\rangle\rangle;z_2\&(1,\{\mathtt{ok}:\mathbf{0},\ \mathtt{quit}:\mathbf{0}\}).\end{aligned}$$

It should be clear from this example that starting from a global type and user-defined processes respecting the global type it is possible to add sender and receivers to each communication obtaining in this way processes written in the runtime syntax.

We call *pure* a process which does not contain message queues.

Operational Semantics. Table 3 shows the basic rules of the process reduction relation $P \longrightarrow P'$. Rule [Link] describes the initiation of a new session among n participants that synchronises over the service name a. The last participant $\bar{a}[n](y_n).P_n$, distinguished by the overbar on the service name, specifies the number n of participants. For this reason we call it the *initiator* of the session. Obviously each session must have a unique initiator. After the connection, the participants will share the private session name s, and the queue associated to s, which is initialized as empty. The variables $y_{\mathtt{p}}$ in each participant $\mathtt{p}$ will then be replaced with the corresponding channel with role, $s[\mathtt{p}]$. The output rules [Send], [Deleg] and [Label] push values, channels and labels, respectively, into the queue of the session s (in rule [Send], $e \downarrow v$ denotes the evaluation of the expression e to the value v). The rules [Recv], [Srec] and [Branch] perform the corresponding complementary operations. Note that these operations check that the sender matches, and also that the message is actually meant for the receiver (in particular, for [Recv], we need to remove the receiving participant from the set of the receivers in order to avoid reading the same message more than once).

Processes are considered modulo structural equivalence, denoted by $\equiv$, and defined by adding the following rules for queues to the standard ones [17]:

$$s : h_1 \cdot (\mathtt{q},\{\mathtt{p}_k\}_{k\in K},z) \cdot (\mathtt{q}',\{\mathtt{p}_k\}_{k\in K'},z') \cdot h_2 \equiv s : h_1 \cdot (\mathtt{q}',\{\mathtt{p}_k\}_{k\in K'},z') \cdot (\mathtt{q},\{\mathtt{p}_k\}_{k\in K},z) \cdot h_2 \quad \text{if } K\cap K' = \emptyset \text{ or } \mathtt{q}\neq\mathtt{q}'$$

$$s : (\mathtt{q},\emptyset,v)\cdot h \equiv s : h \qquad s : (\mathtt{q},\emptyset,l)\cdot h \equiv s : h$$

where z ranges over v, $s[\mathtt{p}]$ and l. The first rule permits rearranging messages when the senders or the receivers are not the same, and also splitting a message for multiple recipients. The last two rules garbage-collect messages that have already been read by all the intended recipients. We use $\longrightarrow^*$ and $\not\longrightarrow$ with the expected meanings.

Table 3. Selected reduction rules

$$a[1](y_1).P_1 \mid \ldots \mid \bar{a}[n](y_n).P_n \longrightarrow (\nu s)(P_1\{s[1]/y_1\} \mid \ldots \mid P_n\{s[n]/y_n\} \mid s : \varnothing) \quad \text{[Link]}$$

$$s[\mathtt{p}]!\langle\{\mathtt{p}_k\}_{k\in K},e\rangle;P \mid s : h \longrightarrow P \mid s : h\cdot(\mathtt{p},\{\mathtt{p}_k\}_{k\in K},v) \quad (e\downarrow v) \quad \text{[Send]}$$

$$s[\mathtt{p}]!\langle\langle\mathtt{q},s'[\mathtt{p}']\rangle\rangle;P \mid s : h \longrightarrow P \mid s : h\cdot(\mathtt{p},\mathtt{q},s'[\mathtt{p}']) \quad \text{[Deleg]}$$

$$s[\mathtt{p}] \oplus \langle\{\mathtt{p}_k\}_{k\in K},l\rangle;P \mid s : h \longrightarrow P \mid s : h\cdot(\mathtt{p},\{\mathtt{p}_k\}_{k\in K},l) \quad \text{[Label]}$$

$$s[\mathtt{p}_j]?(\mathtt{q},x);P \mid s : (\mathtt{q},\{\mathtt{p}_k\}_{k\in K},v)\cdot h \longrightarrow P\{v/x\} \mid s : (\mathtt{q},\{\mathtt{p}_k\}_{k\in K\setminus j},v)\cdot h \quad (j\in K) \quad \text{[Recv]}$$

$$s[\mathtt{p}]?((\mathtt{q},y));P \mid s : (\mathtt{q},\mathtt{p},s'[\mathtt{p}'])\cdot h \longrightarrow P\{s'[\mathtt{p}']/y\} \mid s : h \quad \text{[Srec]}$$

$$s[\mathtt{p}_j]\&(\mathtt{q},\{l_i : P_i\}_{i\in I}) \mid s : (\mathtt{q},\{\mathtt{p}_k\}_{k\in K},l_{i_0})\cdot h \longrightarrow P_{i_0} \mid s : (\mathtt{q},\{\mathtt{p}_k\}_{k\in K\setminus j},l_{i_0})\cdot h \quad (j\in K)\ \ (i_0\in I) \quad \text{[Branch]}$$

3 Communication Type System

The previous section defines the syntax and the global types. This section introduces the communication type system, by which we can check type soundness of the communications which take place inside single sessions.

