A Reputation System for Multirole Sessions ⋆

Viviana Bono[1], Sara Capecchi[1], Ilaria Castellani[2], and
Mariangiola Dezani-Ciancaglini[1]

[1] Dipartimento di Informatica, Università di Torino, corso Svizzera 185, 10149 Torino, Italy
[2] INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis, France

Abstract. We extend role-based multiparty sessions with reputations and policies associated with principals. The reputation associated with a principal in a service is built by collecting her relevant behaviour as a participant in sessions of the service. The service checks the reputation of principals before allowing them to take part in a session, also according to the role they want to play. Furthermore, principals can declare policies that must be fulfilled by the other participants of the same service. These policies are used by principals to check the reputation of the current participants and to decide whether or not to join the service. We illustrate the use of our approach with an example describing a real-world protocol.

Keywords: concurrency, communication-centred computing, session calculi, session types, reputation systems.

1 Introduction

Building on [8] and [10], where flexible role-based multiparty sessions were introduced, we address the question of accomodating dynamic interaction policies in sessions with multiple roles and a varying number of participants for each role, taking into account the histories of principals. The *history* of a principal is a trace of its past interactions with other principals within service sessions. A *service* is an abstraction for a multiparty interaction point, where partners play predefined roles and behave according to a precise communication protocol. A *session* is an activation of a service. Histories are used to build principals' *reputations*. For each service, only the principal's history relative to that service is significant for her reputation in the service. This reputation is checked against the service access policy, before that principal is admitted in a new session of the service with a given role. It is also checked by other potential participants before they engage in a session involving that principal.

Our aim is to provide an enriched role-based session calculus able to deal with principals' reputations, together with a type system ensuring that classical session properties, such as communication safety and progress, continue to hold in the presence of these new features. While borrowing most constructs and typing rules from [8], and notably the *polling* operator that allows a principal to concurrently interact with all principals playing a given role in an ongoing session, our calculus departs from [8] in

⋆ Work partially funded by the ANR-08-EMER-010 grant PARTOUT, and by the MIUR Projects DISCO and IPODS.

that it distinguishes between the notion of *service* and that of *session*, allowing multiple sessions for a single service. We believe this may help modelling real-world scenarios. Think, for example, of an online shop, where there are principals who play the role of sellers and principals who play the role of buyers (notice that each principal may play both roles). The online shop is a service with two roles, "seller" and "buyer". These two roles are of different nature. Sellers should in principle always be available for a transaction with a buyer. Therefore they join the service in a stable way, in order to be present for several successive sessions. We call this a *stable join*. Buyers, instead, might want to join a service only for a single session, to purchase a specific item from some seller. We call this a *one-shot join*. To fix ideas, one may view the initialisation of the service as the start-up of the online shop, the stable join by a seller as the opening of her activity in the online shop, the one-shot join by a buyer as her connection to the online-shop site, the initialisation of a session as the start of an interaction among the sellers and buyers which are currently present in the online shop.

Other points of departure of our calculus with respect to [8] have to do with the introduction of histories. Thus, some constructs of [8] had to be refined in order to account for histories and reputations. For example, in our calculus there are two kinds of sending constructs. The first, denoted !, is the standard message send, while the second, denoted $!^{\bullet}$, represents a *relevant* message send, whose content must be stored in the history of the sender. Moreover, we offer a *choice* primitive for selecting one principal among the best ones for a given role (according to a given policy), if any.

Histories are exploited at various stages of the interaction:

- at service join, to allow the service to select the principals who will take part in future sessions and to allow a principal wishing to join the service to evaluate the reputation of the current participants and proceed or not with the join accordingly;
- at session initiation, to allow the service to select among the stable participants those who will take part in the session, by testing if they satisfy some condition;
- in a poll operation, to allow a participant to interact only with participants which satisfy some condition;
- in a choice operation, to allow a participant to select one of the best participants according to some criterion.

As regards the type system, the main novelties with respect to [8] are the addition of an existential quantification for the choice operator, and a variation of the universal quantification for the polling operator, as in our case polling does not collect all principals in a given role, but only those verifying some condition based on their reputations.

An interesting feature of our calculus is its ability to regulate a principals' participation in a service according to her reputation: if a principal behaves "badly" as a participant in some service, this may result in a restriction of the possible session roles offered by the service to that principal.

The paper is organized as follows. In Section 2 we introduce our approach by a motivating example. In Sections 3 and 4 we present the syntax and the semantics of our calculus. Section 5 defines the type system. Section 6 establishes the properties of our calculus. In Section 7 we draw some conclusions and discuss related work.

2 Example

Let us take a closer look at the online shop example introduced in Section 1. At any time, a buyer can choose among the current sellers according to some criterion, in order to purchase a specific item. For instance, the buyer may ask for sellers who are fast in delivering their products. This selection can be done by inspecting the past behaviour of sellers, that is, their history, assuming that the shop records all histories in a dedicated registry. The selected seller then sends to the buyer the price of the item. The buyer sends back either a positive or a negative answer, represented by the labels `OK` or `KO`, which become part of the buyer history. In case of a positive answer, the seller sends to the buyer the delivery time, which is recorded in the history of the seller and is liable to be tested in future interactions. In case of a negative answer, the transaction aborts.

In this scenario, there can be an arbitrary number of buyers and sellers, joining the online shop dynamically (this join is subject to an acceptance condition imposed by the service, and depending on the histories of the applicants). After a number of buyers and sellers have joined the service, a shopping session can start, in which all the present buyers and sellers may interact concurrently. Therefore, the online shop may be seen as the concurrent execution of several buyer-seller conversations, each of which is abstractly specified by the *global type G* of Figure 1. The global type G starts with a universal quantification over buyers, followed by an existential quantification over sellers satisfying the criterion `fast`. The former represents the spawning of a separate interaction for each buyer, while the latter represents the choice of one of the best sellers according to the criterion `fast`, i.e. one of the fastest sellers (for this choice to be possible, we assume that for each criterion, there exists an ordering on reputations parametrised on this criterion). Hence, interactions only occur between a buyer and the selected fastest seller, and each of these interactions proceeds in parallel with the others.

For simplicity, we only consider send actions to be relevant for the reputation here. One may argue that this restriction is reasonable since the value of a message is produced by the sender and the receiver does not have any control over it. Note however that, in some practical cases, a received value could be relevant for the reputation of the receiver (for instance, a notification from a bank could only be sent to trusted clients).

Buyers and sellers can participate in the online shop service in different ways. The participation can be one-shot: usually buyers join the shop when they want to buy some item and leave it when they have completed their purchase. By contrast, sellers are likely to have a more stable presence. As long as they want to sell their products, they are part of the service and they replicate their behaviour for each shopping session. However, nothing prevents the shop to include stable buyers and one-shot sellers. In other words, stability is a property of the join operation, not of the roles themselves.
Possible processes describing the buyer and the seller are given in Figure 2.

Process B describes a principal `b1` who wants to join just one session of service a, playing the buyer role (`b1` : *buyer*). Once the session starts, `b1` asks (using channel y)

$$
\begin{aligned}
G = \forall\, \iota : \mathit{buyer}.\exists \iota' : \mathtt{fast}(\mathit{seller}).\ & \iota \to \iota'\langle \mathtt{Item} \rangle; \\
& \iota' \to \iota\langle \mathtt{Price} \rangle; \\
& \iota \to^{\bullet} \iota'\{\mathtt{OK}.\iota' \to^{\bullet} \iota\, \langle \mathtt{Deliver} \rangle; \mathrm{end}, \\
& \qquad \mathtt{KO}.\mathrm{end}\}
\end{aligned}
$$

Fig. 1. Global type for buyer-seller interaction

$$\mathsf{B} = a[\mathtt{b1} : \mathit{buyer}](y).\ \{\ y\exists(\iota : \mathtt{fast}(\mathit{seller})).\ \{y!\langle \iota, \mathtt{item}\rangle; y?\langle \iota, (x).\\ \mathtt{if}\ \mathtt{OK}(x)\ \mathtt{then}\ y!^{\bullet}\langle \iota, \mathtt{OK}\rangle; y?\langle \iota, (x_1).\mathbf{0}\rangle\\ \mathtt{else}\ y!^{\bullet}\langle \iota, \mathtt{KO}\rangle; \mathbf{0}\rangle\}\ \}$$

$$\mathsf{S} = \overline{a}[\mathtt{s1} : \mathit{seller}](y).\ \{\ y\forall(\iota : \mathit{buyer}).\{y?\langle \iota, (x).y!\langle \iota, \mathtt{price}\rangle;\\ y?\langle \iota, \{\mathtt{OK}.y!^{\bullet}\langle \iota, \mathtt{deliver}\rangle; \mathbf{0},\\ \mathtt{KO}.\mathbf{0}\}\rangle\rangle\}\ \}$$

Fig. 2. Buyer and seller processes

for one of the fastest sellers. This is done via the choice construct $y\exists(\iota : \mathtt{fast}(\mathit{seller}))$, which chooses among the participants in a given role one of the best according to a particular criterion. The selection over sellers is performed by inspecting their histories (that is, their delivery times), since a records them in a dedicated registry. After the selection, the buyer sends a request for a specific item and waits for the price of the item from the seller. Then, according to the price, she will answer either `OK` or `KO`: in the first case, she expects one further input (the delivery time), while in the latter the conversation ends immediately.

Process S describes a principal `s1` who wants to join the service a in a stable way playing the role of a seller ($\mathtt{s1} : \mathit{seller}$) (the act of joining a service a in a stable way is expressed by $\overline{a}$, to distinguish it from the act of joining a only for one session). Once the seller has joined the service, she waits for the request of an item from all the present buyers. This is realised by the poll constructor $y\forall(\iota : \mathit{buyer})$, which spawns in parallel a copy of the seller for each buyer, where the variable ι is replaced by the buyer identity.

Note that the send construct $!^{\bullet}$ corresponds to the arrow $\rightarrow^{\bullet}$ in the global type and represents a relevant send whose content must be stored in the history of the sender.

This example can be extended by adding two roles: *goldBuyer* and *goldSeller*. Gold buyers can decide whether to buy an item or to ask for assistance. Gold sellers offer additional assistance and sell the same items with an extra cost to cover assistance. Gold sellers may want to interact only with gold buyers who tend to respond positively (i.e., who accepted the proposed price most of the time in their previous interactions), qualified as keen gold buyers. Gold buyers must select either the label `BUY` or the label `AST` to indicate the kind of interaction requested. If a gold buyer wants assistance, she selects one of the fastest available gold seller and sends her an `AST` label to ask for help, then specifies her problem and waits for the response. Figure 3 gives the global type G' for gold buyers and gold sellers, starting with a universal quantification on buyers satisfying the condition `keen`, and Figure 4 shows the incarnations of principals $\mathtt{b}_1$ and $\mathtt{s}_1$ as gold buyer and gold seller, respectively.

$$G' = \forall\, \iota : \mathtt{keen}(\mathit{goldBuyer}, \iota).\exists \iota' : \mathtt{fast}(\mathit{goldSeller}).\iota \rightarrow \iota'\{\ \mathtt{BUY}.\ \iota \rightarrow \iota'\langle \mathtt{Item}\rangle;\\ \iota' \rightarrow \iota\langle \mathtt{Price}\rangle;\\ \iota \rightarrow^{\bullet} \iota'\{\mathtt{OK}.\\ \iota' \rightarrow^{\bullet} \iota\,\langle \mathtt{Deliver}\rangle\,;\mathrm{end},\\ \mathtt{KO}.\mathrm{end}\}\\ \mathtt{AST}.\ \iota \rightarrow \iota'\langle \mathtt{Problem}\rangle;\\ \iota' \rightarrow \iota\langle \mathtt{Solution}\rangle;\mathrm{end}\}$$

