Typing Dynamic Roles in Multiparty Interaction

Pedro Baltazar[1], Vasco T. Vasconcelos[1], and Hugo T. Vieira[2]

[1] LaSIGE, Dept. de Informática, Faculdade de Ciências,
Universidade de Lisboa, Portugal
{pbtz|vv}@di.fc.ul.pt
[2] CITI, Dept. de Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, Portugal
htv@fct.unl.pt

Abstract. We present a type-based analysis for role-based multiparty interaction. Novel to our approach are the notions that a role specified in a protocol may be carried out by several parties, and that one party may assume different roles at different stages of the protocol. We build on Conversation Types by adding roles to protocol specifications. Systems are modeled in π-calculus extended with labeled communication and role annotations. The main result shows that well-typed systems follow the role-based protocols prescribed by the types, addressing systems where roles have dynamic distributed implementations.

1 Introduction

Communication is a central feature of nowadays software systems, as more and more often systems are built using computational resources that are concurrently available and distributed in the network. Extreme examples include operating systems where functionality is distributed between distinct threads in the system, and a service available on the Internet which relies on third-party (remote) service providers to carry out subsidiary tasks. Building software from the composition of distributed interacting pieces is very flexible, at least in principle, since resources can be dynamically discovered and chosen according to criteria such as availability and work load. In such a setting, all interacting parties must agree on the communication protocols, and verification mechanisms that automatically check if the code meets some common protocol specification are then of crucial importance.

A protocol specification describes the set of message exchanges, including when are they to occur and the parties involved in the interaction. A party involved in a protocol may have a spatial meaning, for instance denoting a distinguished site or process, or, more generally, a party may have a behavioral meaning, a *role* in the interaction that may be realized by one or more processes or sites. Conversely, a process may impersonate different roles throughout its execution. Such flexibility is essential to address systems, e.g., where a leader role is impersonated by different sites at different stages of the protocol, and the role of a single site changes accordingly. The challenge is then to verify whether a

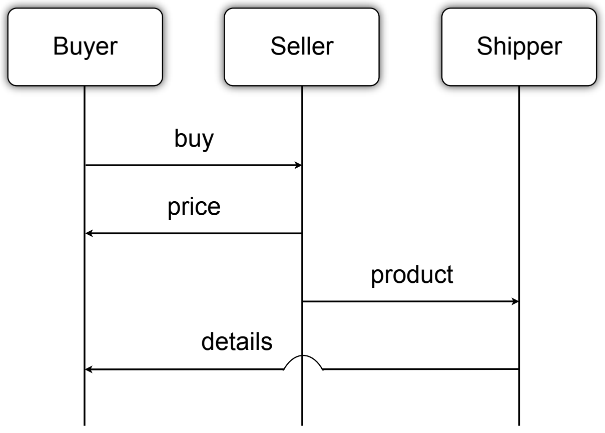

Fig. 1. Purchase Interaction Message Sequence Chart.

system complies to a protocol specification, given such dynamic and distributed implementation of roles, just by inspecting at the source code.

In this paper we present a type-based analysis which checks if systems follow the role-based protocol descriptions prescribed by the types. Our development is based on the conversation type theory [2, 3], extending it with the ability to specify and analyze the roles involved in the interactions. The underlying model of our analysis is based on TyCO [9], used in our setting as an extension to the π-calculus [8], where communication actions specify a message label and the role performing the action. While retaining the simplicity of conversation types, our theory is able to address systems where a single role may be realized by several parties and when processes may dynamically change the role in which they are interacting. This contrasts with related approaches (see, e.g., [7, 4]) where roles have a spatial meaning, as they are mapped into the structure of systems in a static way.

In the remaining of the Introduction we informally describe the type analysis of an example system, and also present a second example to illustrate the flexibility of our approach. Consider the message sequence chart shown in Fig. 1 (taken from [3]), which captures the interaction of a purchase system involving three parties. Messages buy, price, product and details are (sequentially) exchanged between a Buyer, a Seller, and a Shipper. First, Buyer sends to Seller a buy message, then Seller replies to Buyer on message price. After that, Seller sends to Shipper message product and Shipper sends to Buyer message details.