Types and Typing Rules for Pure Runtime Processes. We first define the local types of pure processes, called *action types*. While global types represent the whole protocol, action types correspond to the communication actions, representing sessions from the view-points of single participants.

$$\begin{array}{llll} \text{Action} & T ::= & !\langle\{\mathtt{p}_k\}_{k\in K}, U\rangle;T & \textit{send} \\ & \mid & ?(\mathtt{p}, U);T & \textit{receive} \\ & \mid & \oplus\langle\{\mathtt{p}_k\}_{k\in K}, \{l_i : T_i\}_{i\in I}\rangle & \textit{selection} \\ & \mid & \&(\mathtt{p}, \{l_i : T_i\}_{i\in I}) & \textit{branching} \\ & \mid & \mu\mathbf{t}.T & \textit{recursive} \\ & \mid & \mathbf{t} & \textit{variable} \\ & \mid & \mathsf{end} & \textit{end} \end{array}$$

The *send type* $!\langle\{\mathtt{p}_k\}_{k\in K}, U\rangle;T$ expresses the sending to all $\mathtt{p}_k$ for $k \in K$ of a value or of a channel of type U, followed by the communications of T. The *selection type* $\oplus\langle\{\mathtt{p}_k\}_{k\in K}, \{l_i : T_i\}_{i\in I}\rangle$ represents the transmission to all $\mathtt{p}_k$ for $k \in K$ of a label l_i chosen in the set $\{l_i \mid i \in I\}$ followed by the communications described by T_i. The *receive* and *branching* are dual and only need one sender. Other types are standard.

The relation between action and global types is formalised by the notion of projection as in [12]. The *projection of G onto* $\mathtt{q}$ ($G \upharpoonright \mathtt{q}$) is defined by induction on G:

$$(\mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \langle U\rangle.G') \upharpoonright \mathtt{q} = \begin{cases} !\langle\{\mathtt{p}_k\}_{k\in K}, U\rangle;(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}, \\ ?(\mathtt{p}, U);(G' \upharpoonright \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}_k \text{ for some } k \in K, \\ G' \upharpoonright \mathtt{q} & \text{otherwise.} \end{cases}$$

$$(\mathtt{p} \to \{\mathtt{p}_k\}_{k\in K} : \{l_i : G_i\}_{i\in I}) \upharpoonright \mathtt{q} = \begin{cases} \oplus(\{\mathtt{p}_k\}_{k\in K}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}) & \text{if } \mathtt{q} = \mathtt{p} \\ \&(\mathtt{p}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}) & \text{if } \mathtt{q} = \mathtt{p}_k \text{ for some } k \in K \\ G_1 \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \neq \mathtt{p}, \mathtt{q} \neq \mathtt{p}_k \forall k \in K \text{ and} \\ & G_i \upharpoonright \mathtt{q} = G_j \upharpoonright \mathtt{q} \text{ for all } i, j \in I. \end{cases}$$

$$(\mu\mathbf{t}.G) \upharpoonright \mathtt{q} = \mu\mathbf{t}.(G \upharpoonright \mathtt{q}) \quad \mathbf{t} \upharpoonright \mathtt{q} = \mathbf{t} \quad \mathsf{end} \upharpoonright \mathtt{q} = \mathsf{end}.$$

As an example, we list two of the projections of the global types G_a and G_b of the three-buyer protocol:

$$G_a \upharpoonright 3 = ?\langle 1, \mathsf{string}\rangle; !\langle\{1,2\}, \mathsf{int}\rangle; \&(2, \{\mathsf{ok} : ?\langle 2, \mathsf{string}\rangle; !\langle\{2\}, \mathsf{date}\rangle; \mathsf{end}, \mathsf{quit} : \mathsf{end}\})$$
$$G_b \upharpoonright 1 = ?\langle 2, \mathsf{int}\rangle; ?\langle 2, T\rangle; \oplus\langle\{2\}, \{\mathsf{ok} : \mathsf{end}, \mathsf{quit} : \mathsf{end}\}\rangle$$

where $T = \oplus\langle\{1,3\}, \{\mathsf{ok} : !\langle\{3\}, \mathsf{string}\rangle; ?\langle 3, \mathsf{date}\rangle; \mathsf{end},\ \mathsf{quit} : \mathsf{end}\}\rangle$.

The typing judgements for expressions and pure processes are of the shape:

$$\Gamma \vdash e : S \text{ and } \Gamma \vdash P \triangleright \Delta$$

where Γ is the *standard environment* which associates variables to sort types, service names to global types and process variables to pairs of sort types and action types; Δ is the *session environment* which associates channels to action types. Formally we define:

$$\Gamma ::= \emptyset \mid \Gamma, u : S \mid \Gamma, X : S\ T \text{ and } \Delta ::= \emptyset \mid \Delta, c : T$$

Table 4. Selected typing rules for pure processes

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright n \quad n = \mathrm{pn}(G)}{\Gamma \vdash \bar{u}[n](y).P \triangleright \Delta} \lfloor \textsc{MCast} \rfloor \qquad \frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright \mathtt{p}}{\Gamma \vdash u[\mathtt{p}](y).P \triangleright \Delta} \lfloor \textsc{MAcc} \rfloor$$