Fig. 3. Global type for gold buyer and gold seller interaction

$$\mathsf{GB} = a[\mathtt{b1} : \mathit{goldBuyer}](y).\ \{\, y\exists(\iota : \mathtt{fast}(\mathit{goldSeller})).$$
$$\{\, \mathtt{if\ alright\ then}\ y!\langle \iota, \mathtt{BUY}\rangle; y!\langle \iota, \mathtt{item}\rangle; y?\langle \iota, (x).$$
$$\mathtt{if\ OK}(x)\ \mathtt{then}\ y!^{\bullet}\langle \iota, \mathtt{OK}\rangle; y?\langle \iota, (x_1).\mathbf{0}\rangle$$
$$\mathtt{else}\ y!^{\bullet}\langle \iota, \mathtt{KO}\rangle; \mathbf{0}\rangle$$
$$\mathtt{else}\ y!\langle \iota, \mathtt{AST}\rangle; y!\langle \iota, \mathtt{problem}\rangle; y?\langle \iota, (x').\mathbf{0}\rangle \}\ \}$$

$$\mathsf{GS} = \overline{a}[\mathtt{s1} : \mathit{goldSeller}](y).\ \{\, y\forall(\iota : \mathtt{keen}(\mathit{goldBuyer}, \iota)).$$
$$\{ y?\langle \iota, \{\mathtt{BUY}.\ y?\langle \iota, (x).y!\langle \iota, \mathtt{price}\rangle;$$
$$y?\langle \iota, \{\mathtt{OK}.y!^{\bullet}\langle \iota, \mathtt{deliver}\rangle; \mathbf{0},$$
$$\mathtt{KO}.\mathbf{0}\}\rangle\rangle$$
$$\mathtt{AST}.y?\langle \iota, (x').y!\langle \iota, \mathtt{solution}\rangle; \mathbf{0}\rangle\}\rangle\}\ \}$$

Fig. 4. Gold buyer and gold seller processes

Consider the following process, where the components B, S, GB, and GS are defined in Figures 2 and 4:

$$a\langle G \mid G', \mathtt{rel}\rangle \mid \mathsf{B} \mid \mathsf{S} \mid \mathsf{GB} \mid \mathsf{GS}$$

Here process $a\langle G \mid G', \mathtt{rel}\rangle$ represents a service with name a (the name of the online shop), global type $G \mid G'$ and join condition `rel`. The global types G and G' are given in Figures 1 and 3, respectively, and the condition `rel` (standing for "reliable") expresses different requirements for the join of a principal in the different roles:

- any principal can join a session of the service as an ordinary buyer or seller;
- only principals which have a long enough record of successful transactions can join sessions as gold buyers and gold sellers.

We notice that even if we collect in the histories only `OK`/`KO` answers for buyers and delivery times for sellers, this information is enough to check interesting properties. Indeed, histories are evaluated in different ways by the conditions `keen` and `fast` expressed by participants and by the condition `rel` associated with service a:

- `rel` checks the number of successful transactions of gold buyers, while `keen` checks the percentage of successful transactions over the total number [3];
- `rel` checks the total number of transactions completed by a gold seller (that is, the number of values in her history, no matter whether single values are good or bad), while `fast` checks the average time of delivery.

At runtime, the initialisation of a service will create a dedicated registry for recording the relevant behaviour of principals, thus building up their history in the service.

In the first session, principals $\mathtt{b}_1$ and $\mathtt{s}_1$ can only play the roles of buyer and seller, respectively, so principal $\mathtt{b}_1$ can only buy, without requesting assistance. After some successful sessions $\mathtt{b}_1$ and $\mathtt{s}_1$ will be able to play the roles of gold buyer and gold seller too. Then, principal $\mathtt{b}_1$ will have the possibility to ask for assistance.

After being promoted to gold buyer, a principal must keep up her reputation: only gold buyers who continue to satisfy the `keen` condition are allowed to interact with gold sellers. After a "bad" behaviour, in order to play again the role of a gold buyer, a principal must rebuild herself a "good" reputation as a buyer.

The above system only includes one buyer-seller pair, and one gold buyer-seller pair. In a more realistic scenario, there would be several participants for each role, differing from the processes B, S, GB and GS only for the id (`b1`, `s1`) and the exchanged values.

[3] Note that it is possible to compute the percentage of failed transactions only for buyers, and not for sellers (because the label `OK`/`KO` is sent by the buyer, and only received by the seller).

3 Syntax

We assume the following sets: *value variables*, ranged over by $x, y, z \ldots$, *service names*, ranged over by $a, b, \ldots$, *principals*, ranged over by $\mathtt{id}$, $\mathtt{id}'$, ..., *principal variables*, ranged over by ι, $\iota', \ldots$, *roles*, ranged over by $r, r', \ldots$, *sessions*, ranged over by $s, s', \ldots$, and *labels*, ranged over by $l, l', \ldots$.

The syntax of processes is given in Table 1. It uses the auxiliary definitions of Table 2 and the types of Table 3. The syntax occurring only at runtime appears shaded.

As hinted in the previous section, participants can either join a service in a *stable* modality and be present in all the sessions of the service (which start after they join and end before they leave), or join a service for exactly one session in a *one-shot* modality.

A new service is always opened by an initialiser of the form $a\langle G, \phi \rangle$, where G is a global type and ϕ is a mapping from histories and roles to truth values, representing the condition a principal must satisfy in order to be accepted as a participant with a given role in service a. The initialisation of service a creates a service registry $a[\mathscr{H}, \mathscr{O}_1, \mathscr{O}_2, \phi]$, where the *history set* $\mathscr{H}$ records the current mapping between principals and their histories, $\mathscr{O}_1$ and $\mathscr{O}_2$ are (initially empty) parallel compositions of *offers*, representing the participants who joined the service in a stable or one-shot modality, respectively.

Once the service a has been initialised, principal $\mathtt{id}$ can join the service in role r using $\overline{a}[\mathtt{id} : r, \mathscr{C}(\tilde{r})](x).\{P\}$ or $a[\mathtt{id} : r, \mathscr{C}(\tilde{r})](x).\{P\}$, becoming respectively a stable or a one-shot offer. The join is only allowed if the principal $\mathtt{id}$ does not already appear with the same role among the current offers, and if the histories associated with the other principals present in the service satisfy the set of conditions $\mathscr{C}(\tilde{r})$ expected by $\mathtt{id}$, what we call the *policy* of $\mathtt{id}$. Moreover, the history of principal $\mathtt{id}$ in the service registry must satisfy the acceptance condition ϕ for the required role (in order to get started and allow fresh participants, some conditions must be satisfied by the empty history). Indeed, when a principal joins a service, her history is not necessarily empty, since she could have already joined and quit the service before. For stable join, the acceptance condition will be checked also at the start of each session. We call P the *body of the join*.

The join is implemented by registering the participant in the session registry as the offer $[\mathtt{id} : r](x).P$. The service a can be abandoned by the stable participant p by $\mathtt{quit}(a, p)$.

$P ::=$	
$a\langle G, \phi \rangle$	Service Init
$\overline{u}[\mathtt{id} : r, \mathscr{C}(\tilde{r})](x).\{P\}$	Stable Join
$u[\mathtt{id} : r, \mathscr{C}(\tilde{r})](x).\{P\}$	OneShot Join
$\mathtt{quit}(u, p)$	Service Quit
$\mathtt{quit}\langle c \rangle$	Session Quit
$c!^*\langle p, l\langle \mathscr{I} \rangle \langle e \rangle \rangle$	Send
$c?\langle p, \{l_i\langle \mathscr{I}_i \rangle (x_i).P_i\}_{i \in I} \rangle$	Receive
$c\forall(\iota \notin \mathscr{I} : \mathtt{C}(r, \iota)).\{P\}$	Poll
$c\exists(\iota : \mathtt{B}(r)).\{P\}$	Choice
$P \mid P$	Parallel
$P;P$	Sequential
$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P$	Conditional
$\mathbf{0}$	Nil
X	Recursion variable
$\mu X.P$	Recursion
$(\nu a : G)\ P$	Service restriction
$(\nu s)P$	Session restriction
$s : \mathscr{B}$	Message buffer
$a[\mathscr{H}, \mathscr{O}, \mathscr{O}, \phi]$	Service registry
$a\langle s, \mathscr{P} \rangle$	Session registry

Table 1. Processes

u	::= $x \mid a \mid b \mid \ldots$	Service Id.	h	::= $() \mid h\cdot(l\langle\mathscr{I}\rangle\langle v\rangle, r)$	History
p	::= $\mathtt{id}:r \mid \iota:r$	Participant	$\mathscr{H}$	::= $\emptyset \mid \mathscr{H}\cup(\mathtt{id},h)$	History Set
$\mathscr{I}$	::= $\emptyset \mid \mathscr{I}\cup\mathtt{id} \mid \mathscr{I}\cup\iota$	Princ. Set	$\mathscr{P}$	::= $\emptyset \mid \mathscr{P}\cup(\mathtt{id}:r)$	Part. Set
c	::= $x \mid s[p]$	Channel			
v	::= $\mathtt{true} \mid \ldots \mid a \mid s[\mathtt{id}:r]$	Value	**Conditions**		
e	::= $x \mid v \mid e\wedge e \mid \ldots$	Expression	$\mathtt{C}(\rho,r,\iota)$	::= $\phi(\rho\restriction\iota,\, r)$	Single
m	::= $(\mathtt{id}:r, \mathtt{id}':r', l\langle\mathscr{I}\rangle\langle v\rangle)$	Message	$\mathtt{C}(r,\iota)$	::= $\lambda\rho.\, \mathtt{C}(\rho,r,\iota)$	Poll
$\mathscr{B}$	::= $[] \mid \mathscr{B}\cdot m$	Buffer	$\mathscr{C}(\rho,\tilde{r})$	::=	Basic
$\mathscr{O}$	::=	Offer Set		$\forall\iota.\, \mathtt{C}(\rho,r,\iota) \mid$	$r\in\tilde{r}$
	$\mid \mathbf{0}$	Nil		$\exists\iota.\, \mathtt{C}(\rho,r,\iota) \mid$	$r\in\tilde{r}$
	$\mid [\mathtt{id}:r](x).P$	Offer		$\mathscr{C}_1(\rho,\tilde{r})\wedge\mathscr{C}_2(\rho,\tilde{r}) \mid$	
	$\mid \mathscr{O} \mid \mathscr{O}$	Parallel		$\mathscr{C}_1(\rho,\tilde{r})\vee\mathscr{C}_2(\rho,\tilde{r})$	
			$\mathscr{C}(\tilde{r})$	::= $\lambda\rho.\, \mathscr{C}(\rho,\tilde{r})$	Join

Table 2. Auxiliary definitions

A service registry $a[\mathscr{H},\mathscr{O}_1,\mathscr{O}_2,\phi]$ can initiate a session by creating a new session name s and a session registry $a\langle s,\mathscr{P}\rangle$, where $\mathscr{P}$ records the session participants. The offers in $\mathscr{O}_1$ whose histories satisfy the predicate ϕ for the required role (evaluating each offer's reputation) and all the offers in $\mathscr{O}_2$ will join that session. A session represents a particular instance or activation of a service. Session initiation replaces the variable x in the body of each offer $[\mathtt{id}:r](x).P$ with the corresponding session channel. Session channels are temporary channels created at the start of a session, and their lifetime is that of the session. Each participant has just one session channel and this is her only means for interacting with the other participants within a session.

The output process $c!^*\langle p, l\langle\mathscr{I}\rangle\langle e\rangle\rangle$ sends to p on channel c the value of expression e labelled by the constant l and the set of principals and principal variables $\mathscr{I}$. The symbol $!^*$ stands for two different kinds of send, $!$ and $!^\bullet$: $!$ is used for standard send, while $!^\bullet$ is used to send a message which will be registered in the history of the sender within the service register. The input process $c?\langle p, \{l_i\langle\mathscr{I}_i\rangle(x_i).P_i\}_{i\in I}\rangle$ expects from p on channel c a message with a label l in $\{l_i\}_{i\in I}$ and a set of principals and principal variables $\mathscr{I}$ in $\{\mathscr{I}_i\}_{i\in I}$. If $l=l_i$ and $\mathscr{I}=\mathscr{I}_i$, the value of the message will be replaced for the variable x_i, which is bound in P_i.