Fig. 2 illustrates a possible implementation of the purchase interaction. Process Buyer specifies the creation of a fresh chat that will host the purchase interaction, via the **new** construct. This newly created name is passed to the Seller, via message buyService. The code $\mathit{Seller} \lhd_{\mathbf{Buyer}} \mathit{buyService}(\mathit{chat})$ represents the output of message buyService, in channel Seller, passing name chat.

- labels $\mathcal{L} = \{\mathit{buyService}, \mathit{shipService}, \mathit{buy}, \mathit{product}, \mathit{price}, \mathit{details}\}$
- names $\mathcal{N} = \{x, \mathit{Seller}, \mathit{Shipper}, \mathit{Buyer}, \mathit{chat}\}$
- roles $\mathcal{R} = \{\mathbf{Seller}, \mathbf{Shipper}, \mathbf{Buyer}\}$

$$
\begin{array}{lcl}
\texttt{Buyer} & \overset{\text{def}}{\equiv} & (\mathbf{new}\ \mathit{chat}) \\
 & & \mathit{Seller} \triangleleft_{\mathbf{Buyer}} \mathit{buyService}(\mathit{chat}). \\
 & & \quad \mathit{chat} \triangleleft_{\mathbf{Buyer}} \mathit{buy}(). \\
 & & \quad \mathit{chat} \triangleright_{\mathbf{Buyer}} \mathit{price}(). \\
 & & \quad \mathit{chat} \triangleright_{\mathbf{Buyer}} \mathit{details}() \\
\texttt{Seller} & \overset{\text{def}}{\equiv} & \mathit{Seller} \triangleright_{\mathbf{Seller}} \mathit{buyService}(x). \\
 & & \quad x \triangleright_{\mathbf{Seller}} \mathit{buy}(). \\
 & & \quad x \triangleleft_{\mathbf{Seller}} \mathit{price}(). \\
 & & \quad \mathit{Shipper} \triangleleft_{\mathbf{Seller}} \mathit{shipService}(x). \\
 & & \quad\quad x \triangleleft_{\mathbf{Seller}} \mathit{product}() \\
\texttt{Shipper} & \overset{\text{def}}{\equiv} & \mathit{Shipper} \triangleright_{\mathbf{Shipper}} \mathit{shipService}(x). \\
 & & \quad x \triangleright_{\mathbf{Shipper}} \mathit{product}(). \\
 & & \quad x \triangleleft_{\mathbf{Shipper}} \mathit{details}() \\
\texttt{System} & \overset{\text{def}}{\equiv} & (\texttt{Buyer} \mid \texttt{Seller} \mid \texttt{Shipper})
\end{array}
$$

Fig. 2. Purchase System Code (a).

Also, the role which sends the message is identified (role Buyer). The remaining interactions of the Buyer process (and, in this case, role) are in channel chat: first sends ($\triangleleft$) message buy, after which receives ($\triangleright$) message price and then sends message details.

The Seller process starts by receiving a channel name (that instantiates variable x) in message buyService. Then, in this received channel the Seller process (and role) receives a buy message, after which sends message price. At this point, the Seller asks a third party to join the ongoing interaction, by sending message shipService in Shipper passing the identity of the channel which hosts the ongoing interaction (x). After that, the Seller process keeps interacting in the delegated channel x, sending a product message in it. The Shipper process (and role) receives a channel name in message shipService in channel Shipper, in which the process receives message product and then sends message details.

The implementation shown in Fig. 2 involves three distinguished processes that carry out the three roles identified in the protocol. The type:

$$
\begin{array}{l}
\texttt{Buyer} \rightarrow \texttt{Seller}\ \mathit{buy}(). \\
\texttt{Seller} \rightarrow \texttt{Buyer}\ \mathit{price}(). \\
\texttt{Seller} \rightarrow \texttt{Shipper}\ \mathit{product}(). \\
\texttt{Shipper} \rightarrow \texttt{Buyer}\ \mathit{details}()
\end{array}
\tag{1}
$$

(which describes the protocol illustrated in Fig. 1) captures the interaction in channel chat. Channel chat is passed along from Buyer to Seller and then from Seller to Shipper in messages buyService and shipService, respectively. In order to analyze the protocol distribution between the three parties, we must consider the *slices* of protocol which are delegated in messages buyService and shipService.

