Nested Protocols in Session Types

Romain Demangeon and Kohei Honda

Queen Mary, University of London

Abstract. We propose an improvement to session-types, introducing nested protocols, the possibility to call a subprotocol from a parent protocol. This feature adds expressiveness and modularity to the existing session-type theory, allowing arguments to be passed and enabling higher-order protocols definition. Our theory is introduced through a new type system for protocols handling subprotocol calls, and its implementation in a session-calculus. We propose validation and satisfaction relations between specification and implementation. Sound behaviour is enforced thanks to the usage of kinds and well-formedness, allowing us to ensure progress and subject reduction. In addition, we describe an extension of our framework allowing subprotocols to send back results.

1 Introduction

Decentralised computation is becoming more and more popular thanks to the fast growth of web services and other distributed computing technologies. In such a distributed framework, agents (users, servers, applications) are interacting through message-passing communications, without central control. The programmatic coordination of a large number of independent entities interacting with each other inside a network is a challenging task: without global control, the only place where coordination can come from is local endpoints. How can we specify and ensure correctly coordinated behaviour without having any global control? Session types [11] provide a powerful expressive framework to help solving this issue, focusing on the notion of *session* seen as a unit of conversation among participants called *roles*. The expected scenario of the session is described in a global protocol given as a *global type*, projected into end-point specifications called *local types*, describing the behaviour of each role. Those are enforced locally, either through a static analysis of programs (static validation [8, 3, 13]) or at run-time (monitoring [2]). If each agent in the network conforms to its local type, it is guaranteed that their overall interactions conform to the global specification. In the past few years, the theory of session types has been extended in several directions. On the one hand, new features added to the language of the global types allow one to specify more accurately the interactions inside a protocol, for instance by including logical assertions [3], or information flow [4]. This "internal expressiveness" ensure the satisfaction of auxiliary properties: security ("the messages between Alice and Bob cannot be read by Carol"), or governance ("Alice can send a buying request to Bob only if she has enough money on her

bank account"). On the other hand, extensions of the session mechanisms allow greater control over, for example, how participants join or leave a session (through dynamic multiparty session [9] or through a reputation system [10]) or how sessions can be parametrised, increasing "external expressiveness".

Real-world specifications for decentralised networks are large, complex and often highly modular: for example, such specifications are found in many use cases from the development of a large-scale distributed infrastructure for ocean sciences, Ocean Observatories Initiative [15], with whom we are collaborating. Among the use cases in the project, several protocols, used in different contexts, share the same shape. Moreover, some protocols call other protocols. In order to be able to specify, verify, simplify and organise such complex protocol frameworks effectively, solid improvements to the theory of session-types are needed.

In this paper, we present a novel approach to session-types that addresses the structuring principle itself of protocols, increasing both internal and external expressiveness. We introduce *nesting* of protocols, that is, the possibility to define a subprotocol independently of its parent protocol, which calls the subprotocol explicitly. Through a call, arguments can be passed, such as values, roles and other protocols, allowing higher-order description. At the programming level, subprotocols are realised as *subsessions*: one agent creates a new private session, inviting roles of the parent session (*internal invitations*) as well as other agents from the network (*external invitations*). Uninvited participants of the parent session do not have access to the subsession, allowing one to model private interactions inside public sessions. This contrasts with the current use of subprotocols in the protocol description language Scribble [16], where they only correspond to the in-lining mechanism. As an example, we noticed that in several use cases described in [15], a negotiation procedure *Nego* between two agents is invoked inside a main protocol, and other participants do not take part in that negotiation. The *Nego* procedure has its own description, subject to independent modifications and can be invoked in different contexts for different purposes. The theory we propose introduces such modularity in the framework of session types, yielding a solid, formal verification method for distributed programs.

One strong motivation for introducing subprotocols and subsession is that they are a powerful structuring tool: cross-cutting features such as login, negotiations or security controls can be abstracted from the targeted protocols in a compositional way, becoming subprotocols. If we update a login protocol to enforce stronger security checks, specifications of applications using it do not need be updated. Moreover, nesting allows one to call multiple copies of the same protocol with different arguments, improving flexibility and readability. This allows us to clean up and reorganise a large protocol database in [15], by unifying many protocols with the same shape into one parametrised protocol. Another direct benefit from nesting is allowing a better separation of the different branches by inviting participants only when necessary, reducing complexity and resource usage. For instance, in protocol $\mathcal{P}$ involving Alice, Bob and Carol, if Carol interacts only if a certain condition is met, we can have Carol act in a separate subprotocol, inviting Carol only if her presence is required. In our framework, internal

invitations to a subsession are sent *inside the parent-session*, targeting a specific participant through a linear channel. This extends the existing session-calculi where invitations are always done externally, through shared channels.

We propose in Section 2 a syntax for nested protocols, with dedicated constructors for protocol definitions and protocol calls. In order to ensure sound composition, we introduce the notion of *kinds*, "types for types", and define a notion of well-formedness. In Section 3, we describe a session-calculus (based on the π-calculus [14]) handling subsessions. We describe in Section 4 a static validation and a run-time satisfaction, each linking specifications and processes in the session-calculus. We sketch the main properties of typed processes. Finally we propose an extension to our theory in Section 5, allowing a subsession to have a goal: a *result* that is returned to its parent session.

2 Nested Protocols

Global types Throughout the paper, we use G for global types, T for local types, l for communication labels, s, k for session names, a, b for shared channels, ℓ for transition labels, $\mathtt{r}, \mathtt{r}'$ for role identifiers and $\mathcal{P}$ for protocol identifiers. We use v to describe *values*, which can be base type variables (integers, strings, ...), labels or protocol identifiers. x, y are variables, possibly abstracting any value. For any identifier e, we use $\widetilde{e}$ to abstract the sequence $e_1, \ldots, e_n$ of unspecified length n. We use $\mathcal{R}^+$ (resp. $\mathcal{R}^*$) for the reflexive (resp. reflexive-transitive) closure of the relation $\mathcal{R}$. We assume a Barendregt convention for bound variables.

Global types describe protocols from the network point of view: they consist of sequences of interactions between *roles*. We choose the already existing syntax of multi-party session types (e.g., in [3, 1]) as a basis. The syntax for our global types is given by the following grammar:

$$
\begin{array}{rcll}
G & ::= & \mathtt{let}\ \mathcal{P} = \lambda \widetilde{\mathtt{r}}^1, \widetilde{y} \mapsto \mathtt{new}\ \widetilde{\mathtt{r}}^2.G\ \mathtt{in}\ G' & \text{(declaration)} \\
& | & \mathtt{r}\ \mathtt{calls}\ \mathcal{P}\langle \widetilde{\mathtt{r}}, \widetilde{y} \rangle.G & \text{(call)} \\
& | & \mathtt{r}_1 \to \mathtt{r}_2 : \Sigma_{i \in I}\{l_i(x_i).G_i\} & \text{(com)} \\
& | & G_1 \oplus^{\mathtt{r}} G_2 \mid G_1 \parallel G_2 \mid \mu \mathtt{t}.G \mid \mathtt{t} & \text{(choice,par,rec,rec-var)}
\end{array}
$$

Communications between two roles are specified with $\mathtt{r}_1 \to \mathtt{r}_2 : \Sigma_{i \in I}\{l_i(x_i : S_i).G_i\}$, stating that $\mathtt{r}_1$ has a *directed choice* between several labels $\widetilde{l}$ proposed by $\mathtt{r}_2$. Each branch expects a value x_i and executes the continuation G_i. When I is a singleton, we write $\mathtt{r}_1 \to \mathtt{r}_2 : l(x : S)$. Primitive $\oplus^{\mathtt{r}}$ is *located choice*: the choice for one participant $\mathtt{r}$ between two distinct protocol branches. Parallel composition is denoted by $\parallel$ and recursion by the two operators $\mu \mathtt{t}$ and $\mathtt{t}$. We assume a congruence over global types, handling implicit unfolding of recursion.