Polling $c\forall(\iota\notin\mathscr{I} : \mathtt{C}(r,\iota)).\{P\}$ allows interaction between c and all the principals ι not belonging to $\mathscr{I}$ that instantiate the role r and whose history satisfies the condition $\mathtt{C}(r,\iota)$. Process P (the *body of the poll*) is replicated for each such participant.

G	::=	Global Types	T	::=	Local Types
	$\mid p\to^* p\{l_i\langle U_i\rangle.G_i\}_{i\in I}$	Label. Mess.		$\mid !^*\langle p, \{l_i\langle\mathscr{I}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle$	Selection
	$\mid \forall\,\iota\notin\mathscr{I} : \mathtt{C}(r,\iota).G$	Univ. Quant.		$\mid ?\langle p, \{l_i\langle\mathscr{I}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle$	Branching
	$\mid \exists\,\iota : \mathtt{B}(r).G$	Exist. Quant.		$\mid \forall\,\iota\notin\mathscr{I} : \mathtt{C}(r,\iota).T$	Univ. Quant.
	$\mid G \mid G \mid G\ ;\ G$	Paral., Seq.		$\mid \exists\,\iota : \mathtt{B}(r).T$	Exist. Quant.
	$\mid \mu\mathbf{x}.G \mid \mathbf{x}$	Rec., Var.		$\mid T \mid T \mid T\ ;\ T$	Paral., Seq.
	$\mid \varepsilon \mid \mathrm{end}$	Inact., End		$\mid \mu\mathbf{x}.T \mid \mathbf{x}$	Rec., Var.
U	::= $S \mid T$	Message Types		$\mid \varepsilon \mid \mathrm{end}$	Inact., End
			S	::= $\langle G\rangle \mid \mathrm{bool} \mid \mathrm{string} \mid \ldots$	Sorts

Table 3. Global and local types

Choice $c\exists(\iota : \mathtt{B}(r)).\{P\}$ returns P where ι is replaced by one of the best principals with respect to criterion $\mathtt{B}$ among those playing role r in the session, that is, one of those enjoying the best reputation with respect to $\mathtt{B}$, if any. For instance, if r is *seller* and $\mathtt{B}(r)$ is $\mathtt{fast}(\mathit{seller})$, the choice $c\exists(\iota : \mathtt{B}(r)).\{P\}$ returns P where ι is replaced by one of the fastest sellers, unless there is no seller. This presupposes that for each role r of a service, and for each criterion $\mathtt{B}$ applicable to r, there exists a partial order $\sqsubseteq_{\mathtt{B}}$ on histories, parametrised on $\mathtt{B}$. For instance, $h \sqsubseteq_{\mathtt{fast}} h'$ means that the average delivery time recorded in h is less than or equal to the average delivery time recorded in h'.

Messages have the form $(\mathtt{id} : r, \mathtt{id}' : r', l\langle\mathscr{I}\rangle\langle v\rangle)$, including sender, receiver, label, set of transmitted participants and value as in [8]. Messages exchanged (asynchronously) in session s are stored in the message buffer $s{:}\mathscr{B}$.

Parallel and sequential composition, conditional and recursion are standard. The restriction $(\nu a : G)P$ creates a shared service name that can be used as a reference for a service specified by G, while $(\nu s)P$ represents a new session instance.

Histories h are built by recording some of the labels and values that are sent by the principals, together with the roles the principals belong to. A history set $\mathscr{H}$ is a set of pairs $(\mathtt{id}, h)$ associating a history with a principal $\mathtt{id}$. Histories are used to measure principals' reputation, that is, to check if principals satisfy the conditions and the criteria expressed in join, poll and choice operations. We project history sets on principals as follows:

$$\mathscr{H}\restriction\mathtt{id} = \begin{cases} h & \text{if } (\mathtt{id}, h) \in \mathscr{H}, \\ (\,) & \text{otherwise} \end{cases}$$

Therefore, if ρ is a history set variable, $\phi(\rho\restriction\iota, r)$ expresses a condition (*single* condition) on the history of principal ι in the history set ρ relative to the role r. We denote by $\mathtt{C}(\rho,r,\iota)$ a single condition and by $\mathtt{C}(r,\iota)$ the abstraction with respect to ρ of a single condition. We use $\mathscr{C}(\rho,\tilde{r})$ for conditions (*basic* conditions) obtained from single conditions by universal or existential quantification on principal variables and by closure under conjunction and disjunction. Lastly, $\mathscr{C}(\tilde{r})$ is the abstraction of a basic condition with respect to ρ. For example, $\mathscr{C}(r_1, r_2)$ could be $\lambda\rho.\exists\iota_1.\phi_1(\rho\restriction\iota_1, r_1) \wedge \forall\iota_2.\phi_2(\rho\restriction\iota_2, r_2)$. For processes we adopt the following simplifications, already used in the example of Section 2: we omit empty sets of principals, we omit labels if there is a unique branch (i.e., we write $c!^*\langle p, v\rangle$, $c?\langle p, (x).P\rangle$), we omit empty values (i.e., we write $c!^*\langle p, l\rangle$, $c?\langle p, \{l_i.P_i\}_{i\in I}\rangle$), we use $\iota{:}r$ as short for $\iota{:}\mathtt{C}(r,\iota)$ when $\mathtt{C}(r,\iota)$ holds always true, and we omit roles for quantified principal variables (i.e., we write ι in the body of a quantification on $\iota{:}\mathtt{C}(r,\iota)$ or $\iota{:}\mathtt{B}(r)$). The writing of types is simplified in a similar way.

A process is *initial* if it does not contain free variables and runtime syntax.

To sum up, the syntax of our calculus differs from that of [8] for the following features:

1. we distinguish services and sessions, allowing multiple sessions for a single service;
2. we associate histories with principals participating in services;
3. we associate acceptance conditions with services, allowing them to filter out "bad" principals;
4. we associate policies with principals, allowing them to join a service only if the reputation of the other principals already present in the service satisfies the policies;

5. we add conditions to quantifications, allowing a participant to interact only with selected partners;
6. we offer a choice primitive, allowing a participant to choose one of the best principals (according to some specified criterion) among those playing a given role in a session.

4 Semantics

As usual, the operational semantics is defined modulo a structural equivalence $\equiv$. We assume the standard structural rules for processes [16]. Among the rules for buffers, we have one for swapping independent messages, i.e., messages with different sender or receiver. Moreover, the following rule

$$(\nu s)(a\langle s, \mathscr{P}\rangle \mid s : [\,]) \equiv \mathbf{0}$$

is useful to garbage collect ended sessions.
The reduction rules are given in Tables 4, 5, 6 and 7. We briefly comment on them.

Rule [ServiceInit] initialises a service by reducing $a\langle G, \phi\rangle$. It creates a *permanent* session registry $a[\emptyset, \mathbf{0}, \mathbf{0}, \phi]$ where the history set is empty, and two (initially null) groups of offers are created: the first is the parallel composition of all stable offers, that is, offers by participants who will be present in all sessions of the service, unless they behave "badly" or decide to leave the service. The second is the parallel composition of all one-shot offers, that is, offers by participants who will join one session only of the service and then leave it.

[ServiceInit]
$a\langle G, \phi\rangle \longrightarrow a[\emptyset, \mathbf{0}, \mathbf{0}, \phi]$

[StableJoin]
$\overline{a}[\mathtt{id} : r, \mathscr{C}(\tilde{r})](y).\{P\} \mid a[\mathscr{H}, \mathscr{O}_1, \mathscr{O}_2, \phi] \longrightarrow a[\mathscr{H}\flat\mathtt{id}, \mathscr{O}_1 \mid [\mathtt{id} : r](y).P, \mathscr{O}_2, \phi]$
if $\mathscr{C}(\tilde{r})\mathscr{H}$ and $(\mathtt{id} : r) \notin \mathscr{O}_1 \mid \mathscr{O}_2$ and $\phi(\mathscr{H}\restriction\mathtt{id}, r)$

[OneShotJoin]
$a[\mathtt{id} : r, \mathscr{C}(\tilde{r})](y).\{P\} \mid a[\mathscr{H}, \mathscr{O}_1, \mathscr{O}_2, \phi] \longrightarrow a[\mathscr{H}\flat\mathtt{id}, \mathscr{O}_1, \mathscr{O}_2 \mid [\mathtt{id} : r](y).P, \phi]$
if $\mathscr{C}(\tilde{r})\mathscr{H}$ and $(\mathtt{id} : r) \notin \mathscr{O}_1 \mid \mathscr{O}_2$ and $\phi(\mathscr{H}\restriction\mathtt{id}, r)$

[SessionInit]
$a[\mathscr{H}, \Pi_{i\in I}[\mathtt{id}_i : r_i](y_i).P_i \mid \Pi_{j\in J}[\mathtt{id}_j : r_j](y_j).P_j, \Pi_{k\in K}[\mathtt{id}_k : r_k](y_k).P_k, \phi] \longrightarrow$
$(\nu s)(a\langle s, \{\mathtt{id}_i : r_i \mid i \in I \cup K\}\rangle \mid \Pi_{i\in I\cup K} P_i\{s[\mathtt{id}_i : r_i]/y_i\} \mid s : [\,]) \mid$
$a[\mathscr{H}, \Pi_{i\in I}[\mathtt{id}_i : r_i](y_i).P_i \mid \Pi_{j\in J}[\mathtt{id}_j : r_j](y_j).P_j, \mathbf{0}, \phi]$
if $\forall i \in I.\phi(\mathscr{H}\restriction\mathtt{id}_i, r_i)$ and $\forall j \in J.\neg\phi(\mathscr{H}\restriction\mathtt{id}_j, r_j)$

[ServiceQuit]
$\mathtt{quit}(a, \mathtt{id} : r) \mid a[\mathscr{H}, \mathscr{O}_1, \mathscr{O}_2, \phi] \longrightarrow a[\mathscr{H}, \mathscr{O}_1 \backslash (\mathtt{id} : r), \mathscr{O}_2, \phi]$

[SessionQuit]
$\mathtt{quit}\langle s[\mathtt{id} : r]\rangle \mid a\langle s, \mathscr{P} \cup (\mathtt{id} : r)\rangle \longrightarrow a\langle s, \mathscr{P}\rangle$

Table 4. Reduction rules I

Rules [StableJoin] and [OneShotJoin] perform the registration of a participant as an offer associated with a service, in a stable or one-shot way, respectively. The appli-

[Send]
$s[\mathtt{id}:r]!\langle \mathtt{id}':r', l\langle \mathscr{I}\rangle\langle v\rangle\rangle \mid s:\mathscr{B} \longrightarrow s:\mathscr{B}\cdot(\mathtt{id}:r, \mathtt{id}':r', l\langle \mathscr{I}\rangle\langle v\rangle)$

[SendR]
$$s[\mathtt{id}:r]!^{\bullet}\langle \mathtt{id}':r', l\langle \mathscr{I}\rangle\langle v\rangle\rangle \mid a[\mathscr{H}\cup(\mathtt{id},h), \mathscr{O}_1, \mathscr{O}_2, \phi] \mid s:\mathscr{B} \longrightarrow a[\mathscr{H}\cup(\mathtt{id}, h\cdot(l\langle \mathscr{I}\rangle\langle v\rangle, r)), \mathscr{O}_1, \mathscr{O}_2, \phi] \mid s:\mathscr{B}\cdot(\mathtt{id}:r, \mathtt{id}':r', l\langle \mathscr{I}\rangle\langle v\rangle)$$

[Receive]
$s[\mathtt{id}:r]?\langle \mathtt{id}':r', \{l_i\langle \mathscr{I}_i\rangle(x_i).P_i\}_{i\in I}\rangle \mid s:(\mathtt{id}':r', \mathtt{id}:r, l_k\langle \mathscr{I}_k\rangle\langle v\rangle)\cdot\mathscr{B} \longrightarrow P_k\{v/x_k\} \mid s:\mathscr{B}$ where $k\in I$

Table 5. Reduction rules II

cant specifies: i) her identity id, ii) which role r she wants to play and iii) her policy, i.e. which conditions $\mathscr{C}(\tilde{r})$ must be satisfied by the histories of the principals that are already present. The join is successful if:

1. the histories associated with the principals already present in the service satisfy these conditions, checked by $\mathscr{C}(\tilde{r})\mathscr{H}$;
2. the participant is not already present as an offer, i.e., $(\mathtt{id}:r)\notin \mathscr{O}_1 \mid \mathscr{O}_2$, where we define:
$$(\mathtt{id}:r)\in\mathscr{O} \Leftrightarrow \mathscr{O}\equiv\mathscr{O}' \mid [\mathtt{id}:r](y).P \text{ for some } \mathscr{O}', P$$
3. the principal id has a history satisfying the predicate ϕ for role r, i.e., $\phi(\mathscr{H}\restriction\mathtt{id}, r)$ holds.