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle \{\mathtt{p}_k\}_{k \in K}, e \rangle; P \triangleright \Delta, c : !\langle \{\mathtt{p}_k\}_{k \in K}, S \rangle; T} \lfloor \textsc{Send} \rfloor \qquad \frac{\Gamma, x : S \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c?(\mathtt{q}, x); P \triangleright \Delta, c : ?(\mathtt{q}, S); T} \lfloor \textsc{Rcv} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle\langle \mathtt{p}, c' \rangle\rangle; P \triangleright \Delta, c : !\langle \mathtt{p}, T' \rangle; T, c' : T'} \lfloor \textsc{Deleg} \rfloor \qquad \frac{\Gamma \vdash P \triangleright \Delta, c : T, y : T'}{\Gamma \vdash c?((\mathtt{q}, y)); P \triangleright \Delta, c : ?(\mathtt{q}, T'); T} \lfloor \textsc{Srec} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta' \quad dom(\Delta) \cap dom(\Delta') = \emptyset}{\Gamma \vdash P \mid Q \triangleright \Delta \cup \Delta'} \lfloor \textsc{Conc} \rfloor$$

assuming that we can write $\Gamma, u : S$ only if u does not occur in Γ, briefly $u \notin dom(\Gamma)$ ($dom(\Gamma)$ denotes the domain of Γ, i.e. the set of identifiers which occur in Γ). We use the same convention for $X : S\ T$ and Δ.

Table 4 presents the interesting typing rules for pure processes. Rule $\lfloor$MCAST$\rfloor$ permits to type a service initiator identified by u, if the type of y is the n-th projection of the global type G of u and the number of participants in G (denoted by $\mathrm{pn}(G)$) is n. Rule $\lfloor$MACC$\rfloor$ permits to type the $\mathtt{p}$-th participant identified by u, which uses the channel y, if the type of y is the $\mathtt{p}$-th projection of the global type G of u. The successive six rules associate the input/output processes to the input/output types in the expected way. Note that, according to our notational convention on environments, in rule $\lfloor$DELEG$\rfloor$ the channel which is sent cannot appear in the session environment of the premise, i.e. $c' \notin dom(\Delta) \cup \{c\}$. Rule $\lfloor$CONC$\rfloor$ permits to put in parallel two processes only if their sessions environments have disjoint domains. For example we can derive:

$$\vdash t \oplus \langle \{1,3\}, \mathsf{ok} \rangle; t!\langle \{3\}, \texttt{"Address"} \rangle; t?(3, date); \mathbf{0} \triangleright \{t : T\}$$

where $T = \oplus\langle \{1,3\}, \{\mathsf{ok} : !(\{3\}, \mathsf{string}); ?\langle 3, \mathsf{date} \rangle; \mathsf{end},\ \mathsf{quit} : \mathsf{end}\} \rangle$.

In the typing of the example of the three-buyer protocol the types of the channels y_3 and z_1 are the third projection of G_a and the first projection of G_b, respectively. By applying rule $\lfloor$MCAST$\rfloor$ we can then derive $a : G_a \vdash \mathrm{S} \triangleright \emptyset$. Similarly by applying rule $\lfloor$MACC$\rfloor$ we can derive $b : G_b \vdash \mathrm{C} \triangleright \emptyset$.

Types and Typing Rules for Runtime Processes. We now extend the communication type system to processes containing queues.

Message $\mathtt{T}$ ::=	$!\langle \{\mathtt{p}_k\}_{k \in K}, U \rangle$	*message send*	Generalised $\mathtt{T}$ ::=	T	*action*
|	$\oplus\langle \{\mathtt{p}_k\}_{k \in K}, l \rangle$	*message selection*	|	$\mathtt{T}$	*message*
|	$\mathtt{T}; \mathtt{T}'$	*message sequence*	|	$\mathtt{T}; T$	*continuation*

Message types are the types for queues: they represent the messages contained in the queues. The *message send type* $!\langle \{\mathtt{p}_k\}_{k \in K}, U \rangle$ expresses the communication to all $\mathtt{p}_k$ for $k \in K$ of a value or of a channel of type U. The *message selection type* $\oplus\langle \{\mathtt{p}_k\}_{k \in K}, l \rangle$ represents the communication to all $\mathtt{p}_k$ for $k \in K$ of the label l and $\mathtt{T}; \mathtt{T}'$ represents

sequencing of message types. For example $\oplus\langle\{1,3\},\mathsf{ok}\rangle$ is the message type for the message $(2,\{1,3\},\mathsf{ok})$.

A *generalised type* is either an action type, or a message type, or a message type followed by an action type. Type $\mathtt{T};T$ represents the continuation of the type $\mathtt{T}$ associated to a queue with the type T associated to a pure process. An example of generalised type is $\oplus\langle\{1,3\},\mathsf{ok}\rangle;!\langle\{3\},\mathsf{string}\rangle;?\langle 3,\mathsf{date}\rangle;\mathsf{end}$.

We start by defining the typing rules for single queues, in which the turnstile $\vdash$ is decorated with $\{s\}$ (where s is the session name of the current queue) and the session environments are mappings from channels to message types. The empty queue has empty session environment. Each message adds an output type to the current type of the channel which has the role of the message sender.