The type $?\texttt{Shipper}\, \mathit{product}().\, !\texttt{Shipper}\, \mathit{details}()$ describes the interactions of Shipper in the channel received in the shipService message. From the point of view of the Seller, when it asks Shipper to join the ongoing interaction, the type $?\texttt{Shipper}\, \mathit{product}().\, !\texttt{Shipper}\, \mathit{details}()$ is delegated in the output of message shipService. So, to type the Seller we need to consider both the slice of protocol delegated in the message and the slice retained by the emitting process (the output of message product, typed $!\texttt{Seller}\, \mathit{product}()$). This composition is captured via a *merge* ($\bowtie$) operation, which describes the behavioral combination of the protocols:

$$?\texttt{Shipper}\, \mathit{product}().\, !\texttt{Shipper}\, \mathit{details}() \;\bowtie\, !\texttt{Seller}\, \mathit{product}() \\ \Rightarrow \texttt{Seller} \rightarrow \texttt{Shipper}\, \mathit{product}().\, !\texttt{Shipper}\, \mathit{details}()$$

The merge yields type $\texttt{Seller} \rightarrow \texttt{Shipper}\, \mathit{product}().\, !\texttt{Shipper}\, \mathit{details}()$ which describes the protocol of Seller and Shipper combined, at the moment the Shipper is asked to join the interaction: Seller sends to Shipper message product and Shipper outputs message details. Notice that both a message exchange internal to the system, where both emitting and receiving roles are identified, and a message exchanged with the external environment, where just the role internal to the system is identified, are specified in the type.

The Seller interaction in the channel received in message buyService is typed:

$$?\texttt{Seller}\, \mathit{buy}().\, !\texttt{Seller}\, \mathit{price}().\, \texttt{Seller} \rightarrow \texttt{Shipper}\, \mathit{product}().\, !\texttt{Shipper}\, \mathit{details}()$$

which (behaviorally) combined with the type of the Buyer interaction:

$$!\texttt{Buyer}\, \mathit{buy}().?\texttt{Buyer}\, \mathit{price}().?\texttt{Buyer}\, \mathit{details}()$$

yields the expected protocol, that of (1).

In the system shown in Fig. 2 there is a perfect matching between processes and the roles in which the processes interact. However, as explained before, in our model this does not need to be the case. Fig. 3 shows a distinct implementation of the interaction scheme given in Fig. 1 with two differences. The first difference is that now the Buyer delegates the reception of message details to some specialized mailbox. More specifically, after receiving message price, the Buyer asks MailBox to join the ongoing interaction, passing name chat in message storeService. MailBox receives a name in message storeService, and, assuming the role of Buyer, receives message details in the received channel. The second difference is that the Seller process, in particular the code that follows the reception of the buyService message, is now also responsible for the interactions of the Shipper role. The Seller process starts by interacting in the Seller role (messages buy and price) and then spawns two threads with different roles (Seller and Shipper).

The type of Seller interaction in the channel received in the buyService is the same as before, now explained by the merge of the behaviors of the two parallel threads. Instead, the type of the Buyer interaction is also recovered, now obtained

- labels $\mathcal{L} = \{\mathit{buyService}, \mathit{buy}, \mathit{product}, \mathit{price}, \mathit{details}, \mathit{storeService}\}$
- names $\mathcal{N} = \{x, \mathit{Seller}, \mathit{Buyer}, \mathit{MailBox}, \mathit{chat}\}$
- roles $\mathcal{R} = \{\mathbf{Seller}, \mathbf{Shipper}, \mathbf{Buyer}\}$