In the resulting service registry, id will have exactly one history since the update of a history set with a new principal is given by:

$$\mathscr{H}\flat\mathtt{id} = \begin{cases} \mathscr{H}\cup\{(\mathtt{id},(\,))\} & \text{if } \mathtt{id}\notin\mathscr{D}(\mathscr{H}),\\ \mathscr{H} & \text{otherwise}\end{cases}$$

where $\mathscr{D}(\mathscr{H}) = \{\mathtt{id} \mid (\mathtt{id},h)\in\mathscr{H}\}$.

Notice that, for preserving the order of communications, no channel occurring in the bodies of the joins should occur in the processes which follow the joins. This is assured by the typing rules for the join constructors (rules $\lfloor$STAJOIN$\rfloor$ and $\lfloor$OSJOIN$\rfloor$ in Table 10). For example using [OneShotJoin] and [Par] (see Table 7) we could get:

$$a[\mathtt{id}_1:r_3](y).\{\, s[\mathtt{id}_1:r_1]!\langle \mathtt{id}_2:r_2, \mathtt{true}\rangle\,\}; s[\mathtt{id}_1:r_1]!\langle \mathtt{id}_2:r_2, 5\rangle \mid a[\emptyset, \mathbf{0}, \mathbf{0}, \phi] \longrightarrow s[\mathtt{id}_1:r_1]!\langle \mathtt{id}_2:r_2, 5\rangle \mid a[\{(\mathtt{id}_1,(\,))\}, \mathbf{0}, [\mathtt{id}_1:r_3](y).s[\mathtt{id}_1:r_1]!\langle \mathtt{id}_2:r_2, \mathtt{true}\rangle, \phi]$$

In this way, the participant $\mathtt{id}_2:r_2$ could receive first 5 and then true from $\mathtt{id}_1:r_1$, instead of receiving first true and then 5, as expected.

In case of stable joins, we cannot allow free channels at all, as explained below when rule [SessionInit] is discussed.

Notice that our double join mechanism, with possibly multiple sessions associated with a single service, prevents new participants from intervening in the middle of an ongoing session. In this way, we avoid the need for a locking policy, as required in [8] to assure safe synchronisation.

Rule [SessionInit] initiates a session by reducing $a[\mathscr{H}, \mathscr{O}_1, \mathscr{O}_2, \phi]$. It creates a fresh session name s, a session registry $a\langle s, \mathscr{P}\rangle$ and an empty message buffer $\mathscr{B}$ named s, like the new session. The participant set $\mathscr{P}$ contains:

[Poll]

$$s[\mathtt{id}:r]\forall(\iota \notin \mathscr{I} : \mathtt{C}(r',\iota)).\{P\} \mid a\langle s,\mathscr{P}\rangle \mid a[\mathscr{H},\mathscr{O}_1,\mathscr{O}_2,\phi] \longrightarrow \begin{array}{l} \Pi_{i\in I} s[\mathtt{id}:r]!\langle \mathtt{id}_i : r', \mathtt{YES}\rangle \mid \\ s[\mathtt{id}:r]?\langle \mathtt{id}_i : r', \{\mathtt{YES}.P\{\mathtt{id}_i/\iota\}, \mathtt{NO}.\mathbf{0}\}\rangle \mid \\ \Pi_{j\in J} s[\mathtt{id}:r]!\langle \mathtt{id}_j : r', \mathtt{NO}\rangle \mid \\ s[\mathtt{id}:r]?\langle \mathtt{id}_j : r', \{\mathtt{YES}.\mathbf{0}, \mathtt{NO}.\mathbf{0}\}\rangle \mid \\ a\langle s,\mathscr{P}\rangle \mid a[\mathscr{H},\mathscr{O}_1,\mathscr{O}_2,\phi] \end{array}$$

where $\{\mathtt{id}_i \mid i \in I\} = \{\mathtt{id}' \mid (\mathtt{id}' : r') \in \mathscr{P} \wedge \mathtt{C}(r',\mathtt{id}')\mathscr{H} \wedge \mathtt{id}' \notin \mathscr{I}\}$
and $\{\mathtt{id}_j \mid j \in J\} = \{\mathtt{id}' \mid (\mathtt{id}' : r') \in \mathscr{P} \wedge (\neg\mathtt{C}(r',\mathtt{id}')\mathscr{H} \vee \mathtt{id}' \in \mathscr{I})\}$
if $\iota : r'$ occurs as subject in P

[PassivePoll]

$$s[\mathtt{id}:r]\forall(\iota \notin \mathscr{I} : r').\{P\} \mid a\langle s,\mathscr{P}\rangle \longrightarrow \Pi_{\mathtt{id}':r'\in\mathscr{P}\ \mathtt{id}'\notin\mathscr{I}} P\{\mathtt{id}'/\iota\}$$

if $\iota : r'$ occurs only as object in P

[Choice]

$$s[\mathtt{id}:r]\exists(\iota : \mathtt{B}(r')).\{P\} \mid a\langle s,\mathscr{P}\rangle \mid a[\mathscr{H},\mathscr{O}_1,\mathscr{O}_2,\phi] \longrightarrow \begin{array}{l} s[\mathtt{id}:r]!\langle \mathtt{id}' : r', \mathtt{YES}\rangle \mid \\ s[\mathtt{id}:r]?\langle \mathtt{id}' : r', \{\mathtt{YES}.P\{\mathtt{id}'/\iota\}, \mathtt{NO}.\mathbf{0}\}\rangle \mid \\ \Pi_{j\in J} s[\mathtt{id}:r]!\langle \mathtt{id}_j : r', \mathtt{NO}\rangle \mid \\ s[\mathtt{id}:r]?\langle \mathtt{id}_j : r', \{\mathtt{YES}.\mathbf{0}, \mathtt{NO}.\mathbf{0}\}\rangle \mid \\ a\langle s,\mathscr{P}\rangle \mid a[\mathscr{H},\mathscr{O}_1,\mathscr{O}_2,\phi] \end{array}$$

where $\{\mathtt{id}_j \mid j \in J\} = \{\mathtt{id}'' \mid (\mathtt{id}'' : r') \in \mathscr{P} \wedge \mathtt{id}'' \neq \mathtt{id}'\}$
if $\mathtt{B}(r')\mathscr{H}\mathscr{P} = \mathtt{id}'$

[NoChoice]

$$s[\mathtt{id}:r]\exists(\iota : \mathtt{B}(r')).\{P\} \mid a\langle s,\mathscr{P}\rangle \longrightarrow a\langle s,\mathscr{P}\rangle$$

if $\nexists(\mathtt{id}' : r') \in \mathscr{P}$

Table 6. Reduction rules III

1. the identities and roles of all the offers in $\mathscr{O}_1$ whose histories satisfy ϕ;
2. the identities and roles of all the offers in $\mathscr{O}_2$. We do not check the reputations of participants in $\mathscr{O}_2$ since they were good at the moment of the service join and these participants may be active for one session only.

The new session activates the offers in $\mathscr{O}_1 \mid \mathscr{O}_2$ listed in $\mathscr{P}$ by replacing $s[\mathtt{id}:r]$ for the private channel of the offer with identity $\mathtt{id}$ and role r. The resulting session registry does not have one-shot offers and it has the same stable offers. Note that if $[\mathtt{id}:r](y).P$ is a stable offer with a good history, then we get $P\{s[\mathtt{id}:r]/y\}$ for all s created by reducing the service registry. Therefore, in order to preserve channel linearity, our type system requires that y is the only free channel in P (see rule $\lfloor$STAJOIN$\rfloor$ in Table 10).

Rules [ServiceQuit] and [SessionQuit] allow a participant to leave a service or a session, respectively. When quitting a service, the participant is cancelled from the stable offers (if among them), but she remains in the session registries when present. When quitting a session, the participant is cancelled from the session registry.

In both cases the participant's history remains in the service registry. The cancellation from the stable offers is defined by:

$$\mathscr{O}\backslash(\mathtt{id}:r) = \begin{cases} \mathscr{O}' & \text{if } \mathscr{O} \equiv \mathscr{O}' \mid [\mathtt{id}:r](y).P \text{ for some } \mathscr{O}', P \\ \mathscr{O} & \text{otherwise} \end{cases}$$

Rule [Send] describes the standard asynchronous send, implemented by putting the message in the message buffer s.

Rule [SendR] describes the asynchronous send of a relevant message, which must be registered in the history of the sender. Again, the message is put in the message buffer

s. The intention of recording the message sent is expressed by the programmer by using the symbol $!^\bullet$. The history h of principal $\mathtt{id}$ is extended by the pair $(l\langle\mathscr{I}\rangle\langle v\rangle, r)$.

Rule [Receive] specifies the reception of a matching message from the buffer, and the selection of the corresponding continuation.

The most subtle rules are those for the poll and choice operators. Before examining these rules in detail, we start with some observations. Note first that, in a global type, each communication occurs between two participants of the form $\iota : r$ and $\iota' : r'$, whose principal variables ι and ι' are both quantified, either universally or existentially. A universal quantification on ι in role r may either be simple, as in [8], or *conditional*, i.e., controlled by a condition $\mathtt{C}(r, \iota)$ on the history of ι, which will hold only for some of the session participants (possibly none). Its effect is to spawn in parallel all participants satisfying that condition. An existential quantification on ι in role r is always conditional, i.e., guided by some criterion $\mathtt{B}(r)$ on the history of ι. Its effect is to spawn exactly one participant among those best satisfying that criterion, if any. Now, if both the principal variables ι and ι' are conditionally quantified, this means that each role imposes some condition on the other, and hence some potential interactions between the two - possibly all - should be filtered out. Only the "good pairs" of participants, where each partner satisfies the condition required by the other, should be allowed to interact. Now, the fact that a condition is satisfied by a participant can only be checked by the partner requiring that condition. Hence a cross-checking is necessary. For this reason, in the rules for poll and choice, each participant sends a message `YES` to all participants that are "good" in her view (because they comply with her policy) and a message `NO` to all participants that are "bad". Symmetrically, she waits for either `YES` or `NO` from both good and bad participants. For example, suppose that `b1` and `b2` are the only *goldBuyer*s in session s, and that `b1` is a `keen` *goldBuyer* and `b2` is not. In this case, a *goldSeller* `s1` who wants to interact with all `keen` *goldBuyer*s will send `YES` to `b1`, `NO` to `b2` and wait for either `YES` or `NO` from both `b1` and `b2`. The interaction between `s1` and `b1` will start only if `s1` receives `YES` from `b1`. For this reason, the body P of the poll with `b1` replaced for ι must be guarded by the reception of `YES` from `b1`.