In order to type pure processes in parallel with queues, we need to use generalised types in session environments and further typing rules. The more interesting rules are:

$$\frac{\Gamma \vdash P \triangleright \Delta}{\Gamma \vdash_{\emptyset} P \triangleright \Delta}\ \lfloor\textsc{GInit}\rfloor \qquad \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Gamma \vdash_{\Sigma'} Q \triangleright \Delta' \quad \Sigma \cap \Sigma' = \emptyset}{\Gamma \vdash_{\Sigma\cup\Sigma'} P \mid Q \triangleright \Delta * \Delta'}\ \lfloor\textsc{GPar}\rfloor$$

where the judgement $\Gamma \vdash_{\Sigma} P \triangleright \Delta$ means that P contains the queues whose session names are in Σ. Rule $\lfloor\textsc{GInit}\rfloor$ promotes the typing of a pure process to the typing of an arbitrary process, since a pure process does not contain queues. When two arbitrary processes are put in parallel (rule $\lfloor\textsc{GPar}\rfloor$) we need to require that each session name is associated to at most one queue (condition $\Sigma \cap \Sigma' = \emptyset$). In composing the two session environments we want to put in sequence a message type and an action type for the same channel with role. For this reason we define the composition $*$ between local types as:

$$\mathtt{T} * \mathtt{T}' = \begin{cases} \mathtt{T};\mathtt{T}' & \text{if } \mathtt{T} \text{ is a message type,} \\ \mathtt{T}';\mathtt{T} & \text{if } \mathtt{T}' \text{ is a message type,} \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ represents failure of typing. We extend $*$ to session environments as expected:
$\Delta * \Delta' = \Delta \backslash dom(\Delta') \cup \Delta' \backslash dom(\Delta) \cup \{c : \mathtt{T} * \mathtt{T}' \mid c : \mathtt{T} \in \Delta \ \&\ c : \mathtt{T}' \in \Delta'\}$.
Note that $*$ is commutative, i.e. $\Delta * \Delta' = \Delta' * \Delta$. Also if we can derive message types only for channels with roles, we consider the channel variables in the definition of $*$ for session environments since we want to get for example $\{y : \mathsf{end}\} * \{y : \mathsf{end}\} = \bot$. An example of derivable judgement is:

$\vdash_{\{s\}} P \mid s : (3,\{1,2\},\mathsf{ok}) \triangleright \{s[3] : \oplus\langle\{1,2\},\mathsf{ok}\rangle;!\langle\{1\},\mathsf{string}\rangle;?\langle 1,\mathsf{date}\rangle;\mathsf{end}\}$

where $P = s[3]!\langle\{1\},\texttt{"Address"}\rangle;s[3]?(1,date);\mathbf{0}$.

Subject Reduction. Since session environments represent the forthcoming communications, by reducing processes session environments can change. This can be formalised as in [12] by introducing the notion of reduction of session environments, whose rules are:

- $\{s[\mathtt{p}] : !\langle\{\mathtt{p}_k\}_{k\in K},U\rangle;\ T, s[\mathtt{p}_j] : ?(\mathtt{p},U);T'\} \Rightarrow \{s[\mathtt{p}] : !\langle\{\mathtt{p}_k\}_{k\in K\setminus j},U\rangle;\ T, s[\mathtt{p}_j] : T'\}$ if $j \in K$
- $\{s[\mathtt{p}] : T;\oplus\langle\{\mathtt{p}_k\}_{k\in K},\{l_i : T_i\}_{i\in I}\rangle\} \Rightarrow \{s[\mathtt{p}] : T;\oplus\langle\{\mathtt{p}_k\}_{k\in K},l_i\rangle;T_i\}$
- $\{s[\mathtt{p}] : \oplus\langle\{\mathtt{p}_k\}_{k\in K},l\rangle;T, s[\mathtt{p}_j] : \&(\mathtt{p},\{l_i : T_i\}_{i\in I})\} \Rightarrow \{s[\mathtt{p}] : \oplus\langle\{\mathtt{p}_k\}_{k\in K\setminus j},l\rangle;T, s[\mathtt{p}_j] : T_i\}$ if $j \in K$ and $l = l_i$
- $\{s[\mathtt{p}] : !\langle\emptyset,U\rangle;\ T\} \Rightarrow \{s[\mathtt{p}] : T\} \qquad \{s[\mathtt{p}] : \oplus\langle\emptyset,l\rangle;T\} \Rightarrow \{s[\mathtt{p}] : T\}$
- $\Delta \cup \Delta'' \Rightarrow \Delta' \cup \Delta''$ if $\Delta \Rightarrow \Delta'$.

The first rule corresponds to the reception of a value or channel by the participant $\mathtt{p}_j$, the second rule corresponds to the choice of the label l_i and the third rule corresponds to the reception of the label l by the participant $\mathtt{p}_j$. The fourth and the fifth rules garbage collect read messages.

Using the above notion we can state type preservation under reduction as follows:

Theorem 1 (Type Preservation). *If $\Gamma \vdash_\Sigma P \triangleright \Delta$ and $P \longrightarrow^* P'$, then $\Gamma \vdash_\Sigma P' \triangleright \Delta'$ for some Δ' such that $\Delta \Rightarrow^* \Delta'$.*

Note that the communication safety [12, Theorem 5.5] is a corollary of this theorem. Thus the user-defined processes with the global types can safely communicate since their runtime translation is typable by the communication type system.

4 Progress

This section studies progress: informally, we say that a process has the progress property if it can never reach a deadlock state, i.e., if it never reduces to a process which contains open sessions (this amounts to containing channels with roles) and which is irreducible in any inactive context (represented by another inactive process running in parallel).

Definition 1 (Progress). *A process P has the* progress property *if $P \longrightarrow^* P'$ implies that either P' does not contain channels with roles or $P' \mid Q \longrightarrow$ for some Q such that $P' \mid Q$ is well typed and $Q \not\longrightarrow$.*

We will give an interaction type system which ensures that the typable processes always have the progress property.

Let us say that a *channel qualifier* is either a session name or a channel variable. Let c be a channel, its channel qualifier $\ell(c)$ is defined by: (1) if $c = y$, then $\ell(c) = y$; (2) else if $c = s[\mathtt{p}]$, then $\ell(c) = s$. Let Λ, ranged over by λ, denote the set of all service names and all channel qualifiers.