$$
\begin{array}{lcl}
\texttt{Buyer} & \overset{\text{def}}{\equiv} & (\mathbf{new}\ \mathit{chat}) \\
 & & \mathit{Seller} \triangleleft_{\mathbf{Buyer}} \mathit{buyService}(\mathit{chat}). \\
 & & \quad \mathit{chat} \triangleleft_{\mathbf{Buyer}} \mathit{buy}(). \\
 & & \quad \mathit{chat} \triangleright_{\mathbf{Buyer}} \mathit{price}(). \\
 & & \quad \mathit{MailBox} \triangleleft_{\mathbf{Buyer}} \mathit{storeService}(\mathit{chat}) \\
\texttt{Seller} & \overset{\text{def}}{\equiv} & \mathit{Seller} \triangleright_{\mathbf{Seller}} \mathit{buyService}(x). \\
 & & \quad x \triangleright_{\mathbf{Seller}} \mathit{buy}(). \\
 & & \quad x \triangleleft_{\mathbf{Seller}} \mathit{price}(). \\
 & & \qquad (x \triangleleft_{\mathbf{Seller}} \mathit{product}() \\
 & & \qquad \mid \\
 & & \qquad x \triangleright_{\mathbf{Shipper}} \mathit{product}(). \\
 & & \qquad x \triangleleft_{\mathbf{Shipper}} \mathit{details}()) \\
\texttt{MailBox} & \overset{\text{def}}{\equiv} & \mathit{MailBox} \triangleright_{\mathbf{Mail}} \mathit{storeService}(x). \\
 & & \quad x \triangleright_{\mathbf{Buyer}} \mathit{details}() \\
\texttt{System} & \overset{\text{def}}{\equiv} & (\texttt{Buyer} \mid \texttt{MailBox} \mid \texttt{Seller})
\end{array}
$$

Fig. 3. Purchase System Code (b).

considering the delegated slice of protocol to the MailBox in message storeService (the reception of message details in role Buyer). The purchase interaction of the system shown in Fig. 3 also follows the protocol specification depicted in Fig. 1. From the point of view of our type analysis both systems follow the prescribed protocol, regardless of the spatial configuration of the processes implementing the roles. Notice that the Buyer role is distributed between the Buyer and MailBox processes, and that roles Seller and Shipper are carried out by process Seller.

In the rest of the paper we present our formal development, including the definition of the semantics of the process and type languages and of the type system. Our main result says that given a type specification and a well-typed implementation then each interaction in the process is explained by a corresponding interaction in the types, hence, all interactions in the process follow the protocols prescribed in the types.

2 Process Language

In this section we define the language of processes, and its structural congruence and semantics.

First, the language of processes can be seen as variant of TyCO, with labelled inputs and outputs. We start be considering denumerable sets of *labels* $\mathcal{L}$, *names* $\mathcal{N}$, and *roles* $\mathcal{R}$. In Fig. 4 we present the syntax of *processes*. As usual, the language comprises the inaction $\mathbf{0}$, name restriction $(\mathbf{new}\ x)P$, and parallel composition $P_1 \mid P_2$ of concurrent processes P_1 and P_2. Moreover, the language include terms $x \triangleleft_r l(y).P$ (answers), which represent a process that performs a output ("replies" or "declares") $l(y)$ on channel x with role r, and continues executing P. For the complementary input action, the syntax includes terms $x \triangleright_r \{l_i(x_i).P_i\}_{i\in I}$ (questions) describing a process that inputs ("asks" or "listens") $\{l_i\}_{i\in I}$ on channel x, with *role* r. And after receiving an answer $l_i(y)$ on x, binds x_i to y and continues with P_i, for any $i \in I$. In order to avoid ambiguities, we require each of these terms to have distinct labels. Processes are ranged over by $P, P', \dots, P_1, P_2, \dots$ We take the standard definitions of free and bound names or variables for processes. Processes that differ only in the names of bound variables are deemed equal. And we also assume that every bound name or variable is different from the others. We write $P[x \leftarrow y]$ for the substitution of y to free occurrences of x in P.

$$\begin{array}{rlr}
P ::= & \mathbf{0} \;[\!]\; (\mathbf{new}\ x)P \;[\!]\; (P_1 \mid P_2) \;[\!]\; x \triangleright_r \{l_i(x_i).P_i\}_{i\in I} \;[\!]\; x \triangleleft_r l(y).P & \text{Process terms} \\
l \in \mathcal{L} & & \text{Labels} \\
x, y \in \mathcal{N} & & \text{Names} \\
r, s \in \mathcal{R} & & \text{Roles}
\end{array}$$