The new primitives addressing protocol stratification are `let` and `calls`. We describe the declaration of an auxiliary protocol, to be called by a main protocol, by the notation $\mathtt{let}\ \mathcal{P} = \lambda \widetilde{\mathtt{r}}^1, \widetilde{y} \mapsto \mathtt{new}\ \widetilde{\mathtt{r}}^2.G_1\ \mathtt{in}\ G_2$. In this notation, the protocol G_1 is identified by $\mathcal{P}$ in the main protocol G_2. The participants of G_1 are explicitly separated into two groups, $\widetilde{\mathtt{r}}^1$ are internally invited from the parent session and thus given as arguments to $\mathcal{P}$ together with values $\widetilde{y}$; whereas $\widetilde{\mathtt{r}}^2$ are

externally invited from the network at the beginning of G_1. The counterpart of this constructor is the *protocol call* $\mathtt{r}\ \mathtt{calls}\ \mathcal{P}(\widetilde{\mathtt{r}}, \widetilde{v})$ stating that participant $\mathtt{r}$ executes an auxiliary protocol $\mathcal{P}$ with role arguments $\widetilde{\mathtt{r}}$, value arguments $\widetilde{v}$. Note that $\widetilde{v}$ can contain protocol identifiers, thus allowing higher-order programming.

Kinds As protocols can be abstracted, called and used as arguments, we introduce a simple and concise discipline for protocols, which ensures that they are used in an adequate way, *well-formedness*. In order to formalise this notion, we type all objects appearing in specifications with *kinds* (types for types) $K, S ::= \mathtt{Role} \mid \mathtt{Val} \mid \diamond \mid (K_1 \times \cdots \times K_n) \to K$. We use $\mathtt{Val}$ to denote the value-kinds, which are first-order types for values (like $\mathtt{Nat}$ for integers) or data types (such as $\mathtt{Req}$ in Section 2), $\diamond$ to denote protocol type and $\to$ to denote parametrisation. The presence of higher-order calls allows us to treat protocols whose kinds have shapes like $\mathtt{Role} \times (\mathtt{Role} \to \diamond) \to \diamond$, describing a protocol parametrised by a role and another protocol, the latter parametrised by a role. In the following, we will sometimes adopt an *à la Church* notation for protocol constructors, as in $\mathtt{let}\ P = \lambda \mathrm{buyer} : \mathtt{Role}, \mathit{price} : \mathtt{Nat} \mapsto \mathtt{new}\ \widetilde{r}.G\ \mathtt{in}\ G'$, in order to specify the kinds of the arguments passed to a protocol.

We define well-formedness to rule out unsound protocols. For instance, a protocol where $\mathcal{P}$ has kind $\mathtt{Role} \to \diamond$ but is used with kind $\mathtt{Nat} \to \diamond$ is not *well-kinded*. A protocol containing $(\mathtt{r}_1 \to \mathtt{r}_2 : \mathrm{ok}) \oplus^{\mathtt{r}_0} (\mathtt{r}_2 \to \mathtt{r}_1 : \mathrm{ko})$ is not *projectable* as $\mathtt{r}_1$ has no mean to know which branch $\mathtt{r}_0$ chooses, and thus is not able to know if it must perform an input or an output. In order to define projectability, which ensures that a global type can be coherently projected into local types, we define the *restriction* of a protocol to a role, noted $(G)|_{\mathtt{r}}$ as the global type obtained by removing every constructor of G where $\mathtt{r}$ does not appear. A protocol is projectable if for every choice (directed or located), the difference between the branches are only visible to the roles involved in that choice.

Definition 1 (Well-Formedness)
A global-type G is well-kinded *if there exists τ from all identifiers of G to types satisfying, for all subprotocols of G:*

1. $\mathtt{let}\ \mathcal{P} = (\lambda\widetilde{y} \mapsto \mathtt{new}\ \widetilde{r}\cdot)\ \mathtt{in}\ \cdot$ *is s.t.* $\tau(\mathcal{P}) = \tau(\widetilde{y}) \to \diamond$, *and* $\forall i, \tau(r_i) = \mathtt{Role}$.

2. $\mathtt{r}\ \mathtt{calls}\ \mathcal{P}\langle\widetilde{y}\rangle.G$ *is s.t.* $\tau(\mathtt{r}) = \mathtt{Role}$, $\tau(\mathcal{P}) = \tau(\widetilde{y}) \to \diamond$.

3. for all identifiers $\mathtt{r}$ *in* $\mathtt{r} \to \mathtt{r}' : l(x : S).G$ *and* $G_1 \oplus^{\mathtt{r}} G_2$, $\tau(\mathtt{r}) = \mathtt{Role}$.

A global type G is projectable *if:*

1. for each subterm of G of the form $G_1 \oplus^{\mathtt{r}_0} G_2$, for any free $\mathtt{r} \neq \mathtt{r}_0$, $(G_1)|_{\mathtt{r}} = (G_2)|_{\mathtt{r}}$.

2. for each subterm of G of the form $\mathtt{r} \to \mathtt{r}' : \Sigma_{i \in I}\{l_i(x_i : S_i).G_i\}$, for any role free role $\mathtt{r} \notin \{\mathtt{r}, \mathtt{r}'\}$ and for all $\{i, j\} \subseteq I$, $(G_i)|_{\mathtt{r}} = (G_j)|_{\mathtt{r}}$.

A protocol G is well-formed *when it is well-kinded and projectable, and satisfies the standard linearity condition [1].*

There exists in [9, 12] *mergeability conditions* that allows the authors to be less restrictive in the definition of projectability. Our framework could accommodate this refinement. We do not present it here, for the sake of clarity.

Motivating Examples In this section, we motivate our contribution with three examples extracted from concrete specifications and illustrate higher-order programming with a fourth one.

Resource usage The following example is inspired by the use cases (UC R2.34, UC R2.32) from the OOI project [15]. A negotiation procedure *Nego* is first defined independently, to be used in several different protocols. This negotiation procedure involves two participants trying to agree on a contract: first participant specifies a request, second participant offers a corresponding contract, then both participants enter a loop when the first one can either accept the contract, which ends the protocol or make a counter-offer.

```
let Nego = λr1, r2 ↦
        r1 → r2 : ask(terms).
        μt.
        r2 → r1 : proposition(contract2).
        r1 → r2 : {accept.end
                   counter(contract1).t}
in
client → agent : request(coord). agent → instr : connect
instr → agent : available. agent → client : ack.
agent calls Nego(agent, client).
μt.
   client → instr : {abort(coord).end
                     command(code).
                        instr → client : result(data).t}
```

The main protocol *UseRes* consists of several interactions between three participants (client, agent, instr), processed in the following order: first client sends a request to agent for an instrument he wants to use, agent tries to connect to instr which acknowledges when available. Then, agent negotiates a contract with client (by calling protocol *Nego*). After a successful negotiation, client and instr interact inside a loop, the client sending commands and receiving data. The negotiation phase is considered external: should the auxiliary protocol be modified, for instance to enforce another negotiation policy, the main protocol would remain the same.

Client-Middleware-Server The protocol CMS, presented of the left side of Figure 1 describing a typical service interaction. This protocol initially involves two participants, client starts the interactions by sending a request to the middleware middle. If the latter is able to treat the request directly, it answers to client, if not, it contacts server, calling subprotocol $Contact$ with itself as role argument. In the subprotocol, middle performs an external invitation of server, forwards the request and waits for an answer. After the subprotocol is completed, the answer is forwarded to the client. Nesting, in this example, allows us to invite server to participate only when necessary: if middle can treat the request, server is not even invited. Using subprotocols in such a way allows us to cut a great deal of unnecessary traffic caused by external invitations, saving bandwidth.