The progress property will be analysed via three finite sets: two sets $\mathscr{N}$ and $\mathscr{B}$ of service names and a set $\mathscr{R} \subseteq \Lambda \cup (\Lambda \times \Lambda)$. The set $\mathscr{N}$ collects the service names which are interleaved following the nesting policy. The set $\mathscr{B}$ collects the service names which can be bound. The Cartesian product $\Lambda \times \Lambda$, whose elements are denoted $\lambda \prec \lambda'$, represents a transitive relation. The meaning of $\lambda \prec \lambda'$ is that an input action involving a channel (qualified by) λ or belonging to service λ could block a communication action involving a channel (qualified by) λ' or belonging to service λ'. Moreover $\mathscr{R}$ includes all channel qualifiers and all service names which do not belong to $\mathscr{N}$ or $\mathscr{B}$ and which occur free in the current process. This will be useful to easily extend $\mathscr{R}$ in the assignment rules, as it will be pointed out below. We call $\mathscr{N}$ *nested service set*, $\mathscr{B}$ *bound service set* and $\mathscr{R}$ *channel relation* (even if only a subset of it is, strictly speaking, a relation). Let us give now some related definitions.

Definition 2. *Let $\mathscr{R} ::= \emptyset \mid \mathscr{R}, \lambda \mid \mathscr{R}, \lambda \prec \lambda'$.*

1. $\mathscr{B} \cup \{e\} = \begin{cases} \mathscr{B} \cup \{a\} & \text{if } e = a \text{ is a session name} \\ \mathscr{B} & \text{otherwise.} \end{cases}$

2. $\mathscr{R} \setminus \lambda = \{\lambda_1 \prec \lambda_2 \mid \lambda_1 \prec \lambda_2 \in \mathscr{R} \;\&\; \lambda_1 \neq \lambda \;\&\; \lambda_2 \neq \lambda\} \cup \{\lambda' \mid \lambda' \in \mathscr{R} \;\&\; \lambda' \neq \lambda\}$
3. $\mathscr{R} \setminus\setminus \lambda = \begin{cases} \mathscr{R} \setminus \lambda & \text{if } \lambda \text{ is minimal in } \mathscr{R} \\ \bot & \text{otherwise.} \end{cases}$
4. $\mathscr{R} \uplus \mathscr{R}' = (\mathscr{R} \cup \mathscr{R}')^+$
5. $\mathsf{pre}(\ell(c), \mathscr{R}) = \mathscr{R} \uplus \{\ell(c)\} \uplus \{\ell(c) \prec \lambda \mid \lambda \in \mathscr{R} \;\&\; \ell(c) \neq \lambda\}$

where $\mathscr{R}^+$ *is the transitive closure of (the relation part of)* $\mathscr{R}$ *and* λ *is* minimal *in* $\mathscr{R}$ *if* $\nexists \lambda' \prec \lambda \in \mathscr{R}$.

Note, as it easy to prove, that $\uplus$ is associative. A channel relation is *well formed* if it is irreflexive, and does not contain cycles. A channel relation $\mathscr{R}$ is *channel free* ($\mathsf{cf}(\mathscr{R})$) if it contains only service names.

In Table 5 we introduce selected rules for the interaction type system. The judgements are of the shape: $\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}$ where Θ is a set of *assumptions* of the shape $X[y] \blacktriangleright \mathscr{R} \;;\; \mathscr{N} \;;\; \mathscr{B}$ (for recursive definitions) with the variable y representing the channel parameter of X.

We say that a judgement $\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}$ is *coherent* if: (1) $\mathscr{R}$ is well formed; (2) $\mathscr{R} \cap (\mathscr{N} \cup \mathscr{B}) = \emptyset$. We assume that the typing rules are applicable if and only if *the judgements in the conclusion are coherent*.

We will give now an informal account of the interaction typing rules, through a set of examples. It is understood that all processes introduced in the examples can be typed with the communication typing rules given in the previous section.

Table 5. Selected interaction typing rules

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash \bar{a}[n](y).P \blacktriangleright \mathscr{R}\{a/y\}; \mathscr{N}; \mathscr{B}} \{\textsc{MCast}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash a[\mathrm{p}](y).P \blacktriangleright \mathscr{R}\{a/y\}; \mathscr{N}; \mathscr{B}} \{\textsc{MAcc}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash \bar{a}[n](y).P \blacktriangleright \mathscr{R} \setminus\setminus y; \mathscr{N} \cup \{a\}; \mathscr{B}} \{\textsc{MCastN}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash a[\mathrm{p}](y).P \blacktriangleright \mathscr{R} \setminus\setminus y; \mathscr{N} \cup \{a\}; \mathscr{B}} \{\textsc{MAccN}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad \mathsf{cf}(\mathscr{R} \setminus\setminus y)}{\Theta \vdash \bar{u}[n](y).P \blacktriangleright \mathscr{R} \setminus\setminus y; \mathscr{N}; \mathscr{B} \cup \{u\}} \{\textsc{MCastB}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad \mathsf{cf}(\mathscr{R} \setminus\setminus y)}{\Theta \vdash u[\mathrm{p}](y).P \blacktriangleright \mathscr{R} \setminus\setminus y; \mathscr{N}; \mathscr{B} \cup \{u\}} \{\textsc{MAccB}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash c!\langle\{\mathrm{p}_k\}_{k \in K}, e\rangle; P \blacktriangleright \{\ell(c)\} \cup \mathscr{R}; \mathscr{N}; \mathscr{B} \cup \{e\}} \{\textsc{Send}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash c?(\mathrm{q}, x); P \blacktriangleright \mathsf{pre}(\ell(c), \mathscr{R}); \mathscr{N}; \mathscr{B}} \{\textsc{Rcv}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B}}{\Theta \vdash c!\langle\langle \mathrm{p}', c'\rangle\rangle; P \blacktriangleright \{\ell(c), \ell(c'), \ell(c) \prec \ell(c')\} \uplus \mathscr{R}; \mathscr{N}; \mathscr{B}} \{\textsc{DeleG}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad \mathscr{R} \subseteq \{\ell(c), y, \ell(c) \prec y\}}{\Theta \vdash c?((\mathrm{q}, y)); P \blacktriangleright \{\ell(c)\}; \mathscr{N}; \mathscr{B}} \{\textsc{SRec}\}$$