Fig. 4. Syntax of Process

The structural congruence between process $P_1 \equiv P_2$ is defined as the least equivalence relation that fulfills the conditions expressed in Fig. 5, and that preserves the process constructors.

$$\begin{gathered}
P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \\
(\mathbf{new}\ x)(\mathbf{new}\ y)P \equiv (\mathbf{new}\ y)(\mathbf{new}\ x)P \\
(\mathbf{new}\ x)P_1 \mid P_2 \equiv (\mathbf{new}\ x)(P_1 \mid P_2) \quad \text{if } x \notin \mathrm{fn}(P_2) \\
(\mathbf{new}\ x)\mathbf{0} \equiv \mathbf{0}
\end{gathered}$$

Fig. 5. Structural Congruence

The semantics of processes is defined as the least relation $P \xrightarrow{\lambda} P'$ that satisfies the rules in Fig. 6. Here, λ is a *usage label* and is either empty (τ) or a communication ($x : r \to s : l$). Usage labels are important to entangle and link the semantics of processes with that of typing environments. In the sequel, they will be the backbone to define and proof preservation of well–typed processes. The meaning associated to usage labels is that, one step reduction with label

$x : r \to s : l$ denotes that a message labelled l is communicated from a party assuming role r to another with role s. And a reduction with label τ means that the reduction occurs in a restricted name.

$$x \triangleright_r \{l_i(x_i).P_i\}_{i \in I} \mid x \triangleleft_s l(y).P \xrightarrow{x:r\to s:l} P_k[x_k \leftarrow y] \mid P \qquad \text{if } \exists k \in I, l = l_k \qquad \text{(Red-Comm)}$$

$$\frac{P_1 \xrightarrow{\lambda} P_1'}{P_1 \mid P_2 \xrightarrow{\lambda} P_1' \mid P_2} \qquad \text{(Red-Par)}$$

$$\frac{P \xrightarrow{\lambda} P' \qquad \lambda \in \{\tau, x : r \to s : l\}}{(\mathbf{new}\ x)P \xrightarrow{\tau} (\mathbf{new}\ x)P'} \qquad \text{(Red-New1)}$$

$$\frac{P \xrightarrow{\lambda} P' \qquad \lambda = x : r \to s : l \qquad y \neq x}{(\mathbf{new}\ y)P \xrightarrow{\lambda} (\mathbf{new}\ y)P'} \qquad \text{(Red-New2)}$$

$$\frac{P_1 \equiv P_1' \qquad P_1' \xrightarrow{\lambda} P_2' \qquad P_2' \equiv P_2}{P_1 \xrightarrow{\lambda} P_2} \qquad \text{(Red-Struct)}$$

$$\lambda ::= \tau \ [\!]\ x : r \to s : l$$

Fig. 6. Operational Semantics.

For example, the application of rule [Red-Comm] in our Example 2 yields the following transition. The process

$$\begin{gathered} \mathit{Shipper} \triangleleft_{\mathbf{Seller}} \mathit{shipService}(\mathit{chat}).\mathit{chat} \triangleleft_{\mathbf{Seller}} \mathit{product}.\mathbf{0} \mid \\ \mathit{Shipper} \triangleright_{\mathbf{Shipper}} \mathit{shipService}(x).x \triangleright_{\mathbf{Shipper}} \mathit{product}.x \triangleleft_{\mathbf{Shipper}} \mathit{details}.\mathbf{0} \end{gathered}$$

reduces to

$$\mathit{chat} \triangleleft_{\mathbf{Seller}} \mathit{product}.\mathbf{0} \mid \mathit{chat} \triangleright_{\mathbf{Shipper}} \mathit{product}.\mathit{chat} \triangleleft_{\mathbf{Shipper}} \mathit{details}.\mathbf{0}$$

with label $\mathit{Shipper} : \mathbf{Seller} \to \mathbf{Shipper} : \mathit{shipService}$.

3 Type System

In this section we present a behavioral type system for conversation processes. To each process and channel we assign the "conversation" (exchange of messages between roles) which takes place in that channel.