Using *gB* and *gS* as short for *goldBuyer* and *goldSeller* we get:

$$s[\mathtt{s1} : gS]\forall(\iota : \mathtt{keen}(gB, \iota).\{P\} \longrightarrow \begin{array}{l} s[\mathtt{s1} : gS]!\langle \mathtt{b1} : gB, \mathtt{YES}\rangle \mid \\ s[\mathtt{s1} : gS]?\langle \mathtt{b1} : gB, \{\mathtt{YES}.P\{\mathtt{b1}/\iota\}, \mathtt{NO}.\mathbf{0}\}\rangle \mid \\ s[\mathtt{s1} : gS]!\langle \mathtt{b2} : gB, \mathtt{NO}\rangle \mid \\ s[\mathtt{s1} : gS]?\langle \mathtt{b2} : gB, \{\mathtt{YES}.\mathbf{0}, \mathtt{NO}.\mathbf{0}\}\rangle \end{array}$$

For instance, if `b1` only wants to interact with `fast` *goldSeller*s, and `s1` is not `fast`, then `b1` will reply `NO` and the interaction will not take place.

This discussion explains the "agreement protocol" in the reduction rule [Poll]. However, there is a further subtlety to take into account. Notice that a quantified principal variable ι may be sent in the content of a message, as part of the set $\mathscr{I}$. As explained in [8], this is essential to avoid ambiguity in the routing of messages. A paradigmatic example is a forwarder:

$$\forall\, \iota_1 : r_1.\forall\, \iota_2 : \mathtt{C}(r_2, \iota_2).\iota_1 \to \iota_2\ \mathtt{OK}; \forall\, \iota_3 : \mathtt{C}'(r_3, \iota_3).\iota_2 \to \iota_3\ \mathtt{OK}\langle \iota_1\rangle$$

If the message sent by ι_2 would not contain ι_1 and there would be more than one principal in role r_1, then the participants in role r_3 could not predict the number of messages

they should receive, which is $n_1 \times n_2$, where n_1 is the number of principals in role r_1 and n_2 is the number of principals in role r_2 satisfying condition $\mathtt{C}(r_2, \iota_2)$.

Now, we want to argue that if a principal identifier is transmitted in a message inside the body of a universal quantification, then this quantification cannot be conditional. Indeed, since such a quantification would occur both in the sending and in the receiving process, and the history of the transmitted principal could change between the time of sending and the time of receiving, a mismatch could arise if the quantification were allowed to be conditional, thus invalidating the property of communication safety. For example, if `b1` : *buyer* sends to `s1` : *seller* all the names of `reliable` couriers, and `c1` : *courier* is `reliable` when `b1` sends the message, but no more `reliable` when `s1` receives it, then the message would remain forever in the buffer. For this reason the typing rules of Section 5 guarantee that all principal variables which occur in messages are universally quantified without conditions.

To formalise these concepts, it is useful to distinguish the two ways in which a principal variable ι may occur in a process P. We say that ι occurs in P:

- as *subject*, if for some role r, P contains a subprocess $-!^*\langle \iota : r, -\langle -\rangle\langle -\rangle\rangle -$ or a subprocess $-?\langle \iota : r, \{-\langle -\rangle(-).-\}\rangle$;
- as *object*, if P contains a subprocess $-!^*\langle -, -\langle \mathscr{I}\rangle\langle -\rangle\rangle$ such that $\iota \in \mathscr{I}$, or a subprocess $-?\langle -, \{-\langle \mathscr{I}_i\rangle(-).-\}_{i\in I}\rangle$ such that $\iota \in \mathscr{I}_i$ for some $i \in I$.

Clearly, the cross-checking described above is sensible only if the quantified principal variable occurs as subject in the body of the quantification. This is always true for well-typed processes in the case of existential quantification. For this reason, in rule [Choice] participant $s[\mathtt{id} : r]$ sends `YES` to a principal of $\mathscr{P}$ playing role r' and having one of the best histories according to the criterion $\mathtt{B}(r')$, and `NO` to all the remaining principals of $\mathscr{P}$ playing role r', and then she waits for `YES` or `NO` from all of them. Rule [NoChoice] is used when there is no principal in role r'.

Similarly, if the quantified principal variable occurs as subject in rule [Poll], participant $s[\mathtt{id} : r]$ sends `YES` or `NO` to all principals of $\mathscr{P}$ playing role r', according to whether their histories satisfy the condition $\mathtt{C}(r', \iota)$ or not, and waits for `YES` or `NO` from all of them. Instead, if ι only occurs as an object in the body of an universal quantification, we apply rule [PassivePoll], which simply spawns in parallel copies of the body with identifiers replaced for the principal variable, as in [8].

A last observation is that a quantification of a participant which does not occur in the body is useless and for this reason our type system does not allow it.

In the contextual rule [Par] the *evaluation contexts* are defined by:

$$[\mathsf{If}-\mathsf{T}] \quad \mathtt{if\ true\ then}\ P\ \mathtt{else}\ Q \longrightarrow P \qquad [\mathsf{If}-\mathsf{F}] \quad \mathtt{if\ false\ then}\ P\ \mathtt{else}\ Q \longrightarrow Q$$

$$[\mathsf{Par}] \quad \frac{P \mid Q \longrightarrow P' \mid Q'}{\mathscr{E}[P] \mid Q \longrightarrow \mathscr{E}[P'] \mid Q'} \qquad [\mathsf{Congr}] \quad \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

Table 7. Reduction rules IV

$$\mathscr{E} ::= [-] \mid \mathscr{E} \mid P \mid \mathscr{E};P \mid (va:G)\mathscr{E} \mid (vs)\mathscr{E} \mid s[\mathtt{id}:r]!^*\langle \mathtt{id}':r', l\langle \mathscr{I} \rangle \langle \mathscr{E} \rangle \rangle$$
$$\mid \mathtt{if}\ \mathscr{E}\ \mathtt{then}\ P\ \mathtt{else}\ P \mid \mathscr{E} \wedge e \mid v \wedge \mathscr{E} \mid \ldots$$

We assume that bound names in $\mathscr{E}$ and free names in Q are disjoint in rule [Par]. Notice that the standard contextual rule:

$$\frac{P \longrightarrow P'}{\mathscr{E}[P] \longrightarrow \mathscr{E}[P']}$$

is derived from rules [Par] and [Congr].

5 Typing

5.1 Types

The syntax of global and local types is given in Table 3. The main novelty with respect to [8] is the addition of the existential quantification. Moreover, our universal quantification is different, as our polling construct does not accept all principals in a certain role, but only the ones verifying a given condition based on their history. Therefore, we concentrate on these two kinds of global and local types and refer the reader to [8] for the other kinds.

The projection from global types to local types is defined as in [8] but for the case of quantifiers, which is given in Table 8.

$$(\forall \iota \notin \mathscr{I} : \mathtt{C}(r,\iota).G) \upharpoonright (\mathtt{id}:r) = G\{\mathtt{id}/\iota\} \upharpoonright (\mathtt{id}:r) \mid (\forall \iota \notin \mathscr{I} \cup \{\mathtt{id}\} : \mathtt{C}(r,\iota).G) \upharpoonright (\mathtt{id}:r) \quad \text{if } \mathtt{id} \notin \mathscr{I}$$
$$(\forall \iota \notin \mathscr{I} : \mathtt{C}(r,\iota).G) \upharpoonright (\mathtt{id}:r') = \forall \iota \notin \mathscr{I} : \mathtt{C}(r,\iota).G \upharpoonright (\mathtt{id}:r') \text{ if } r' \neq r \text{ or } \mathtt{id} \in \mathscr{I}$$
$$(\exists \iota : \mathtt{B}(r).G) \upharpoonright (\mathtt{id}:r) = G\{\mathtt{id}/\iota\} \upharpoonright (\mathtt{id}:r)$$
$$(\exists \iota : \mathtt{B}(r).G) \upharpoonright (\mathtt{id}:r') = \exists \iota : \mathtt{B}(r).G \upharpoonright (\mathtt{id}:r') \qquad \text{if } r' \neq r$$

Table 8. Projection of quantified global types

Well-formed global types must satisfy all conditions given in [8], i.e., they must be syntactically correct, projectable and linear.

5.2 Typing rules

As usual, to type sessions we use a session environment, ranged over by Δ, which associates local types with channels, as well as a standard environment, ranged over by

$$\frac{}{\Gamma \vdash \mathtt{true}, \mathtt{false} : bool}\ \lfloor \textsc{Bool} \rfloor \quad \frac{\Gamma \vdash e_i : bool \quad (i = 1,2)}{\Gamma \vdash e_1 \vee e_2 : bool}\ \lfloor \textsc{Or} \rfloor \quad \ldots$$

$$\frac{}{\Gamma \vdash \emptyset}\ \lfloor \textsc{PSE} \rfloor \quad \frac{\Gamma \vdash \mathscr{I}}{\Gamma \vdash \mathscr{I} \cup \{\mathtt{id}\}}\ \lfloor \textsc{PSI} \rfloor \quad \frac{\Gamma \vdash \mathscr{I} \quad \iota : r \in \Gamma}{\Gamma \vdash \mathscr{I} \cup \{\iota\}}\ \lfloor \textsc{PSV} \rfloor$$

$$\frac{}{\Gamma \vdash \mathtt{id} : r}\ \lfloor \textsc{PA} \rfloor \quad \frac{\iota : r \in \Gamma}{\Gamma \vdash \iota : r}\ \lfloor \textsc{PAV} \rfloor$$

Table 9. Typing rules for expressions and participants

$$\frac{\Delta : \mathrm{end}}{\Gamma \vdash_{\mathtt{id}} \mathbf{0} \triangleright \Delta}\ \lfloor \textsc{Nil} \rfloor \qquad \frac{}{\Gamma, X : \Delta \vdash_{\mathtt{id}} X \triangleright \Delta}\ \lfloor \textsc{RVar} \rfloor \qquad \frac{\Gamma, X : \Delta \vdash_{\mathtt{id}} P \triangleright \Delta}{\Gamma \vdash_{\mathtt{id}} \mu X.P \triangleright \Delta}\ \lfloor \textsc{Rec} \rfloor$$

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash_{\mathtt{id}} P \triangleright y : G \upharpoonright (\mathtt{id} : r)}{\Gamma \vdash_{\mathtt{id}} \overline{u}[\mathtt{id} : r, \mathscr{C}(\tilde{r})](y).\{P\} \triangleright \emptyset}\ \lfloor \textsc{StaJoin} \rfloor$$

$$\frac{\Gamma \vdash u : \langle G \rangle \quad \Gamma \vdash_{\mathtt{id}} P \triangleright \Delta, y : G \upharpoonright (\mathtt{id} : r) \quad \mathtt{ns}(\Delta)}{\Gamma \vdash_{\mathtt{id}} u[\mathtt{id} : r, \mathscr{C}(\tilde{r})](y).\{P\} \triangleright \Delta}\ \lfloor \textsc{OSJoin} \rfloor$$

$$\frac{\Delta : \mathrm{end}}{\Gamma, a : \langle G \rangle \vdash_{\mathtt{id}} \mathtt{quit}(a, \mathtt{id} : r) \triangleright \Delta}\ \lfloor \textsc{SerQuit} \rfloor \qquad \frac{\Delta : \mathrm{end}}{\Gamma \vdash_{\mathtt{id}} \mathtt{quit}\langle c \rangle \triangleright \Delta, c : \mathrm{end}}\ \lfloor \textsc{SesQuit} \rfloor$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash \mathscr{I}_j \quad \Gamma \vdash e : S_j \quad \Gamma \vdash_{\mathtt{id}} P \triangleright \Delta, c : T_j \quad j \in I}{\Gamma \vdash_{\mathtt{id}} c!^*\langle p, l_j\langle \mathscr{I}_j \rangle \langle e \rangle\rangle; P \triangleright \Delta, c :!^*\langle p, \{l_i\langle \mathscr{I}_i \rangle \langle S_i \rangle.T_i\}_{i \in I}\rangle}\ \lfloor \textsc{VSend} \rfloor$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash \mathscr{I}_i \qquad \Gamma, x_i : S_i \vdash_{\mathtt{id}} P_i \triangleright \Delta, c : T_i \quad \forall i \in I}{\Gamma \vdash_{\mathtt{id}} c?\langle p, \{l_i\langle \mathscr{I}_i \rangle (x_i).P_i\}_{i \in I}\rangle \triangleright \Delta, c :?\langle p, \{l_i\langle \mathscr{I}_i \rangle \langle S_i \rangle.T_i\}_{i \in I}\rangle}\ \lfloor \textsc{VRcv} \rfloor$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash \mathscr{I}_j \quad \Gamma \vdash_{\mathtt{id}} P \triangleright \Delta, c : T_j \quad j \in I}{\Gamma \vdash_{\mathtt{id}} c!\langle p, l_j\langle \mathscr{I}_j \rangle \langle c' \rangle\rangle; P \triangleright \Delta, c :!^*\langle p, \{l_i\langle \mathscr{I}_i \rangle \langle T \rangle.T_i\}_{i \in I}\rangle, c' : T}\ \lfloor \textsc{CSend} \rfloor$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash \mathscr{I}_i \qquad \Gamma \vdash_{\mathtt{id}} P_i \triangleright \Delta, c : T_i, x_i : T \quad \forall i \in I}{\Gamma \vdash_{\mathtt{id}} c?\langle p, \{l_i\langle \mathscr{I}_i \rangle (x_i).P_i\}_{i \in I}\rangle \triangleright \Delta, c :?\langle p, \{l_i\langle \mathscr{I}_i \rangle \langle T \rangle.T_i\}_{i \in I}\rangle}\ \lfloor \textsc{CRcv} \rfloor$$