$$\frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad \Theta \vdash Q \blacktriangleright \mathscr{R}'; \mathscr{N}'; \mathscr{B}'}{\Theta \vdash P \mid Q \blacktriangleright \mathscr{R} \uplus \mathscr{R}'; \mathscr{N} \cup \mathscr{N}'; \mathscr{B} \cup \mathscr{B}'} \{\textsc{Conc}\} \qquad \frac{\Theta \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \quad a \notin \mathscr{R} \cup \mathscr{N}}{\Theta \vdash (\nu a)P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \setminus a} \{\textsc{NRes}\}$$

$$\frac{}{\Theta, X[y] \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \vdash X\langle e, c\rangle \blacktriangleright \mathscr{R}\{\ell(c)/y\}; \mathscr{N}; \mathscr{B} \cup \{e\}} \{\textsc{Var}\}$$

$$\frac{\Theta, X[y] \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \vdash P \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \qquad \Theta, X[y] \blacktriangleright \mathscr{R}; \mathscr{N}; \mathscr{B} \vdash Q \blacktriangleright \mathscr{R}'; \mathscr{N}'; \mathscr{B}'}{\Theta \vdash \mathsf{def}\, X(x, y) = P \,\mathsf{in}\, Q \blacktriangleright \mathscr{R}'; \mathscr{N}'; \mathscr{B}'} \{\textsc{Def}\}$$

The crucial point to prove the progress property is to assure that a process, seen as a parallel composition of single threaded processes and queues, cannot be blocked in a configuration in which:

1. there are no thread ready for a session initialization (i.e. of the form $\bar{a}[n](y).P$ or $a[\mathrm{p}](y).P$). Otherwise the process could be reactivated by providing it with the right partners.
2. all subprocesses are either non-empty queues or processes waiting to perform an input action on a channel whose associated queue does not offer an appropriate message.

Progress inside a single service is assured by the communication typing rules in § 3. This will follow as an immediate corollary of Theorem 2. The channel relation is essentially defined to analyse the interactions between services: this is why in the definition of $\mathsf{pre}(\ell(c),\mathscr{R})$ we put the condition $\ell(c) \neq \lambda$. A basic point is that a loop in $\mathscr{R}$ represents the possibility of a deadlock state. For instance take the processes:

$$P_1 = b[1](y_1).\bar{a}[2](z_2).y_1?(2,x);z_2!\langle 1,\mathsf{false}\rangle;\mathbf{0}$$
$$P_2 = \bar{b}[2](y_2).a[1](z_1).z_1?(2,x');y_2!\langle 1,\mathsf{true}\rangle;\mathbf{0}.$$

In process P_1 we have that an input action on service b can block an output action on service a and this determines $b \prec a$. In process P_2 the situation is inverted, determining $a \prec b$. In $P_1 \mid P_2$ we will then have a loop $a \prec b \prec a$. In fact $P_1 \mid P_2$ reduces to

$$Q \;=\; (\nu s)(\nu r)\;\; (s[1]?(2,x);r[1]!\langle 2,\mathsf{false}\rangle;\mathbf{0} \mid r[2]?(1,x');s[2]!\langle 1,\mathsf{true}\rangle;\mathbf{0})$$

which is stuck. It is easy to see that services a and b have the same types, thus we could change b in a in P_1 and P_2 obtaining P_1' and P_2' with two instances of service a and a relation $a \prec a$. But also $P_1' \mid P_2'$ would reduce to Q. Hence we must forbid also loops on single service names (i.e. the channel relation cannot be reflexive).

Rule $\{\textsc{Rcv}\}$ asserts that the input action can block all other actions in P, while rule $\{\textsc{Send}\}$ simply adds $\ell(c)$ in $\mathscr{R}$ to register the presence of a communication action in P. In fact output is asynchronous, thus it can be always performed. Rule $\{\textsc{Deleg}\}$ is similar to $\{\textsc{Send}\}$ but asserts that a use of $\ell(c)$ must precede a use of $\ell(c')$: the relation $\ell(c) \prec \ell(c')$ needs to be registered since an action blocking $\ell(c)$ also blocks $\ell(c')$.