Dynamic distribution We then describe on the right side of Figure 1 a third example, inspired by a concrete protocol from the Array Network Facility, used

```
let Contact = λagent, req ↦
        new server.
        agent → server : request(req).
        server → agent : answer(ans).
        end
in
client → middle : request(req0).
        (middle → client : answer(ans0).end)
⊕middle
        (middle calls Contact(middle, req0).
        middle → client : answer(ans0).end)
```

```
let Treat =
λr1, r2 ↦
        new worker.
        r1 → worker : raw(data).
        worker → r2 : processed(data).
        end
in
parallel_n(
        source calls Treat(source, target)
        ).
end
```

Fig. 1. Protocols CMS and ANF

for processing seismic data. Here, the operator $\texttt{parallel}_n(G)$ is used as a shortcut for n parallel copies of the protocol G. In this protocol, data comes in a raw state from a participant source and should reach participant target processed. In the body of each of the n parallel executions, source calls the subprotocol $Treat$ inviting a new participant worker and using it to process data. This protocol is run in networks where many computing units can accept temporarily the worker role. In this example, stratification is used to present in a clean way the execution of thousands of copies of the same protocol. As each copy is implemented by a different session, the different calls to the subprotocol are actually independent from each other.

Marketplace Finally, we propose a protocol for a virtual marketplace in which participants have the possibility to engage in trade actions with other participants. General protocols Buy and $Sell$ are defined to handle these buying and selling. The encounter between two agents follows the same procedure (handshake, authentication, possibility to cancel the transaction) whatever the reason of their meeting is. This common procedure is abstracted in $Meet$ and a protocol identifier *Action* is given as an argument to $Meet$ calls, meant to be substituted by Buy or $Sell$ (or any similar protocol). Thus, $Meet$ is an higher-order protocol, parametrised with protocol *Action*.

```
let Buy = λ agent : Role, seller : Role, item : Tradable ↦ ...
in let Sell = λ agent : Role, buyer : Role, item : Tradable ↦ ...
in let Meet = λ agent : Role, partner : Role,
                    item : Tradable, Action : (Role → Role → Tradable → ◇) ↦ ...
agent calls Action⟨partner, item⟩ ...
in ...
alice calls Meet⟨bob, kettle, Buy⟩. carol calls Meet⟨bob, teacup, Sell⟩ ...
```

The protocols presented in this section are well-formed: notice that protocol CMS is projectable, in each branch of the choice $\oplus^{\text{middle}}$, the restriction on client is "middle → client : answer.end". Kinds for subprotocols presented in the examples are as follows: $Nego$: Role × Role → ◇, $Contact$: Role × Req → ◇, $Treat$: Role × Role → ◇, $Buy, Sell$: Role × Role × Tradable → ◇, $Meet$: Role × Role × Tradable × (Role × Role × Tradable → ◇) → ◇

Local types and Projection Local types describe a global conversation from the partial point-of-view of a participant and are used to validate and monitor distributed programs. Their syntax is given by:

$$
\begin{array}{rcl}
T & ::= & \texttt{get}[\mathbf{r}]?_{i\in I}\{l_i(x_i : S_i).T_i\} \quad | \quad \texttt{send}[\mathbf{r}]!_{i\in I}\{l_i(x_i : S_i).T_i\} \\
& | & T \parallel T \quad | \quad T \oplus T \quad | \quad \mu \mathbf{t}.T \quad | \quad \mathbf{t} \quad | \quad \texttt{end} \\
& | & \texttt{call}\ \mathcal{P} : G\ \texttt{with}\ (\widetilde{v}\ \texttt{as}\ \widetilde{y} : \widetilde{S})\&(\widetilde{\mathbf{r}}^2).T \\
& | & \texttt{ent}\ \mathcal{P}[\mathbf{r}]\langle\widetilde{v}\rangle\ \texttt{from}\ \mathbf{r}.T \quad | \quad \texttt{req}\ \mathcal{P}[\mathbf{r}]\langle\widetilde{v}\rangle\ \texttt{to}\ \mathbf{r}.T
\end{array}
$$

Creating a subsession for protocol $\mathcal{P}$ having global type G is specified by $\texttt{call}\ \mathcal{P} : G\ \texttt{with}\ (\widetilde{v}\ \texttt{as}\ \widetilde{y} : \widetilde{S})\&(\widetilde{\mathbf{r}}^2)$, with $\widetilde{v}$ as value arguments and involving external invitations for roles $\widetilde{\mathbf{r}}^2$. Internal invitations are handled using two specific constructors, as they are meant to be performed on the parent session channel: `ent` specifies the act of accepting such an invitation, `req` specifies the dual action. Syntax contains endpoint primitives for communications, specified by `get` for the receiver side and `send` for the sender side, as well as constructors for parallel, choice and recursion. We handle equivalence of types through recursions and parallel compositions implicitly. In the following, we omit trailing occurrences of `end`.

Projection from global to local types is defined w.r.t. a protocol environment, associating protocols identifiers to their contents. Environment is updated by `let in` constructors. We present below the projection rule for `call` and `let`. For the former the result of the projection depends on the participant we project on, $\mathbf{r}_{\text{proj}}$. If it is the subprotocol initiator $\mathbf{r}^{\mathcal{A}}$ it is responsible for creating the subsession (`call`) and sending the internal invitations (`req`). If it participates in the subprotocol, it has to accept an internal invitation (`ent`). Projection on other constructors is standard.

$$
(\texttt{let}\ \mathcal{P} = \lambda\widetilde{\mathbf{r}}^1.\widetilde{y} \mapsto \texttt{new}\ \widetilde{\mathbf{r}}^2.G_{\mathcal{P}}\ \texttt{in}\ G') \Downarrow^{\mathbf{Env}}_{\mathbf{r}^p} = G' \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathbf{r}}^1,\widetilde{y};\widetilde{\mathbf{r}}^2;G_{\mathcal{P}})}_{\mathbf{r}^p}
$$

$$
(\mathbf{r}^A\ \texttt{calls}\ \mathcal{P}(\widetilde{\mathbf{r}}^0,\widetilde{v}).G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathbf{r}}^1,\widetilde{y};\widetilde{\mathbf{r}}^2;G_{\mathcal{P}})}_{\mathbf{r}^p} =
$$

$$
\begin{cases}
\text{if } \mathbf{r}^p = \mathbf{r}^A,\ \mathbf{r}^A \notin \widetilde{\mathbf{r}^0} \\
\quad \texttt{call}\ \mathcal{P} : G_{\mathcal{P}}\ \texttt{with}\ (\widetilde{v}\ \texttt{as}\ \widetilde{y})\&(\widetilde{\mathbf{r}}^2).[(G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathbf{r}}^1,\widetilde{y};\widetilde{\mathbf{r}}^2;G_{\mathcal{P}})}_{\mathbf{r}^p} \\
\quad \parallel \texttt{req}\ \mathcal{P}[\mathbf{r}_0^1]\langle\widetilde{v}\rangle\ \texttt{to}\ \mathbf{r}_0^0 \parallel \cdots \parallel \texttt{req}\ \mathcal{P}[\mathbf{r}_n^1]\langle\widetilde{v}\rangle\ \texttt{to}\ \mathbf{r}_n^0] \\
\text{if } \mathbf{r}^p = \mathbf{r}^A \text{ and } \mathbf{r}^A = \mathbf{r}_i^0 \\
\quad \texttt{call}\ \mathcal{P} : G_{\mathcal{P}}\ \texttt{with}\ (\widetilde{v}\ \texttt{as}\ \widetilde{y})\&(\widetilde{\mathbf{r}}^2).[(G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathbf{r}}^1,\widetilde{y};\widetilde{\mathbf{r}}^2,G_{\mathcal{P}})}_{\mathbf{r}^p} \\
\quad \parallel \texttt{ent}\ \mathcal{P}[\mathbf{r}_i^1]\langle\widetilde{v}\rangle\ \texttt{from}\ \mathbf{r}^A\ \parallel \texttt{req}\ \mathcal{P}[\mathbf{r}_0^1]\langle\widetilde{v}\rangle\ \texttt{to}\ \mathbf{r}_0^0 \parallel \cdots \parallel \texttt{req}\ \mathcal{P}[\mathbf{r}_n^1]\langle\widetilde{v}\rangle\ \texttt{to}\ \mathbf{r}_n^0] \\
\text{if } \mathbf{r}^p \neq \mathbf{r}^A \text{ and } \mathbf{r}^p = \mathbf{r}_i^0 \\
\quad \texttt{ent}\ \mathcal{P}[\mathbf{r}_i^1]\langle\widetilde{v}\rangle\ \texttt{from}\ \mathbf{r}^A.(G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathbf{r}}^1,\widetilde{y};\widetilde{\mathbf{r}}^2;G_{\mathcal{P}})}_{\mathbf{r}^p} \\
\text{Otherwise} \\
\quad (G) \Downarrow^{\mathbf{Env},\mathcal{P}\mapsto(\widetilde{\mathbf{r}}^1,\widetilde{y};\widetilde{\mathbf{r}}^2;G_{\mathcal{P}})}_{\mathbf{r}^p}
\end{cases}
$$