$$\frac{\Gamma, \iota : r \vdash_{\mathtt{id}} P \triangleright c : T \quad \mathtt{ubi}(T, \iota) \wedge \mathtt{noo}(T, \iota)}{\Gamma \vdash_{\mathtt{id}} c\forall(\iota \notin \mathscr{I} : \mathtt{C}(r, \iota)).\{P\} \triangleright c : \forall\, \iota \notin \mathscr{I} : \mathtt{C}(r, \iota).T}\ \lfloor \textsc{Poll} \rfloor$$

$$\frac{\Gamma, \iota : r \vdash_{\mathtt{id}} P \triangleright c : T \quad \mathtt{ubi}(T, \iota)}{\Gamma \vdash_{\mathtt{id}} c\forall(\iota \notin \mathscr{I} : r).\{P\} \triangleright c : \forall \iota \notin \mathscr{I} : r.T}\ \lfloor \textsc{PollAll} \rfloor$$

$$\frac{\Gamma, \iota : r \vdash_{\mathtt{id}} P \triangleright c : T \quad \mathtt{sub}(T, \iota) \wedge \mathtt{noo}(T, \iota)}{\Gamma \vdash_{\mathtt{id}} c\exists(\iota : \mathtt{B}(r)).\{P\} \triangleright c : \exists\, \iota : \mathtt{B}(r).T}\ \lfloor \textsc{Choice} \rfloor$$

$$\frac{\Gamma \vdash_{\mathtt{id}} P_1 \triangleright \Delta_1 \quad \Gamma \vdash_{\mathtt{id}} P_2 \triangleright \Delta_2}{\Gamma \vdash_{\mathtt{id}} P_1; P_2 \triangleright \Delta_1; \Delta_2}\ \lfloor \textsc{Seq} \rfloor \qquad \frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash_{\mathtt{id}} P_1 \triangleright \Delta \quad \Gamma \vdash_{\mathtt{id}} P_2 \triangleright \Delta}{\Gamma \vdash_{\mathtt{id}} \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \triangleright \Delta}\ \lfloor \textsc{If} \rfloor$$

$$\frac{\Gamma \vdash_{\mathtt{id}} P_1 \triangleright \Delta_1 \quad \Gamma \vdash_{\mathtt{id}} P_2 \triangleright \Delta_2}{\Gamma \vdash_{\mathtt{id}} P_1 \mid P_2 \triangleright \Delta_1 \mid \Delta_2}\ \lfloor \textsc{Par}_{\mathtt{id}} \rfloor \qquad \frac{\Gamma, a : \langle G \rangle \vdash_{\mathtt{id}} P \triangleright \Delta}{\Gamma \vdash_{\mathtt{id}} (\nu a : G)P \triangleright \Delta}\ \lfloor \textsc{Res}_{\mathtt{id}} \rfloor$$

Table 10. Type system $\vdash_{\mathtt{id}}$

$$\frac{\Gamma \vdash_{\mathtt{id}} P \triangleright \Delta}{\Gamma \vdash P \triangleright \Delta}\ \lfloor \text{NOid} \rfloor \qquad \frac{\Gamma \vdash a : \langle G \rangle}{\Gamma \vdash a\langle G, \phi \rangle \triangleright \emptyset}\ \lfloor \text{INIT} \rfloor$$

$$\frac{\Gamma \vdash P_1 \triangleright \Delta_1 \quad \Gamma \vdash P_2 \triangleright \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \mid \Delta_2}\ \lfloor \text{PAR} \rfloor \qquad \frac{\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a : G)P \triangleright \Delta}\ \lfloor \text{RES} \rfloor$$

Table 11. Type system $\vdash$

Γ, which associates sorts with value variables, global sorts with service names, roles with principal variables, and session environments with process variables.

$$\Delta ::= \emptyset \mid c : T \qquad \Gamma ::= \emptyset \mid \Gamma, x : S \mid a : \langle G \rangle \mid \Gamma, \iota : r \mid \Gamma, X : \Delta$$

Table 9 gives the typing rules for expressions and participants, taken from [8].

Our typing for processes assures that the joins and quits of two different principals cannot be *sequentialised*. This condition means that if there is an order among the actions of different principals, this must be made explicit via some communications, and should not be hidden by a sequentialisation (for instance, in the example of Section 2, we want to allow the same principal to perform some actions first as a *seller* and then as a *goldSeller*, but we do not want the actions of principal `b1` to depend on the actions of principal `s1`, without informing them both).

There are two kinds of typing judgments for processes. The most liberal judgment is $\Gamma \vdash P \triangleright \Delta$: it says that under the assumptions in Γ the channels in the process P have the local types prescribed by Δ. The judgment $\Gamma \vdash_{\mathtt{id}} P \triangleright \Delta$ assures also that `id` is the only principal occurring in P. This is used to guarantee the condition discussed above.

Table 10 contains the rules for the system $\vdash_{\mathtt{id}}$, which we briefly comment.

The session environments for $\mathbf{0}$ (rule $\lfloor$NIL$\rfloor$) can only contain the types ε and end: this is enforced by the premise Δ : end, which means that all types occurring in Δ are either ε or end. Rules $\lfloor$RVAR$\rfloor$ and $\lfloor$REC$\rfloor$ for recursion are standard.

As usual, rules $\lfloor$STAJOIN$\rfloor$ and $\lfloor$OSJOIN$\rfloor$ check that the local type of the participant channel coincides with the projection of the global type for the required role. Moreover, to type a stable service join we require that the participant channel is the only channel in the body of the join. This is necessary in order to assure that the application of rule [SessionInit] preserves the linearity of channels, see page 11. Peculiar to our system is also the condition in rule $\lfloor$OSJOIN$\rfloor$, stating that all channels but y in P have types terminating by end or by a recursion variable. The reason for this restriction is to prevent channels in P from being used in processes following P (see the discussion at page 10). To this aim we define the predicate $\mathtt{ns}(T)$, letting $\dagger$ range over $\{?, !^*\}$:

$$\mathtt{ns}(\dagger\langle p, \{l_i\langle \mathscr{I}_i \rangle \langle U_i \rangle . T_i\}_{i \in I} \rangle) = \textstyle\bigwedge_{i \in I} \mathtt{ns}(T_i) \quad \mathtt{ns}(T \mid T') = \mathtt{ns}(T) \wedge \mathtt{ns}(T')$$
$$\mathtt{ns}(\forall\, \iota' \notin \mathscr{I} : \mathtt{C}(r, \iota').T) = \mathtt{ns}(\exists\, \iota' : \mathtt{B}(r).T) = \mathtt{ns}(\mu \mathbf{x}.T) = \mathtt{ns}(T' ;\, T) = \mathtt{ns}(T)$$
$$\mathtt{ns}(\mathbf{x}) = \mathtt{ns}(\text{end}) = \mathtt{true} \quad \mathtt{ns}(\varepsilon) = \mathtt{false}$$

We then extend this predicate to session environments by letting $\mathtt{ns}(\Delta) = \bigwedge_{c:T \in \Delta} \mathtt{ns}(T)$.

A participant may ask to quit a service (rule $\lfloor$SERQUIT$\rfloor$) at any point. She will cease to take part in the service starting from the first session initiated after her withdrawal. Instead, rule $\lfloor$SESQUIT$\rfloor$ prescribes that a session may be quit only after the participant has terminated her task.

$$\mathtt{ubi}(\dagger\langle p,\{l_i\langle\mathscr{I}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle,\iota)=\begin{cases}\mathtt{true} & \text{if } p=\iota:r \text{ for some } r \text{ or}\\ & \iota\in\mathscr{I}_i \text{ for all } i\in I\\ \mathtt{false} & \text{otherwise}\end{cases}$$

$$\mathtt{ubi}(\forall\,\iota'\notin\mathscr{I}:\mathtt{C}(r,\iota').T,\iota)=\mathtt{ubi}(\exists\,\iota':\mathtt{B}(r).T,\iota)=\mathtt{ubi}(\mu\mathbf{x}.T,\iota)=\mathtt{ubi}(T,\iota)$$
$$\mathtt{ubi}(T\mid T',\iota)=\mathtt{ubi}(T\;;\;T',\iota)=\mathtt{ubi}(T,\iota)\wedge\mathtt{ubi}(T',\iota)$$
$$\mathtt{ubi}(\mathbf{x},\iota)=\mathtt{ubi}(\varepsilon,\iota)=\mathtt{ubi}(\mathrm{end},\iota)=\mathtt{true}$$

$$\mathtt{noo}(\dagger\langle p,\{l_i\langle\mathscr{I}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle,\iota)=\begin{cases}\mathtt{true} & \text{if } \iota\notin\mathscr{I}_i \text{ for all } i\in I,\\ \mathtt{false} & \text{otherwise}\end{cases}$$

$$\mathtt{noo}(\forall\,\iota'\notin\mathscr{I}:\mathtt{C}(r,\iota').T,\iota)=\mathtt{noo}(\exists\,\iota':\mathtt{B}(r).T,\iota)=\mathtt{noo}(\mu\mathbf{x}.T,\iota)=\mathtt{noo}(T,\iota)$$
$$\mathtt{noo}(T\mid T',\iota)=\mathtt{noo}(T\;;\;T',\iota)=\mathtt{noo}(T,\iota)\wedge\mathtt{noo}(T',\iota)$$
$$\mathtt{noo}(\mathbf{x},\iota)=\mathtt{noo}(\varepsilon,\iota)=\mathtt{noo}(\mathrm{end},\iota)=\mathtt{true}$$

$$\mathtt{sub}(\dagger\langle p,\{l_i\langle\mathscr{I}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle,\iota)=\begin{cases}\mathtt{true} & \text{if } p=\iota:r \text{ for some } r,\\ \mathtt{false} & \text{otherwise}\end{cases}$$

$$\mathtt{sub}(\forall\,\iota'\notin\mathscr{I}:\mathtt{C}(r,\iota').T,\iota)=\mathtt{sub}(\exists\,\iota':\mathtt{B}(r).T,\iota)=\mathtt{sub}(\mu\mathbf{x}.T,\iota)=\mathtt{sub}(T,\iota)$$
$$\mathtt{sub}(T\mid T',\iota)=\mathtt{sub}(T\;;\;T',\iota)=\mathtt{sub}(T,\iota)\vee\mathtt{sub}(T',\iota)$$
$$\mathtt{sub}(\mathbf{x},\iota)=\mathtt{sub}(\varepsilon,\iota)=\mathtt{sub}(\mathrm{end},\iota)=\mathtt{false}$$

Table 12. The predicates ubi, noo and sub

The system types communications on channel c with participant p allowing different labels, sequences of participants, values and continuations (rules $\lfloor$VSEND$\rfloor$ and $\lfloor$VRCV$\rfloor$). Also the exchange of channels can be typed (rules $\lfloor$CSEND$\rfloor$ and $\lfloor$CRCV$\rfloor$). Notice that the type system does not allow sending a channel with $!^\bullet$ since there is no meaning in putting a channel name in the history of a principal. Moreover, the session names occurring in channels are restricted and therefore, in order to do that, we should enlarge the scope of these restrictions, making it impossible to use the current structural equivalence to cancel exhausted sessions.