Three different sets of rules handle service initialisations. In rules $\{\textsc{MCast}\}$-$\{\textsc{MAcc}\}$, which are liberal on the occurrences of the channel y in P, the service name a replaces y in $\mathscr{R}$. Rules $\{\textsc{MCastn}\}$-$\{\textsc{MAccn}\}$ can be applied only if the channel y associated to a is minimal in $\mathscr{R}$ .This implies that once a is initialised in P all communication actions on the channel with role instantiating y must be performed before any input communication action on a different channel in P. The name a is added to the nested service set. Remarkably, via rules $\{\textsc{MCastn}\}$-$\{\textsc{MAccn}\}$ we can prove progress when services are nested, generalising the typing strategy of [6]. The rules $\{\textsc{MCastb}\}$ and $\{\textsc{MAccb}\}$ add u to the bound service set whenever u is a service name. These rules are much more restrictive: they require that y is the only free channel in P and that it is minimal. Thus no interaction with other channels or services is possible. This safely allows u to be a variable (since nothing is known about it before execution except its type) or a restricted name (since no channel with role can be made inaccessible at runtime by a restriction on u). Note that rule $\{\textsc{Nres}\}$ requires that a occurs neither in $\mathscr{R}$ nor in $\mathscr{N}$.

The sets $\mathscr{N}$ and $\mathscr{B}$ include all service names of a process P whose initialisations is typed with $\{\text{MCASTN}\}$-$\{\text{MACCN}\}$, $\{\text{MCASTB}\}$-$\{\text{MACCB}\}$, respectively. Note that for a service name which will replace a variable this is assured by the (conditional) addition of e to $\mathscr{B}$ in the conclusion of rule $\{\text{SEND}\}$. The sets $\mathscr{N}$ and $\mathscr{B}$ are used to assure, via the coherence condition $\mathscr{R} \cap (\mathscr{N} \cup \mathscr{B}) = \emptyset$, that *all* participants to the same service are typed either by the first two rules or by the remaining four. This is crucial to assure progress. Take for instance the processes P_1 and P_2 above. If we type the session initialisation on b using rule $\{\text{MACCN}\}$ or $\{\text{MACCB}\}$ in P_1 and rule $\{\text{MCAST}\}$ in P_2 no inconsistency would be detected. But rule $\{\text{CONC}\}$ does not type $P_1 \mid P_2$ owing to the coherence condition. Instead if we use $\{\text{MACC}\}$ in P_1, we detect the loop $a \prec b \prec a$. Note that we could not use $\{\text{MCASTN}\}$ or $\{\text{MCASTB}\}$ for b in P_2 since y_2 is not minimal.

Rules $\{\text{MCASTN}\}$-$\{\text{MACCN}\}$ are useful for typing delegation. An example is process B of the three-buyer protocol, in which the typing of the subprocess

$$z_2!\langle\{1\}, \mathit{quote}\text{ - }\mathit{contrib}\text{ - }99\rangle; z_2!\langle\langle 1, y_2\rangle\rangle; z_2\&(1, \{\mathsf{ok} : \mathbf{0},\ \mathsf{quit} : \mathbf{0}\})$$

gives $z_2 \prec y_2$. So by using rule $\{\text{MCAST}\}$ we would get first $b \prec y_2$ and then the cycle $y_2 \prec b \prec y_2$. Instead using rule $\{\text{MCASTN}\}$ for b we get in the final typing of B either $\{a\};\{b\};\emptyset$ or $\emptyset;\{a,b\};\emptyset$ according to we use either $\{\text{MCAST}\}$ or $\{\text{MCASTN}\}$ for a.

Rule $\{\text{SREC}\}$ avoids to create a process where two different roles in the same session are put in sequence. Following [23] we call this phenomenon self-delegation. As an example consider the processes

$$P_1 = b[1](z_1).a[1](y_1).y_1!\langle\langle 2, z_1\rangle\rangle; \mathbf{0}$$
$$P_2 = \bar{b}[2](z_2).\bar{a}[2](y_2).y_2?((1,x)); x?(2,w); z_2!\langle 1, \mathsf{false}\rangle; \mathbf{0}$$

and note that $P_1 \mid P_2$ reduces to $(\nu s)(\nu r)(s[1]?(2,w); s[2]!\langle 1, \mathsf{false}\rangle; \mathbf{0})$ which is stuck. Note that $P_1 \mid P_2$ is typable by the communication type system but P_2 is not typable by the interaction type system, since by typing $y_2?((1,x)); x?(2,w); z_2!\langle 1, \mathsf{false}\rangle; \mathbf{0}$ we get $y_2 \prec z_2$ which is forbidden by rule $\{\text{SREC}\}$.

A closed runtime process P is *initial* if it is typable both in the communication and in the interaction type systems. The progress property is assured for all computations that are generated from an initial process.

Theorem 2 (Progress). *All initial processes have the progress property.*

It is easy to verify that the (runtime) version of the three-buyer protocol can be typed in the interaction type system with $\{a\};\{b\};\emptyset$ and $\emptyset;\{a,b\};\emptyset$ according to which typing rules we use for the initialisation actions on the service name a. Therefore we get

Corollary 1. *The three-buyer protocol has the progress property.*

5 Conclusions and Related Work

The programming framework presented in this paper relies on the concept of global types that can be seen as the language to describe the model of the distributed communications, i.e., an abstract high-level view of the protocol that all the participants will have to respect in order to communicate in a multiparty communication. The programmer will then write the program to implement this communication protocol; the system

will use the global types (abstract model) and the program (implementation) to generate a runtime representation of the program which consists of the input/output operations decorated with explicit senders and receivers, according to the information provided in the global types. An alternative way could be that the programmer directly specifies the senders and the receivers in the communication operations as our low-level processes; the system could then infer the global types from the program. Our communication and interaction type systems will work as before in order to check the correctness and the progress of the program. Thus the programmer can choose between a top-down and a bottom-up style of programming, while relying on the same properties checked and guaranteed by the system.