If the initiator $\mathbf{r}^{\mathcal{A}}$ of the subsession also takes part in it, the projection on $\mathbf{r}^{\mathcal{A}}$ specifies that it invites itself. It is easy to add to our language a dedicated constructor handling session-invitation directly, without inducing communication at the network level. For the sake of clarity, we do not include such a constructor in this paper.

We present below the projection of CMS on its two roles. G_{CMS} is the global type of the whole protocol and G_C the global type of $Contact$.

$$
\begin{array}{ll}
G_{CMS} \Downarrow^{\emptyset}_{\text{client}} & = \texttt{send}[\text{middle}]!\{\text{request}(req_0)\}.\texttt{get}[\text{middle}]?\{\text{answer}(ans_0)\} \\
G_{CMS} \Downarrow^{\emptyset}_{\text{middlew}} & = \texttt{get}[\text{client}]?\{\text{request}(req_0)\}. \\
 & \quad \texttt{send}[\text{client}]!\{\text{answer}(ans_0)\} \\
 & \quad \oplus \\
 & \quad (\texttt{call}\ Contact : G_C\ \texttt{with}\ (req_0\ \texttt{as}\ req : \texttt{Req})\&(\text{server}). \\
 & \quad\ \ (\texttt{req}\ Contact[\text{agent}]\langle req\rangle\ \texttt{to}\ \text{middle} \parallel \\
 & \quad\ \ \texttt{ent}\ Contact[\text{agent}]\langle req\rangle\ \texttt{from}\ \text{middle} \parallel \\
 & \quad\ \ \texttt{send}[\text{client}]!\{\text{answer}(ans_0)\}))
\end{array}
$$

3 Session-Calculus

Our session-calculus, based on the π-calculus [14], contains usual primitives from existing session-calculi [3], as well as dedicated primitives for session creation and internal (on-session) invitations. Names are divided into shared channels a, b, u (standard π-names) and session channels s, k. The former are used to send and receive external invitations, the latter to handle all session interactions.

$$
\begin{array}{rcl}
P & ::= & \mathbf{0} \mid P|P \mid a(x).P \mid \overline{a}\langle s\rangle.P \mid P + P \\
 & \mid & k?[\mathtt{r},\mathtt{r}]_{i\in I}\{l_i(x_i).P_i\} \mid k![\mathtt{r},\mathtt{r}]l\langle v\rangle.P \mid (\nu u)\ P \\
 & \mid & \texttt{new}\ s\ \texttt{on}\ s\ \texttt{with}\ (\widetilde{v})\&(\widetilde{a}\ \texttt{as}\ \widetilde{\mathtt{r}}).P \\
 & \mid & s \downarrow [\mathtt{r}, \mathtt{r} : \mathtt{r}](x).P \mid s \uparrow [\mathtt{r}, \mathtt{r} : \mathtt{r}]\langle s\rangle.P \mid \mu X(x).P\langle v\rangle \mid X\langle v\rangle
\end{array}
$$

We denote by $k?[\mathtt{r}_1, \mathtt{r}_2]_{i\in I}\{l_i(x_i).P_i\}$ a branching input on session k from $\mathtt{r}_1$ to $\mathtt{r}_2$, with continuations $(P_i)_{i\in I}$. The dual primitive is $k![\mathtt{r}_1, \mathtt{r}_2]l\langle v\rangle.P$. Creation of a subsession is done with $\texttt{new}\ s\ \texttt{on}\ k\ \texttt{with}\ (\widetilde{v})\&(\widetilde{a}\ \texttt{as}\ \widetilde{\mathtt{r}}^2)$ with s being the subsession, k the parent session, $\widetilde{v}$ the arguments and $\widetilde{a}$ the channels on which the external invitations are sent. Operator $s \downarrow [\mathtt{r}^1, \mathtt{r}^2 : \mathtt{r}^3](x).P$ is the action of waiting on s for an internal invitation sent by $\mathtt{r}^1$ to $\mathtt{r}^2$ in order to play role $\mathtt{r}^3$ in a subsession x. Finally, $s \uparrow [\mathtt{r}^1, \mathtt{r}^2 : \mathtt{r}^3]\langle s\rangle.P$ is its dual action. Inputs and outputs on shared channels, choice, parallel composition and inactive process $\mathbf{0}$ are inherited from the π-calculus. We omit trailing occurrences of $\mathbf{0}$. Structural congruence $\equiv$ for processes is defined in the usual way.

Semantics is given by reduction rules below, defined w.r.t. a notion of evaluation context $\mathbf{E} ::= [\,] \mid P \mid \mathbf{E} \mid (\nu a)\ \mathbf{E}$. The crucial rule of our system is (**subs**) where a session creation operator `new` is destructed in order to create external invitations on shared channels. (**join**) handles internal session invitation, other rules are standard:

(**comS**) $$\overline{\mathbf{E}[s![\mathtt{r}_1,\mathtt{r}_2]l_j\langle\widetilde{v}\rangle.P \mid s?[\mathtt{r}_1,\mathtt{r}_2]_{i\in I}\{l_i(\widetilde{x_i}).P_i\}] \to \mathbf{E}[P \mid P_j\{\widetilde{v}/\widetilde{x_j}\}]}$$

(**comC**) $$\overline{\mathbf{E}[\overline{a}\langle\widetilde{v}\rangle.P \mid a(\widetilde{y}).Q] \to \mathbf{E}[P \mid Q\{\widetilde{v}/\widetilde{y}\}]}$$

(**subs**) $$\frac{\widetilde{\mathtt{r}}^2 = (\mathtt{r}_1^2, \ldots, \mathtt{r}_n^2) \qquad \widetilde{a} = (a_1, \ldots, a_n)}{\mathbf{E}[\texttt{new}\ s\ \texttt{on}\ k\ \texttt{with}\ (\widetilde{v})\&(\widetilde{a}\ \texttt{as}\ \widetilde{\mathtt{r}}^2).P] \to \mathbf{E}[P \mid \overline{a_1}\langle s[\mathtt{r}_1^2]\rangle \mid \ \ldots\ \mid \overline{a_n}\langle s[\mathtt{r}_n^2]\rangle]}$$

(**join**) $$\overline{\mathbf{E}[s \uparrow [\mathtt{r}, \mathtt{r}' : \mathtt{r}'']\langle k\rangle.P \mid s \downarrow [\mathtt{r}, \mathtt{r}' : \mathtt{r}''](x).Q] \to \mathbf{E}[P \mid Q\{k/x\}]}$$