The typing of a quantification requires a unique channel in the session environment (rules $\lfloor$POLL$\rfloor$, $\lfloor$POLLALL$\rfloor$ and $\lfloor$CHOICE$\rfloor$). As argued in the previous section, participants that occur in messages (i.e., that occur as objects in processes) should be universally quantified without conditions on their histories. There is another condition that must be satisfied in order to avoid message ambiguity: participants who are universally quantified should appear (either as a subject or as an object) in every communication occurring in the body of the quantification. In order to check the above conditions, it is handy to define three predicates on local types and principal variables. The predicate $\mathtt{ubi}(T,\iota)$ is true if all selections/branchings in T contain ι. The predicate $\mathtt{noo}(T,\iota)$ is true if no selection/branching in T has ι as object. The predicate $\mathtt{sub}(T,\iota)$ is true if there is at least one selection/branching in T having ι as subject. More precisely, letting $\dagger$ range over $\{?,!^*\}$, these predicates are defined by the clauses in Table 12.

To type the poll (rules $\lfloor$POLL$\rfloor$ and $\lfloor$POLLALL$\rfloor$) in such a way that we avoid ambiguous messages, it is necessary that ι occurs in all selections/branchings of T, condition assured by $\mathtt{ubi}(T,\iota)$. Moreover, if the poll is conditional (rule $\lfloor$POLL$\rfloor$), then ι cannot occur as an object in T. For this reason we require $\mathtt{noo}(T,\iota)$ too.

In rule $\lfloor$CHOICE$\rfloor$ $\mathtt{sub}(T,\iota)$ assures that ι occurs at least once as a subject in T, and $\mathtt{noo}(T,\iota)$ assures that ι does not occur as an object in T.

$$\frac{\Gamma \vdash P \triangleright \Delta}{\Gamma \vdash (\nu s)P \triangleright \Delta} \lfloor \textsc{SesRes} \rfloor \qquad \frac{\Gamma \vdash \Delta \triangleright \mathrm{end}}{\Gamma \vdash s : [\,] \triangleright \Delta} \lfloor \textsc{EBuff} \rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta, y : T}{\Gamma \vdash [\mathtt{id} : r](y).P \triangleright \Delta} \lfloor \textsc{Off} \rfloor \qquad \frac{\Gamma \vdash \mathscr{O}_1 \triangleright \Delta_1 \quad \Gamma \vdash \mathscr{O}_2 \triangleright \Delta_2}{\Gamma \vdash \mathscr{O}_1 \mid \mathscr{O}_2 \triangleright \Delta_1 \mid \Delta_2} \lfloor \textsc{ParOff} \rfloor$$

$$\frac{\Gamma \vdash s : \mathscr{B} \triangleright \Delta \quad \Gamma \vdash v : U}{\Gamma \vdash s : (\mathtt{id} : r, \mathtt{id}' : r', l\langle \mathscr{I} \rangle \langle v \rangle) \cdot \mathscr{B} \triangleright \{s[\mathtt{id} : r] : !\langle \mathtt{id}' : r', l\langle \mathscr{I} \rangle \langle U \rangle \rangle\}; \Delta} \lfloor \textsc{SMess} \rfloor$$

$$\frac{\Gamma \vdash \mathscr{O}_1 \triangleright \Delta_1 \quad \Gamma \vdash \mathscr{O}_2 \triangleright \Delta_2}{\Gamma \vdash a[\mathscr{H}, \mathscr{O}_1, \mathscr{O}_2, \phi] \triangleright \Delta_1 \mid \Delta_2} \lfloor \textsc{SerReg} \rfloor \qquad \frac{}{\Gamma \vdash a\langle s, \mathscr{P} \rangle \triangleright \emptyset} \lfloor \textsc{SesReg} \rfloor$$

Table 13. Typing rules for runtime processes

For typing the sequential composition of processes, in rule $\lfloor \textsc{Seq} \rfloor$ we use the sequential composition of session environments defined by:

$$\Delta; \Delta' = \Delta \setminus \mathscr{D}(\Delta') \cup \Delta' \setminus \mathscr{D}(\Delta) \cup \{c : \Delta(c); \Delta'(c) \parallel c \in \mathscr{D}(\Delta) \cap \mathscr{D}(\Delta')\}$$

Rule $\lfloor \textsc{Par}_{\mathtt{id}} \rfloor$ (as well as rule $\lfloor \textsc{Par} \rfloor$ in Table 11) uses the following parallel composition of session environments:

$$\Delta \mid \Delta' = \Delta \setminus \mathscr{D}(\Delta') \cup \Delta' \setminus \mathscr{D}(\Delta) \cup \{c : (\Delta(c) \mid \Delta'(c)) \parallel c \in \mathscr{D}(\Delta) \cap \mathscr{D}(\Delta')\}$$

Notice that a service initialisation cannot be sequentialised, since it cannot be typed in the system $\vdash_{\mathtt{id}}$, but only in the system $\vdash$ (rule $\lfloor \textsc{Init} \rfloor$ in Table 11).

The rules of Tables 10 and 11 are enough for typing user processes. For typing runtime processes, we extend the syntax of local types with *message types* of the shape $!\langle \mathtt{id} : r, l\langle \mathscr{I} \rangle \langle U \rangle \rangle$ and use all the rules in the tables above plus the rules of Table 13. We notice that the rules for typing the registries are simpler than the corresponding rule in [8], thanks to our distinction between services and sessions.

6 Properties

Our calculus enjoys type safety, which is obtained from the properties of subject reduction (Subsection 6.1) and progress (Subsection 6.2). Moreover, there is an interesting relation between the local types and the possible future reputations (Subsection 6.3).

6.1 Subject Reduction

In order to state the subject reduction property, we need to define a reduction relation on session environments, which describes how these environments evolve during process execution. Table 14 gives this relation, which mimics the sending and receiving of values and channels. The sets of identifiers in the reduction rules for quantifiers are arbitrary. In this table, we consider types in Δ modulo an equivalence relation reflecting the equivalence relation on buffers, and we define type contexts $\mathscr{T}$ as:

$$\mathscr{T} ::= [-] \parallel \mathscr{T} \mid T \parallel T \mid \mathscr{T} \parallel \mathscr{T}; T$$

We need to start from a well-typed initial process in order to assure that participants respect the prescriptions of some global type. We say that a process P is *reachable* if there is a well-typed initial process P_0 such that $P_0 \longrightarrow^* \mathscr{E}[P]$.

$$\{s[\mathtt{id}:r]:!^*\langle \mathtt{id}':r', \{l_i\langle \mathscr{S}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle\} \Rightarrow \{s[\mathtt{id}:r]:!^*\langle \mathtt{id}':r', l_k\langle \mathscr{S}_k\rangle\langle U_k\rangle\rangle; T_k\} \qquad k\in I$$

$$\{s[\mathtt{id}:r]:!^*\langle \mathtt{id}':r', l_k\langle \mathscr{S}_k\rangle\langle U_k\rangle\rangle, s[\mathtt{id}':r']:?\langle \mathtt{id}:r, \{l_i\langle \mathscr{S}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle\} \Rightarrow \{s[\mathtt{id}:r]:\varepsilon, s[\mathtt{id}':r']:T_k\} \qquad k\in I$$

$$\{s[\mathtt{id}:r]:\forall\, \iota\notin\mathscr{S}:\mathtt{C}(r',\iota).T\} \Rightarrow \{s[\mathtt{id}:r]:\Pi_{i\in I}!\langle \mathtt{id}_i:r', \mathtt{YES}\rangle; ?\langle \mathtt{id}_i:r', \{\mathtt{YES}.T\{\mathtt{id}_i/\iota\}, \mathtt{NO}.\varepsilon\}\rangle \mid \Pi_{j\in J}!\langle \mathtt{id}_j:r', \mathtt{NO}\rangle; ?\langle \mathtt{id}_j:r', \{\mathtt{YES}.\varepsilon, \mathtt{NO}.\varepsilon\}\} \quad \text{where } \forall i\in I\cup J.\mathtt{id}_i\notin\mathscr{S}$$

$$\{s[\mathtt{id}:r]:\forall\, \iota\notin\mathscr{S}:r'.T\} \Rightarrow \{s[\mathtt{id}:r]:\Pi_{i\in I}T\{\mathtt{id}_i/\iota\}\} \qquad \text{where } \forall i\in I.\mathtt{id}_i\notin\mathscr{S}$$

$$\{s[\mathtt{id}:r]:\exists\, \iota:\mathtt{B}(r').T\} \Rightarrow \quad \{s[\mathtt{id}:r]:!\langle \mathtt{id}':r', \mathtt{YES}\rangle; ?\langle \mathtt{id}':r', \{\mathtt{YES}.T\{\mathtt{id}'/\iota\}, \mathtt{NO}.\varepsilon\}\rangle \mid \Pi_{j\in J}!\langle \mathtt{id}_j:r', \mathtt{NO}\rangle; ?\langle \mathtt{id}_j:r', \{\mathtt{YES}.\varepsilon, \mathtt{NO}.\varepsilon\}\}$$

$$\{s[\mathtt{id}:r]:T\}\cup\Delta \Rightarrow \{s[\mathtt{id}:r]:T'\}\cup\Delta' \text{ implies } \{s[\mathtt{id}:r]:\mathscr{T}[T]\}\cup\Delta \Rightarrow \{s[\mathtt{id}:r]:\mathscr{T}[T']\}\cup\Delta'$$

$$\Delta\Rightarrow\Delta' \text{ implies } \Delta\cup\Delta''\Rightarrow\Delta'\cup\Delta''$$

Table 14. Reduction of session environments

As usual for session calculi, the reduction of processes gives rise to the reduction of session environments.

Theorem 1. *If P is a reachable process and $\Gamma\vdash P\triangleright\Delta$ and $P\longrightarrow^* P'$, then $\Gamma\vdash P'\triangleright\Delta'$ for some Δ' such that $\Delta\Rightarrow^*\Delta'$.*

6.2 Communication Safety and Progress

As usual, communication safety assures that every receiver will find an appropriate message in the buffer and, conversely, that every message in the buffer will be fetched by a matching receiver.

Definition 1. *A process P is* communication safe *if:*

- $P\equiv\mathscr{E}[s[\mathtt{id}:r]?\langle \mathtt{id}':r', \{l_i\langle \mathscr{S}_i\rangle(x_i).P_i\}_{i\in I}\rangle]$ *implies that* $\mathscr{E}[\mathbf{0}]\longrightarrow^*\mathscr{E}'[s:(\mathtt{id}':r', \mathtt{id}:r, l_k\langle \mathscr{S}_k\rangle\langle v\rangle)\cdot\mathscr{B}]$ *with* $k\in I$*;*
- $P\equiv\mathscr{E}[s:(\mathtt{id}':r', \mathtt{id}:r, l_k\langle \mathscr{S}_k\rangle\langle v\rangle)\cdot\mathscr{B}]$ *implies that* $\mathscr{E}[\mathbf{0}]\longrightarrow^*\mathscr{E}'[s[\mathtt{id}:r]?\langle \mathtt{id}':r', \{l_i\langle \mathscr{S}_i\rangle(x_i).P_i\}_{i\in I}\rangle]$ *with* $k\in I$.