We are currently designing and implementing a modelling and specification language with multiparty session types [19] for the standards of business and financial protocols with our industry collaborators [20,21]. This consists of three layers: the first layer is a global type which corresponds to a signature of class models in UML; the second one is for conversation models where signatures and variables for multiple conversations are integrated; and the third layer includes extensions of the existing languages (such as Java [13]) which implement conversation models. We are currently considering to extend this modelling framework with our type discipline so that we can specify and ensure progress for executable conversations.

Multiparty sessions. The first papers on multiparty session types are [2] and [12]. The work [2] uses a distributed calculus where each channel connects a master end-point and one or more slave endpoints; instead of global types, they solely use (recursion-free) local types. In type checking, local types are projected to binary sessions, so that type safety is ensured using duality, but it loses sequencing information: hence progress in a session interleaved with other sessions is not guaranteed.

The present calculus is an essential improvement from [12]; both processes and types in [12] share a vector of channels and each communication uses one of these channels, while our user processes and global types are simpler and user-friendly without these channels. The global types in [12] have a parallel composition operator, but its projectability from global to local types limits to disjoint senders and receivers; hence it does not increase expressivity.

The present calculus is more liberal than the calculus of [12] in the use of declarations, since the definition and the call of recursive processes are obliged to use the same channel variable in [12]. Similarly the delegation in [12] requires that the same channel is sent and received for ensuring subject reduction, as analysed in [23]. Our calculus solves this issue by having channels with roles, as in [9] (see the example at page 430). As a consequence some recursive processes, which are stuck in [12], are type-sound and reducible in our calculus, satisfying the interaction type system.

Different approaches to the description of service-oriented multiparty communications are taken in [3,4]. In [3], the global and local views of protocols are described in two different calculi and the agreement between these views becomes a bisimulation between processes; [4] proposes a distributed calculus which provides communications either inside sessions or inside locations, modelling merging running sessions. The type-safety and progress in interleaved sessions are left as an open problem in [4].

Progress. The majority of papers on service-oriented calculi only assure that clients are never stuck inside a *single* session, see [1,7,12] for detailed discussions, including comparisons between the session-based and the traditional behavioural type systems of mobile processes, e.g. [22,15]. We only say here that our interaction type system is inspired by deadlock-free typing systems [14,15,22]. In [1,7,12], structured session primitives help to give simpler typing systems for progress.

The first papers considering progress for interleaved sessions required the nesting of sessions in Java [8,6] and SOC [1,16,5]. The present approach significantly improves the binary session system for progress in [7] by treating the following points:

(1) asynchrony of the communication with queues, which enhances progress;
(2) a general mechanism of process recursion instead of the limited permanent accepts;
(3) a more liberal treatment of the channels which can be sent; and
(4) the standard semantics for the reception of channels with roles, which permits to get rid of process sequencing.

None of the previous work had treated progress across interfered, dynamically interleaved multiparty sessions.

Acknowledgements. We thank Kohei Honda and the Concur reviewers for their comments on an early version of this paper and Gary Brown for his collaboration on an implementation of multiparty session types.

References

1. Acciai, L., Boreale, M.: A Type System for Client Progress in a Service-Oriented Calculus. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 642–658. Springer, Heidelberg (2008)
2. Bonelli, E., Compagnoni, A.: Multipoint Session Types for a Distributed Calculus. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 240–256. Springer, Heidelberg (2008)
3. Bravetti, M., Zavattaro, G.: Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
4. Bruni, R., Lanese, I., Melgratti, H., Tuosto, E.: Multiparty Sessions in SOC. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 67–82. Springer, Heidelberg (2008)
5. Bruni, R., Mezzina, L.G.: A Deadlock Free Type System for a Calculus of Services and Sessions (2008), `http://www.di.unipi.it/~bruni/publications/ListOfDrafts.html`
6. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous Session Types and Progress for Object-Oriented Languages. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 1–31. Springer, Heidelberg (2007)
7. Dezani-Ciancaglini, M., de Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)
8. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)

9. Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. Acta Informatica 42(2/3), 191–225 (2005)
10. Honda, K.: Types for Dyadic Interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
11. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 22–138. Springer, Heidelberg (1998)
12. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM, New York (2008)
13. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142. Springer, Heidelberg (2008)
14. Kobayashi, N.: A Partially Deadlock-Free Typed Process Calculus. ACM TOPLAS 20(2), 436–482 (1998)
15. Kobayashi, N.: A New Type System for Deadlock-Free Processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
16. Lanese, I., Vasconcelos, V.T., Martins, F., Ravara, A.: Disciplining Orchestration and Conversation in Service-Oriented Computing. In: SEFM 2007, pp. 305–314. IEEE Computer Society Press, Los Alamitos (2007)
17. Milner, R.: Communicating and Mobile Systems: the π-Calculus. CUP (1999)
18. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
19. Scribble Project, `http://www.scribble.org`
20. UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme (2002), `http://www.iso20022.org`
21. Web Services Choreography Working Group. Web Services Choreography Description Language, `http://www.w3.org/2002/ws/chor/`
22. Yoshida, N.: Graph Types for Monadic Mobile Processes. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 371–386. Springer, Heidelberg (1996)
23. Yoshida, N., Vasconcelos, V.T.: Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In: SecRet 2006. ENTCS, vol. 171, pp. 73–93. Elsevier, Amsterdam (2007)