(**choice**) $$\frac{P_i \to P_i'}{\mathbf{E}[(P_1 + P_2)] \to \mathbf{E}[P_i']}$$

As an example consider the following processes:

$$
\begin{aligned}
P_{\text{alice}} &= a(x).x![\text{client}, \text{middle}]\text{request}(\text{“kettle”}).x?[\text{middle}, \text{client}]\text{answer}(ans_0) \\
P_{\text{bob}} &= \overline{a}\langle s\rangle.s?[\text{client}, \text{middle}]\text{request}(req_0). \\
&\qquad (s![\text{middle}, \text{client}]\text{answer}(ans_0) \\
&\quad + (\texttt{new}\ k\ \texttt{on}\ s\ \texttt{with}\ (req_0)\&(c\ \texttt{as}\ \text{server}). \\
&\qquad s\uparrow[\text{middle}, \text{middle} : \text{agent}]\langle k\rangle \mid s\downarrow[\text{middle}, \text{middle} : \text{agent}](z). \\
&\qquad z![\text{agent}, \text{server}]\text{request}\langle req_0\rangle.z?[\text{server}, \text{agent}]\text{answer}(ans_r). \\
&\qquad s![\text{middle}, \text{client}]\text{answer}\langle ans_r\rangle) \\
P_{\text{carol}} &= c(y).y?[\text{agent}, \text{server}]\text{request}(req).y![\text{server}, \text{agent}]\text{answer}(ans)
\end{aligned}
$$

P_{alice}, P_{bob} and P_{carol} are processes ready to play, respectively roles client, middle and server in the CMS protocol. P_{alice} (resp. P_{carol}) is a simple process, ready to accept an external invitation to the parent session on a (resp. to the child session on c) and to behave as expected. P_{bob} is more complex: it sends an invitation on a, and after receiving a request it chooses, as specified in Figure 1, between answering directly on the session channel or contacting the server through a subsession. In this case, the new session channel k is created and one internal invitation to play role agent in k is sent and accepted by P_{bob} itself, then it proceeds as expected. We describe a reduction sequence for the composition of these three processes:

$$
\begin{aligned}
P_{\text{alice}} \mid P_{\text{carol}} \mid P_{\text{bob}} \rightarrow\rightarrow \quad & s?[\text{middle}, \text{client}]\text{answer}(ans_0) \mid P_{\text{carol}} \\
& \mid (\dots) + (\texttt{new}\ k\ \texttt{on}\ s\ \texttt{with}\ (req_0)\&(c\ \texttt{as}\ \text{server})\dots) \\
\rightarrow \quad & s?[\text{middle}, \text{client}]\text{answer}(ans_0) \mid P_{\text{carol}} \mid \overline{c}\langle k\rangle \\
& \mid s\uparrow[\text{middle}, \text{middle} : \text{agent}]\langle k\rangle \mid s\downarrow[\text{middle}, \text{middle} : \text{agent}](z)\dots \\
\rightarrow \quad & s?[\text{middle}, \text{client}]\text{answer}(ans_0) \mid P_{\text{carol}} \\
& \mid \overline{c}\langle k\rangle \mid k![\text{middle}, \text{server}]\text{request}(req_0)\dots \\
\rightarrow \quad & s?[\text{middle}, \text{client}]\text{answer}(ans_0) \\
& \mid k![\text{agent}, \text{server}]\text{request}(req_0)\dots \\
& \mid k?[\text{agent}, \text{server}]\text{request}(req)\dots \quad \rightarrow\rightarrow\rightarrow \quad \mathbf{0}
\end{aligned}
$$

After two communications on a and s, the reduct of P_{bob} reaches the located choice. We suppose it chooses the second branch. Thus, an output on c containing session name k is created. Then the internal self-invitation for k is performed, and, finally, the external invitation of P_{carol} on c. Three reductions can still be played, two on k and one on s.

Validation We describe a static way to ensuring that processes conforms to formal specifications. The *global environment* Γ relates shared channels to the type of the invitation they carry, protocol names to their code and session channels to the global type they implement. $a : T[\mathtt{r}]$ means that a is used to send and receive invitations to play role $\mathtt{r}$ with local type T, $\mathcal{P} : (\widetilde{\mathtt{r}}^1, \widetilde{y}; \widetilde{\mathtt{r}}^2; G_{\mathcal{P}})$ describes the participants, arguments and code of protocol $\mathcal{P}$, finally, $s : G$ means that protocol G can be implemented on s. The *session environment* Δ relates pairs of session channels and roles $s[\mathtt{r}]$ to local types. $s[\mathtt{r}] : T$ means that in session s, participant $\mathtt{r}$ still has to perform the actions of T. $s[\mathtt{r}]^\bullet : T$ (resp. $s[\mathtt{r}]^\circ : T$) stands for the *capability* to invite externally (resp. internally) someone to play role $\mathtt{r}$ in s. In the following, we consider only environments which are mappings, and we will write $\Delta(s[\mathtt{r}])$. Additionally, we write $\Delta(s) = 0$ when s does not appear in Δ and $s[\mathtt{r}]^-$ to denote either $s[\mathtt{r}]^\circ$, $s[\mathtt{r}]^\bullet$ or $s[\mathtt{r}]$. We allow "garbage collection" for session environment: $(\Delta, s[\mathtt{r}] : \mathtt{end}) = \Delta$.

$$
\begin{array}{lcl}
\Gamma & ::= & \emptyset \mid \Gamma, a : T[\mathtt{r}] \mid \Gamma, \mathcal{P} : (\widetilde{\mathtt{r}}^1, \widetilde{y}; \widetilde{\mathtt{r}}^2; G_{\mathcal{P}}) \mid \Gamma, s : G \\
\Delta & ::= & \emptyset \mid \Delta, s[\mathtt{r}] : T \mid \Delta, s[\mathtt{r}]^\bullet : T \mid \Delta, s[\mathtt{r}]^\circ : T
\end{array}
$$

A typing judgement $\Gamma \vdash P \triangleright \Delta$ means that under the global environment Γ, the process P is validated by the session environment Δ. We use $\vdash v : S$ to notify that value v has kind S. The validation rules are as follows:

$$(\mathbf{I},\mathbf{O})\ \frac{\Gamma \vdash P \triangleright \Delta, x[\mathtt{r}] : T \qquad \Gamma(a) = T[\mathtt{r}]}{\Gamma \vdash a(x).P \triangleright \Delta} \qquad \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma(a) = T[\mathtt{r}]}{\Gamma \vdash \overline{a}\langle s\rangle.P \triangleright \Delta, s[\mathtt{r}]^\bullet : T}$$

$$(\mathbf{C})\ \frac{(\Gamma \vdash P_i \triangleright \Delta, s[\mathtt{r}'] : T_i \qquad \vdash y_i : S_i)_{i\in I}}{\Gamma \vdash s?[\mathtt{r},\mathtt{r}']_{i\in I}\{l_i(y_i).P_i\} \triangleright \Delta, s[\mathtt{r}'] : \mathtt{get}[\mathtt{r}]?_{i\in I}\{l_i(x_i : S_i).T_i\}}$$

$$(\mathbf{S})\ \frac{\Gamma \vdash P \triangleright \Delta, s[\mathtt{r}] : T_j \qquad \vdash v : S_j}{\Gamma \vdash s![\mathtt{r},\mathtt{r}']l_j\langle v\rangle.P \triangleright \Delta, s[\mathtt{r}] : \mathtt{send}[\mathtt{r}']!_{i\in I}\{l_i(x_i : S_i).T_i\}}$$