It is well known [1] that interleaving different services can destroy communication safety also in sessions without roles. In the present calculus also nested joins can destroy communication safety, since joins can fail when one of the required conditions is not satisfied. So we will only consider processes that use a single service and which can be typed with a derivation where:

1. session environments which appear in premises or conclusions of the system $\vdash_{\mathtt{id}}$ contain at most one association between a local type and a channel;
2. in rule $\lfloor$SEQ$\rfloor$, if the session environment of the first premise is empty, then the session environment of the second premise must be empty too.

The first condition assures that communications on two different channels can only occur in two parallel threads. The second condition forbids nested joins, since the first

condition assures that the session environments for typing joins are empty. It allows instead sequentialisation of joins (when both session environments are empty), sequentialisation of communications on the same channel (when both session environments assign types to this channel), and communications on one channel followed by one join (when the first session environment assigns a type to this channel and the second session environment is empty). We denote by $\vdash^\star$ such kind of derivations.

The calculus of [8] requires a locking/unlocking mechanism to ensure that a service is "well-locked", i.e., that it does not allow a principal to join an ongoing session. Our distinction between services and sessions makes all services well-locked without having to synchronise joins, as hinted previously.

Lemma 1. *Let P be an initial process not containing restrictions. If $a : \langle G \rangle \vdash^\star P \rhd \emptyset$ and $P \longrightarrow^* P'$, then P' is communication safe.*

In session calculi, progress does not only ask for the absence of service interleaving, but also for the presence of all required participants. In [8] too, progress is assured under the condition that the needed principals can join. In our calculus:

– polls can properly reduce also when no principal satisfies the required condition;
– choices always reduce.

This means that we can avoid to add processes in parallel when defining progress.

The most important peculiarities of our calculus are:

– service registries are permanent and they can always reduce by rule [SessionInit];
– service joins can require conditions which are not satisfied.

The standing availability of rule [SessionInit] implies that reducibility by this rule cannot be considered to assure progress.

Definition 2. *A process P has the* progress property *if $P \longrightarrow^* P'$ implies that either P' does not contain runtime channels, or there exists P'' such that $P' \longrightarrow P''$ using a rule different from* [SessionInit] *and P'' has the progress property.*

According to this definition a process with progress can reduce to a parallel composition of service registers and service joins with unsatisfied conditions, which can only reduce by rule [SessionInit] to itself (modulo structural equivalence), since the generated sessions have no participants and so they can be garbage collected.

The progress proof essentially uses communication safety, and the observation that, starting from an initial process with a single service, the required registries and named buffers will be present for sure.

Theorem 2. *Let $P \equiv a\langle G, \phi \rangle \mid P_0$ be an initial process not containing restrictions. If $a : \langle G \rangle \vdash^\star P \rhd \emptyset$, then P has the progress property.*

6.3 Local Types for Reputations

We now discuss how to take advantage from local types to predict possible future reputations of principals. To this end, it is handy to define reductions which activate at most one session for each service.

Definition 3. *A reduction is* one-session *if rule* [SessionInit] *can be applied to a session registry for service a only if the current process does not contain $a\langle s, \mathscr{P}\rangle$ for some s, $\mathscr{P}$.*

Note that in [8] a service contains only one session, so all reductions are one-session.

Let h be the history of a principal `id` at the end of the execution of a session in a one-session reduction. Then, in the next session for the same service, principal `id` is allowed to play a role r only if $\phi(h,r)$.

The local type of a role in a service, together with the number of participants in a session, allows us to compute an upper bound to the number of occurrences of a fixed label in the possible histories - of principals playing that role - which can be generated by executing the session, provided the label does not occur under recursion. More precisely, if n_r is the number of principals playing role r in the session, then the number of occurrences of label l in the histories of a role with local type T is bounded by $\#(T,l)$ defined by:

$$\#(!^{\bullet}\langle p, \{l_i\langle \mathscr{I}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle, l) = \begin{cases} max\{\#(T_{i_0},l)+1, \#(T_i,l) \mid i \in I \setminus \{i_0\}\} & \text{if } l = l_{i_0} \ \&\ i_0 \in I, \\ max\{\#(T_i,l) \mid i \in I\} & \text{otherwise.} \end{cases}$$

$$\begin{array}{llll} \#(!\langle p, \{l_i\langle \mathscr{I}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle, l) & = max\{\#(T_i,l) \mid i \in I\} & \#(\mu \mathbf{x}.T, l) & = 0 \\ \#(?\langle p, \{l_i\langle \mathscr{I}_i\rangle\langle U_i\rangle.T_i\}_{i\in I}\rangle, l) & = max\{\#(T_i,l) \mid i \in I\} & \#(\mathbf{x}, l) & = 0 \\ \#(\forall\, \iota \notin \mathscr{I} : \mathtt{C}(r,\iota).T, l) & = \#(T,l) \times n_r & \#(\varepsilon, l) & = 0 \\ \#(\exists\, \iota : \mathtt{B}(r).T, l) & = \#(T,l) & \#(\mathrm{end}, l) & = 0 \\ \#(T_1 \mid T_2, l) & = \#(T_1,l) + \#(T_2,l) & \#(T_1\ ;\ T_2, l) & = \#(T_1,l) + \#(T_2,l) \end{array}$$

We can exploit this information to choose ϕ bounding the number of occurrences of label l in (part of) the histories, when using one-session reduction. It is enough to set $\phi(h,r)$ to $m + \#(T,l) \leq M$, where m is the number of occurrences of l in the considered part of h, type T is the local type of r and M is the desired bound.

For example, we can modify the *goldBuyer* of Figure 3 by recording in her history the labels `BUY` and `AST`. The local type T of the *goldBuyer* then contains

$$!^{\bullet}\langle \iota : goldSeller, \{\mathtt{BUY}\ldots., \mathtt{AST}\ldots.\}\rangle$$

and $\#(T, \mathtt{AST})$=1. Therefore, if we want to limit to 3 the number of assistance calls in the last 20 transitions, the joining condition for the *goldBuyer* can hold true only if in the last 19 transitions the number of `AST` is less than or equal to 2.

7 Conclusions and related work

In this paper, we studied a role-based multiparty session calculus that takes into account the history of principals, in order to measure their reputation and regulate accordingly their participation in future conversations. Histories are dynamically built by collecting actions performed by principals, in such a way that, if a participant "behaves badly" in a service, this will hinder her further attempts to join the service with particular roles and her possibilities to be chosen by other participants via a poll or a choice operation.

Since in our setting the reputation associated with a principal is *objective* and not subjective (i.e., it is based on real interactions and not on other principals' opinions), one of the major problems arising in reputation systems, *unfair ratings*, is avoided.

We managed to model the regulation of a principal's behaviour depending on her reputation: in Section 2, we showed how a principal's "bad behaviour" may restrict the range of session roles offered by the service to that principal. This is our main result.

However, our solution still suffers from some limitations, in particular we can only type a limited form of delegation, the same as in [8], that does not allow general scenarios to be modelled. The limitation is due to the fact that session environments in the typing rules for poll and choice must contain exactly one channel, while the session environments for typing delegation have at least two channels. Therefore it is impossible to create a channel to be delegated before a poll or a choice. The only way out is to create and discharge it afterwards, unused, by means of a join, right before sending it.

Session calculi were proposed in the mid-nineties to model communication protocols among concurrent and mobile processes. We refer to [9] and [17] for overviews. Since the original proposal of [12], such calculi have been extensively studied and enriched with various features. Initially dealing with binary protocols (often representing an interaction between a user and a server), session calculi have been subsequently extended to *multiparty* sessions [13], involving several principals interacting on an equal footing. More recently, multiparty sessions have been extended with design by contracts [2], dependent types for parametricity [18], upper bounds on buffer sizes [7], exception handling [5], access and information flow control [4] and monitors [3]. The present paper mainly builds on the *role-based* multiparty calculus of [8], as previously discussed.

The study and formalisation of *reputation* has similarly attracted a great deal of interest in recent years. We refer to existing surveys [15, 14, 11] for a general introduction to *reputation systems*. It is interesting to notice that the reputation system associated with our calculus can be classified, according to [15], as a non-probabilistic experience-based system, where principals are evaluated by inspecting their history, which is built by recording their past interaction with other principals.

As it is grounded on the π-calculus, our proposal may be directly compared with the Calculus for Trust Management (`ctm`) [6], a process calculus for modelling trust based systems. Principals in `ctm` have two components: the protocol and the policy. Protocols are π-calculus style processes. Policies are made of two parts: logic formulae (similar to our single conditions), which describe the rules for taking decisions on the basis of past experiences; experiences (similar to our histories), which collect the messages exchanged in interactions between principals. The treatment of [6] differs from ours in that policies and histories are local and associated with each principal, while we store them in a registry which is global for all participants in a given service. In our calculus, histories are made of sent values and may be checked by both services and other principals involved in the same service; in `ctm`, histories are made of received values and are checked locally before granting access to local resources. Moreover, in `ctm` the focus is on barbed equivalences among principals, while we are mainly concerned with supplying a type system to check communication safety.

Acknowledgments. We would like to thank Nobuko Yoshida and Pierre-Malo Deniélou for useful comments.

References

1. L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In F. v. Breugel and M. Chechik, editors, *Proc. CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
2. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In P. Gastin and F. Laroussinie, editors, *Proc. CONCUR'10*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
3. S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini. Information Flow Safety in Multiparty Sessions. In B. Luttik and F. Valencia, editors, *EXPRESS'11*, volume 64 of *EPTCS*, pages 16–31, 2011.
4. S. Capecchi, I. Castellani, M. Dezani Ciancaglini, and T. Rezk. Session Types for Access and Information Flow Control. In P. Gastin and F. Laroussinie, editors, *Proc. CONCUR'10*, volume 6269 of *LNCS*, pages 237–252. Springer, 2010.
5. S. Capecchi, E. Giachino, and N. Yoshida. Global Escape in Multiparty Sessions. In K. Lodaya and M. Mahajan, editors, *Proc. FSTTCS'10*, volume 8 of *LIPIcs*, pages 338–351. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.
6. M. Carbone, M. Nielsen, and V. Sassone. A Calculus of Trust Management. In K. Lodaya and M. Mahajan, editors, *Proc. FSTTCS'04*, volume 3328 of *LNCS*, pages 161–173. Springer, 2004.
7. P.-M. Deniélou and N. Yoshida. Buffered Communication Analysis in Distributed Multiparty Sessions. In P. Gastin and F. Laroussinie, editors, *Proc. CONCUR'10*, volume 6269 of *LNCS*, pages 343–357. Springer, 2010.
8. P.-M. Deniélou and N. Yoshida. Dynamic Multirole Session Types. In M. Sagiv, editor, *Proc. POPL'11*, pages 435–446. ACM, 2011.
9. M. Dezani-Ciancaglini and U. de' Liguoro. Sessions and Session Types: an Overview. In C. Laneve and J. Su, editors, *Proc. WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010.
10. E. Giachino, M. Sackman, S. Drossopoulou, and S. Eisenbach. Softly Safely Spoken: Role Playing for Session Types. Presented at *PLACES '09*, 2009.
11. K. Hoffman, D. Zage, and C. Nita-Rotaru. A Survey of Attack and Defence Techniques for Reputation Systems. *ACM Computing Surveys*, 42:1:1–1:31, 2009.
12. K. Honda. Types for Dyadic Interaction. In E. Best, editor, *Proc. CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.
13. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In G. C. Necula and P. Wadler, editors, *Proc. POPL'08*, pages 273–284. ACM Press, 2008.
14. A. Jøsang and J. Golbeck. Challenges for Robust Trust and Reputation Systems. In T. Dimitrakos and F. Martinelli, editors, *Proc. STM'09*, volume 244 of *ENTCS*. Elsevier, 2009.
15. K. Krukow, M. Nielsen, and V. Sassone. Trust Models in Ubiquitous Computing. *Philosophical Transactions of the Royal Society*, 366:3781–3793, 2008.
16. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. CUP, 1999.
17. V. T. Vasconcelos. Sessions, from Types to Programming Languages. *EATCS Bulletin*, 103:53–73, 2011.
18. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised Multiparty Session Types. In L. Ong, editor, *Proc. FOSSACS'10*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.