We start by defining the type language, and its semantics. Next, we introduce a merge operation between conversation types, and exemplify its use. Finally,

$$\begin{array}{rlr}
B & ::= \mathbf{end} \;[]\; B \mid B \;[]\; \rho\{M_i.B_i\}_{i\in I} & \text{types} \\
\rho & ::= \,!r \;[]\; ?r \;[]\; r \rightarrow s & \text{communication prefixes} \\
M & ::= l(B) & \text{message types} \\
l & \in \mathcal{L} & \text{Labels} \\
r, s & \in \mathcal{R} & \text{Roles}
\end{array}$$

Fig. 7. Syntax of Conversation Types

we state the main result, which demonstrates how the correctness of a process is captured by the typing system.

Fig. 7 defines the syntax of conversation types. The behavior of a channel is **end** if nothing is to be "said" in that channel. The parallel composition $B_1 \mid B_2$ describes a channel where two conversations B_1 and B_2 occur concurrently. And, $\rho\{M_i.B_i\}_{i\in I}$ represents "saying" $\{M_i\}_{i\in I}$ and after having the possibility of continuing as B_i, for all $i \in I$. Once more, to disallow confusions, we consider that all messages $\{M_i\}_{i\in I}$ are different. The three possible communication prefix have the following meanings:

- $!r$ - output (declared/replying) with role r;
- $?r$ - input (listening/asking) with role r;
- $r \rightarrow s$ - a communication from role r to role s.

The prefixes $!r$ and $?r$ are called *unmatched*, since they represent waiting for a question and reply, respectively. And the prefix $r \rightarrow s$ is called a *match*, because it represents an already identified exchange question-reply between roles r and s. The matched prefix $r \rightarrow s$ will be crucial to verify if in a conversation nothing is left unanswered or unlistened. The predicate $\mathbf{matched}(B)$ characterizes all types containing only matched prefixes.

The structural congruence $B_1 \equiv B_2$, which defines equivalent types, is the least equivalence relation where parallel composition is commutative and associative, $\mathbf{end} \mid B \equiv B$, and that preserves the type constructors. In the sequel, we identify congruent types.

The semantics of types is based on the idea that the type of process $P_1 \mid P_2$ must be obtained from the types of P_1 and P_2. And, the resulting type must capture all possible communications between the two threads via matched prefixes. In Fig. 8 we present the operational semantics of conversation types. Rule [B-Red-Com] expresses the principle that reduction is limited to matched communications, i.e., matched prefixes $r \rightarrow s$. Hence, no interaction may occur between parallel types, which means they represent two concurrent independent conversations. Type independence is defined by apartness: two types B_1 and B_2 are apart, noted $B_1 \# B_2$, if they do not have labels in common. The definition of merge imposes a linear usage of labels and guarantees race-free conversations,

$$r \to s\{M_i.B_i\}_{i\in I} \xrightarrow{r\to s:l_k} B_k \qquad \forall k \in I \qquad \text{(B-Red-Com)}$$

$$\frac{B_1 \xrightarrow{r\to s:l} B_2}{B_1 \mid B \xrightarrow{r\to s:l} B_2 \mid B} \qquad \text{(B-Red-Par)}$$

Fig. 8. Types Semantics

where, at any moment, there are at most two parties able to exchange a given message.

In our running Example 2, the conversation which takes place in channel *chat* is given by the type:

$$\begin{array}{l}\mathbf{Buyer} \to \mathbf{Seller}\{\mathit{buy}().\mathbf{Seller} \to \mathbf{Buyer}\{\mathit{price}().\\ \mathbf{Seller} \to \mathbf{Shipper}\{\mathit{product}().\mathbf{Shipper} \to \mathbf{Buyer}\{\mathit{details}().\mathbf{end}\}\}\}\}\end{array}$$

Guided by the above requirements, in Fig. 9 we define a merge operation intended to resolve and match communications. Given types B_1 and B_2,

$$B_1 \bowtie B_2 \Rightarrow B$$

means that $B_1 \bowtie B_2$ resolves to B. And, either B comes from matching all possible communications or by resolving to $B_1 \mid B_2$. For the sake of space we omit the symmetric version of rules (PAR), (TAU), (SHF) and (BRK).