$$(\mathbf{P})\ \frac{\Gamma \vdash P \triangleright \Delta, s[\mathtt{r}] : T \qquad \Gamma(\mathcal{P}) = (\widetilde{\mathtt{r}}^1, \widetilde{y}; \widetilde{\mathtt{r}}^2; G) \qquad G\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}''} = T''}{\Gamma \vdash s \uparrow [\mathtt{r},\mathtt{r}' : \mathtt{r}'']\langle k\rangle.P \triangleright \Delta, s[\mathtt{r}] : \mathtt{req}\ \mathcal{P}[\mathtt{r}'']\langle\widetilde{v}\rangle\ \mathtt{to}\ \mathtt{r}'.T, k[\mathtt{r}'']^\circ : T''}$$

$$(\mathbf{J})\ \frac{\Gamma \vdash P \triangleright \Delta, s[\mathtt{r}'].T, x[\mathtt{r}''] : T'' \qquad \Gamma(\mathcal{P}) = (\widetilde{\mathtt{r}}^1, \widetilde{y}; \widetilde{\mathtt{r}}^2; G) \qquad G\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}''} = T''}{\Gamma \vdash s \downarrow [\mathtt{r},\mathtt{r}' : \mathtt{r}'']\langle x\rangle.P \triangleright \Delta, s[\mathtt{r}'] : \mathtt{ent}\ \mathcal{P}[\mathtt{r}'']\langle\widetilde{v}\rangle\ \mathtt{from}\ \mathtt{r}.T}$$

$$(\mathbf{New})\ \frac{\begin{array}{c}\Gamma \vdash P \triangleright \Delta, s[\mathtt{r}] : T, k[\mathtt{r}_1^1]^\circ : T'_1, \ldots, k^\circ[\mathtt{r}_n^1] : T'_n, k^\bullet[\mathtt{r}_1^2] : T'_{n+1}, \ldots, k^\bullet[\mathtt{r}_m^2] : T'_{n+m} \\ \Gamma(\mathcal{P}) = (\widetilde{\mathtt{r}}^1, \widetilde{y}; \widetilde{\mathtt{r}}^2; G) \qquad \forall i, \Gamma(a_i) = T'_{i+n}[\mathtt{r}_{i+n}] \\ \forall i, G\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}_i^1} = T'_i \qquad \forall j, G\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}_j^2} = T'_{j+n} \qquad \vdash \widetilde{v} : S \qquad \Gamma(k) : \mathcal{P}\{\widetilde{v}/\widetilde{y}\}\end{array}}{\Gamma \vdash \mathtt{new}\ k\ \mathtt{on}\ s\ \mathtt{with}\ (\widetilde{v})\&(\widetilde{a}\ \mathtt{as}\ \widetilde{\mathtt{r}}^2).P \triangleright \Delta, s[\mathtt{r}] : \mathtt{call}\ \mathcal{P} : G\ \mathtt{with}\ (\widetilde{v}\ \mathtt{as}\ \widetilde{y} : \widetilde{S})\&(\widetilde{\mathtt{r}}^2).T}$$

$$(\mathbf{N},\mathbf{P})\ \frac{}{\Gamma \vdash \mathbf{0} \triangleright \emptyset} \qquad \frac{\Gamma \vdash P_1 \triangleright \Delta_1 \qquad \Gamma \vdash P_2 \triangleright \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \otimes \Delta_2}$$

$$(\mathbf{S1})\ \frac{\Gamma \vdash P_1 \triangleright \Delta, s[\mathtt{r}] : T_1 \qquad \Gamma \vdash P_2 \triangleright \Delta, s[\mathtt{r}] : T_2}{\Gamma \vdash P_1 + P_2 \triangleright \Delta, s[\mathtt{r}] : T_1 \oplus T_2}$$

$$(\mathbf{S2},\mathbf{R})\ \frac{\Gamma \vdash P \triangleright \Delta, s[\mathtt{r}] : T_i \qquad i \in \{1,2\}}{\Gamma \vdash P \triangleright \Delta, s[\mathtt{r}] : T_1 \oplus T_2} \qquad \frac{\Gamma, a : T[\mathtt{r}] \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)\ P \triangleright \Delta}$$

Rule (**New**) is the crux of this type system, as it ensures subsessions are called in a sound way. To type the process $\mathtt{new}\ k\ \mathtt{on}\ s\ \mathtt{with}\ (\widetilde{v})\&(\widetilde{a}\ \mathtt{as}\ \widetilde{\mathtt{r}}^2).P$, the session channel k should be associated with a protocol $G\{\widetilde{v}/\widetilde{y}\}$ matching the one present in the local type of $\mathtt{r}$ in the parent session s: $\mathtt{call}\ \mathcal{P} : G\ \mathtt{with}\ (\widetilde{v}\ \mathtt{as}\ \widetilde{y} : \widetilde{S})\&(\widetilde{\mathtt{r}}^2).T$ and global environment Γ should map $\mathcal{P}$ to $(\widetilde{\mathtt{r}}^1, \widetilde{y}; \widetilde{\mathtt{r}}^2; G)$. The endpoint projections $(T_p)_{1\leq p\leq n+m}$ of $\mathcal{P}$ are divided into two sets, the ones that correspond to roles $(\mathtt{r}_i^1)_{1\leq i\leq n}$ internally invited, and the ones that correspond to roles $(\mathtt{r}_j^2)_{1\leq j\leq m}$ externally invited through $\widetilde{a}$. Capabilities $(k[\mathtt{r}_i]^\circ : T_i)_i, (k[\mathtt{r}_j]^\bullet : T_j)_j$ for both types of invitations are given to the continuation process P. Rule (**P**) types a process whose role $\mathtt{r}$ on session s consists in sending a internal invitation to play role $\mathtt{r}''$ in session k. The process is required to hold the capability for $k[\mathtt{r}'']$, we ensure it corresponds to the type of the invitation. Its counterpart (**J**) ensures that role $\mathtt{r}$ of session s after receiving an invitation for $k[\mathtt{r}'']$, gets the corresponding local type T'' in its Δ. Rules (**I**) and (**O**) handle external invitations. As in the internal case, we ensure that the sending process has the corresponding capability. Rules (**C**) and (**S**) address branching communications on session channels. In both rules we ensure that the values communicated x_i, v have the same value-type as the identifiers y_i in the type. Summations are handled by two rules (**S1**) and (**S2**). If the local type specifies a choice between two branches, the process can either implement this choice with the + constructor, or implement only one branch of the choice. This illustrates the fact that the decision can be made at implementation time (for instance a middleware implementing CMS

which always contacts the server) or at run-time (a middleware which can proceed both ways according to the request). Rule (**Pa**) requires a small explanation, as it allows one to split local types into two branches. We define the $\otimes$ operator with $\Delta_1 \otimes \emptyset = \Delta_1$, $\Delta_1 \otimes (\Delta_2, s[\mathtt{r}]^- : T) = (\Delta_1, s[\mathtt{r}]^- : T) \otimes \Delta_2$ if $\Delta_1(s[\mathtt{r}]) = 0$ and $(\Delta_1, s[\mathtt{r}]^- : T_1) \otimes (\Delta_2, s[\mathtt{r}]^- : T_2) = (\Delta_1, s[\mathtt{r}]^- : T_1 \parallel T_2) \otimes \Delta_2$. Thus, when splitting the session environment in a parallel constructor, we allow the splitting of a single local type composed of two parallel subtypes. Finally, rule (**N**) specifies that the session environment should be empty to type $\mathbf{0}$. This ensures that the processes eventually complete the local types of their specification.

Following the typing rules, one can type the processes introduced in Section 3 as follows: $\Gamma \vdash P_{\text{alice}} \triangleright \emptyset$, $\Gamma \vdash P_{\text{bob}} \triangleright s[\text{middle}] : T_{\text{middle}}, s[\text{client}]^\bullet : T_{\text{client}}$, $\Gamma \vdash P_{\text{carol}} \triangleright \emptyset$ with $\Gamma = a : T_{\text{client}}, c : T_{\text{server}}, Contact : (\text{agent}, \text{req}; \text{server}; G_{Contact}), s : CMS, k : Contact, G_{Contact} \Downarrow^{\emptyset}_{\text{server}} = T_{\text{server}}$, and $T_{\text{client}} = G_{CMS} \Downarrow^{\emptyset}_{\text{client}}, T_{\text{middle}} = G_{CMS} \Downarrow^{\emptyset}_{\text{middle}}$ as defined in Section 2.

Session environments for P_{alice} and P_{carol} are empty: processes are not bound to do anything as long as they did not receive an invitation. Session environment for P_{bob} contains both the local type for the role middle played by the process and the capability to send an external invitation for client in the same session. The capability to send an external invitation to server is not created yet.

4 Properties

In this section we justify our theory with two main propositions, subject reduction and progress. First, we define a satisfaction relation relating dynamically processes and specifications. We introduce Labelled Transition Systems for both the processes and the specification. Labels are defined by $\ell ::= \quad \tau \mid \overline{a}\langle v \rangle \mid a\langle v \rangle \mid s?[\mathtt{r}, \mathtt{r}']l\langle k \rangle \mid s![\mathtt{r}, \mathtt{r}']l\langle k \rangle \mid s \downarrow [\mathtt{r}, \mathtt{r}' : \mathtt{r}'']\langle k \rangle \mid s \uparrow [\mathtt{r}, \mathtt{r}' : \mathtt{r}'']\langle k \rangle$ The subject of a label $\mathbf{sbj}(\ell)$ is defined intuitively for all labels, knowing that $\mathbf{sbj}(\tau) = 0$. Labels $\overline{a}\langle v \rangle, a\langle v \rangle, s![\mathtt{r}, \mathtt{r}']l\langle k \rangle$, $s \uparrow [\mathtt{r}, \mathtt{r}' : \mathtt{r}'']\langle k \rangle$ and τ (resp. $s?[\mathtt{r}, \mathtt{r}']l\langle k \rangle$ and $s \downarrow [\mathtt{r}, \mathtt{r}' : \mathtt{r}'']\langle k \rangle$) are denoted as *output labels* (resp. *input labels*). In the satisfaction relation defined below, output labels are the ones played by the process, to which the specification must answer (thus τ and $a\langle v \rangle$ are considered outputs), and the input labels are the ones the specification plays, to which the process must answer. Transitions for processes $P \xrightarrow{\ell} P'$ follow the reduction semantics. The most relevant transitions for specifications, defined w.r.t. a global environment Γ, are presented in Figure 2.

Definition 2 (Satisfaction) *We say that $\mathcal{R}_\Gamma$ is a satisfaction relation between process P and specification Δ, if:*

whenever $\Delta \xrightarrow[\Gamma]{\ell} \Delta'$ with an input label ℓ, then $P \xrightarrow{\ell} P'$ and $P' \mathcal{R}_\Gamma \Delta'$,

whenever $P \xrightarrow{\ell} P'$ with an output label ℓ, then $\Delta \xrightarrow[\Gamma]{\ell} \Delta'$ and $P' \mathcal{R}_\Gamma \Delta'$.

The largest relation $\mathcal{R}_\Gamma$ is called satisfaction w.r.t. Γ *denoted* $\mathbf{sat}(P, \Delta)_\Gamma$. *In this case, we say that P satisfies Δ w.r.t. Γ (we omit this last part when Γ is clear from context)*

$$(\textbf{Ssub})\frac{\begin{array}{c}\Gamma(\mathcal{P}) = (\widetilde{\mathtt{r}}^1, \widetilde{y} : \widetilde{S}; \widetilde{\mathtt{r}}^2; G_{\mathcal{P}}) \qquad \forall i, G_{\mathcal{P}}\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}_i^1} = T_i' \\ \forall j, G_{\mathcal{P}}\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}_j^2} = T_j'' \qquad \Gamma(k) = G\{\widetilde{v}/\widetilde{y}\}\end{array}}{s[\mathtt{r}] : \mathtt{call}\ \mathcal{P} : G\ \mathtt{with}\ (\widetilde{v}\ \mathtt{as}\ \widetilde{y} : \widetilde{S})\&(\widetilde{\mathtt{r}}^2).T \xrightarrow[\Gamma]{\tau} s[\mathtt{r}] : T, (k[\mathtt{r}_i^1] : T_i')_i, (k[\mathtt{r}_j^2] : T_j'')_j}$$

$$(\textbf{Sout})\frac{\Gamma(a) = T[\mathtt{r}]}{k[\mathtt{r}]^\bullet : T \xrightarrow[\Gamma]{\overline{a}\langle k\rangle} \emptyset} \qquad (\textbf{Sin})\frac{\Gamma(a) = T[\mathtt{r}]}{\emptyset \xrightarrow[\Gamma]{a\langle k\rangle} k[\mathtt{r}] : T} \qquad (\textbf{ScomC})\frac{}{k[\mathtt{r}'']^\bullet : T'' \xrightarrow[\Gamma]{\tau} k[\mathtt{r}''] : T''}$$

$$(\textbf{Sjoin})\frac{\Gamma(\mathcal{P}) = (\widetilde{\mathtt{r}}^1, \widetilde{y} : \widetilde{S}; \widetilde{\mathtt{r}}^2; G_{\mathcal{P}}) \qquad G_{\mathcal{P}}\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}''} = T''}{s[\mathtt{r}'] : \mathtt{ent}\ \mathcal{P}[\mathtt{r}'']\langle\widetilde{v}\rangle\ \mathtt{from}\ \mathtt{r}.T \xrightarrow[\Gamma]{s\downarrow[\mathtt{r},\mathtt{r}':\mathtt{r}'']\langle k\rangle} s[\mathtt{r}'] : T, k : [\mathtt{r}''] : T''}$$

$$(\textbf{Sparti})\frac{\Gamma(\mathcal{P}) = (\widetilde{\mathtt{r}}^1, \widetilde{y} : \widetilde{S}; \widetilde{\mathtt{r}}^2; G_{\mathcal{P}}) \qquad G_{\mathcal{P}}\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}''} = T''}{s[\mathtt{r}] : \mathtt{req}\ \mathcal{P}[\mathtt{r}'']\langle\widetilde{v}\rangle\ \mathtt{to}\ \mathtt{r}'.T, k[\mathtt{r}'']^\circ : T'' \xrightarrow[\Gamma]{s\uparrow[\mathtt{r},\mathtt{r}':\mathtt{r}'']\langle k\rangle} s[\mathtt{r}] : T}$$

$$(\textbf{Sinvit})\frac{\begin{array}{c}\Gamma(\mathcal{P}) = (\widetilde{\mathtt{r}}^1, \widetilde{y} : \widetilde{S}; \widetilde{\mathtt{r}}^2; G_{\mathcal{P}}) \qquad G_{\mathcal{P}}\{\widetilde{v}/\widetilde{y}\} \Downarrow_{\mathtt{r}''} = T'' \\ s[\mathtt{r}] : \mathtt{ent}\ \mathcal{P}[\mathtt{r}'']\langle\widetilde{v}\rangle\ \mathtt{from}\ \mathtt{r}.T, s[\mathtt{r}'] : \mathtt{req}\ \mathcal{P}[\mathtt{r}'']\langle\widetilde{v}\rangle\ \mathtt{to}\ \mathtt{r}.T', k[\mathtt{r}'']^\circ : T''\end{array}}{\xrightarrow[\Gamma]{\tau} s[\mathtt{r}] : T, s[\mathtt{r}'] : T', k[\mathtt{r}''] : T''}$$

Fig. 2. Transitions for specifications (excerpt)

We justify the soundness of our framework by relating the static validation to the dynamic satisfaction, through *correspondence*. If a process is validated by a specification Δ, it is able to behave as described in Δ. From this property, we derive subject reduction, which ensures that validation is preserved by reduction. A session environment is *coherent* if it is composed of projections of well-formed global types. A coherent session environment is *simple* if it consists of a single session. A process is *unblocked* if it does not contain hidden channels and if its session channel is never under a prefix whose subject is a shared channel, except when the latter binds the former. If an unblocked process is validated by a simple coherent session environment, interactions at session channels can proceed. If, further, the original global type is non-recursive, the process can eventually complete all interactions at its session-environment.

Proposition 3 (Soundness of the type system)

(Correspondence) If $\Gamma \vdash P \triangleright \Delta$ *then* $\mathbf{sat}(P, \Delta)_\Gamma$.

(Subject Reduction) If $\Gamma \vdash P \triangleright \Delta$ *and* $P \to P'$ *then there exists* Δ' *s.t.* $\Gamma \vdash P' \triangleright \Delta'$.

(Progress) If P *is unblocked and* $\Gamma \vdash P \triangleright \Delta$ *such that* Δ *is simple, then there exists* P' *s.t.* $P \to^+ P'$, $\Gamma \vdash P' \triangleright \Delta'$ *and* Δ' *is coherent.*

(Coherence) If P *is unblocked and* $\Gamma \vdash P \triangleright \Delta$ *such that* Δ *is simple, and moreover* Δ *does not contain recursions, then there exists* P' *s.t.* $P \to^* P'$ *and* $\Gamma \vdash P' \triangleright \emptyset$.

5 Returning a result

We introduce the notion of *result* of a session as an object (which can be a value or even a protocol), sent back to the initiator of the session. *Protocols with results* allow us to describe complex governance properties, such as ensuring that a privately negotiated price corresponds to the one proposed publicly in the parent protocol. Suppose we want to ensure that, in CMS, the answer ans given in the subprotocol $Contact$ by server is the same as ans_0 sent by middle to client. Information can be transmitted from a parent session to a subsession, but the converse is not possible. Continuation-Passing-Style is a possible solution: we convert the end of the CMS protocol into a continuation K, send it as argument when calling $Contact$ and call K inside $Contact$ with ans. However, this may not lead to a clean descriptive framework. Thus we choose to use a dedicated mechanism. The syntax of global types with results adds $\texttt{r returns}(res : S)$ and $(res : S) \leftarrow \texttt{r calls } \mathcal{P}\langle \widetilde{\texttt{r}}, \widetilde{v} \rangle$ (replacing `end` and $\texttt{r calls } \mathcal{P}\langle \widetilde{v} \rangle$). The former constructor ends the session by specifying that the protocol returns the value identified in the session by res and that `r` is responsible for doing it, the latter specifies that we call a subprotocol which eventually produces a result res. Kinds ensure that the returned result has the type expected by the initiator.

We present corresponding modifications to CMS. Inside the $Contact$ protocol, we ask agent to send the result ans back to the parent protocol. In the latter, the result ans_0 is expected when calling $Contact$, thus we ensure that the answer sent by the server in the subprotocol is the same as the one sent to client in the parent one. Local types use similar constructors and implementation of result is done through cross-session communications.

$$
\begin{array}{l}
\texttt{let } Contact = (\text{agent}, req : \texttt{Req})\{ \\
\ldots \text{agent } \texttt{returns}(ans : \texttt{Req}) \\
\texttt{in} \\
\ldots ans_0 : \texttt{Req} \leftarrow \text{middle } \texttt{calls } Contact\langle \text{middle}, req_0 \rangle \ldots
\end{array}
$$

In the framework presented above, subsessions are executed in parallel with the parent session. The result mechanism allows one to include synchronisation between the two sessions:

$$
\begin{array}{l}
\ldots \text{ Alice } \texttt{waits for } contract \texttt{ calling } \text{Nego}(\text{Alice}, \text{Carol}). \\
\quad \text{Alice} \rightarrow \text{Bob} : \texttt{Data}(contract) \ldots
\end{array}
$$

Here participant Alice starts a negotiation subsession with Carol. When the negotiation is over, she sends the result of the subsession to Bob, participant of the parent session not invited in the subsession. This has two advantages, first Bob can know the result of a subsession without going through the internal invitation procedure, and it prevents both Alice and Bob to perform actions in the parent session as long as the subsession is not over.

6 Conclusion and Future Works

To our knowledge, there does not exist other works addressing the notion of nested session types, or protocol calls inside session types. The closest contribu-

tion is [9], which introduces parametrisation of protocols through *dynamic session types*. Parametrisation allows one single two-party protocol to be applied to each pair of agents in a large network. Our framework contains more than simple parametrisation, it presents nesting and introduces kinds and higher-order programming. Another related work is [7]: the authors describe a global language for *choreographies*, implementing global types, protocols interleaved in the same choreography can be merged together into a single global type, removing costly invitations. The authors actually proceed in a direction different from ours, by trying to unify every protocol into a single superprotocol. Their approach focuses on implementation, while ours focus on types. We believe session type theory benefits independently from both methods. Our contribution makes use of the same formal framework as [3, 13, 2]. Each of these contributions adds expressiveness, in different directions (logical assertions, ghost states, monitoring), to a large common theory for validation of distributed programs with session types. The whole theory (including this work) is put in practise by the development of the Scribble language [16] and the collaboration with the OOI project [15].

We are currently investigating how the result mechanism can be improved (in the context of [16]). Currently, the result is sent to the initiator. Broadcasting the result to every member of the subsession might also be a desirable feature. Moreover, our results are restricted to value-types, but some use cases of [15] specify that a negotiation subprotocol produces a contract that is used in the parent protocol to control interactions. Although it leads to technical challenges, we believe our framework can eventually accommodate such behaviours by using dependent types, introducing abstract logical predicate decided at run-time inside global types. Exceptions handling in a distributed asynchronous framework, remains a challenging task, even if some progress have been made in [6] and [5]. Yet exceptions are absolutely necessary when specifying real-world protocols. We believe that nested protocols give a simple way to handle exceptions, by making explicit blocks of computation.

Acknowledgements We thank the CONCUR reviewers for their comments, our colleagues in Mobility Reading Group for discussions, and the OOI project and Matthew Arrott for their feedback. This work is supported by Ocean Observatories Initiative [15] and EPSRC grants EP/F002114/1 and EP/G015481/1.

References

1. Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, pages 418–433, 2008.
2. Laura Bocchi, Pierre-Malo Denéilou, Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Dynamic and static safety validation in distributed programs through multiparty sessions. (submitted), 2012.
3. Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *LNCS*, pages 162–176, 2010.

4. Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Information flow safety in multiparty sessions. In *EXPRESS*, volume 64 of *EPTCS*, pages 16–30, 2011.
5. Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 8 of *LIPIcs*, pages 338–351. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.
6. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
7. Marco Carbone and Fabrizio Montesi. Merging multiparty protocols in multiparty choreographies. (unpublished, presented at PLACES), 2012.
8. Mario Coppo and Mariangiola Dezani-Ciancaglini. Structured communications with concurrent constraints. In *TGC*, pages 104–125, 2008.
9. Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446, 2011.
10. Mariangiola Dezani-Ciancaglini. A reputation system for multirole sessions. Invited talk at TGC, September 2011.
11. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
12. Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In Antonio Cerone and Stefan Gruner, editors, *SEFM*, pages 323–332. IEEE Computer Society, 2008.
13. Romain Demangeon Laura Bocchi and Nobuko Yoshida. A multiparty multisession logic. (submitted), 2012.
14. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, 1980.
15. Ocean Observatories Initiative (OOI). `http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/`.
16. Scribble Project homepage. `www.scribble.org`.