Returning to Example 2, the type of channel *chat* is obtained by first performing the following reduction, where we apply (APT) and (TAU), using the equivalence $B \mid \mathbf{end} \equiv B$. ($\mathbf{s} = \mathbf{Seller}$ and $\mathbf{sh} = \mathbf{Shipper}$)

$$\frac{\mathbf{end} \bowtie !\mathbf{sh}\{\mathit{details}.\mathbf{end}\} \Rightarrow !\mathbf{sh}\{\mathit{details}.\mathbf{end}\}}{!\mathbf{s}\{\mathit{product}.\mathbf{end}\} \bowtie !\mathbf{sh}\{\mathit{product}.?\mathbf{sh}\{\mathit{details}.\mathbf{end}\}\} \Rightarrow \mathbf{s} \to \mathbf{sh}\{\mathit{product}.!\mathbf{sh}\{\mathit{details}.\mathbf{end}\}\}}$$

As usual, a *type environment* is a partial map Γ that assigns conversation types to names, and we write $\Gamma \vdash x : B$ to say that Γ assigns B to name $x \in \mathcal{N}$. By $\Gamma, x : B$ we denote a type environment Γ' that $(\Gamma' \vdash x : B)$ assigns B to x and behaves as Γ on other names. In Fig. 10 we introduce the type system for conversation types. In the rules we use $\Gamma_1 \bowtie \Gamma_2$ to represent the environment obtained by merging the behaviors associated to each name in Γ_1 and Γ_2. For example, $a : B_1 \bowtie a : B_2$ yields environment $a : B$ provided that $B_1 \bowtie B_2 \Rightarrow B$. The merge of behavioral types is defined in Fig. 9,

In our Example 2, the merge example given above appears when typing the subprocess $\mathit{Shipper} \triangleleft_{\mathbf{Seller}}\ \mathit{shipService}(\mathit{chat}).\mathit{chat} \triangleleft_{\mathbf{Seller}}\ \mathit{product}.\mathbf{0}$

$$\dfrac{\dfrac{\dfrac{}{\mathit{chat} : \mathbf{end}, \mathit{Shipper} : \mathbf{end} \vdash \mathbf{0}}\ \text{T-END}}{\mathit{chat} : !\mathbf{s}\{\mathit{product}.\mathbf{end}\}, \mathit{Shipper} : \mathbf{end} \vdash \mathit{chat} \triangleleft_{\mathbf{Seller}}\ \mathit{product}.\mathbf{0}}\ \text{T-OUT}}{\mathit{Shipper} : !\mathbf{s}\{\mathit{shipService}(B).\mathbf{end}\}, \mathit{chat} : !\mathbf{s}\{\mathit{product}.\mathbf{end}\} \bowtie B \vdash \ldots}\ \text{T-OUT}$$

$$B_1 \bowtie B_2 \Rightarrow B_1 \mid B_2 \qquad \text{if } B_1 \# B_2 \tag{APT}$$

$$\frac{B_2 \bowtie B_3 \Rightarrow B' \qquad B' \bowtie B_1 \Rightarrow B}{(B_1 \mid B_2) \bowtie B_3 \Rightarrow B} \tag{PAR}$$

$$\frac{B_i \bowtie B'_i \Rightarrow B''_i \qquad \forall i \in I}{!r_1\{M_i.B_i\}_{i\in I} \bowtie ?r_2\{M_i.B'_i\}_{i\in I} \Rightarrow r_1 \to r_2\{M_i.B''_i\}_{i\in I}} \tag{TAU}$$

$$\frac{B_i \bowtie \rho_1\{M_j.B'_j\}_{j\in J} \Rightarrow B''_i \qquad \mathbf{not}(\rho_1\{M_i.\mathbf{end}\}_{i\in I} \# B_i) \qquad \forall i \in I}{\rho_2\{M_i.B_i\}_{i\in I} \bowtie \rho_1\{M_j.B'_j\}_{j\in J} \Rightarrow \rho_2\{M_i.B''_i\}_{i\in I}} \tag{SHF}$$

$$\frac{B_i \bowtie B \Rightarrow B' \qquad \rho\{M_i.\mathbf{end}\}_{i\in I} \# B' \qquad \forall i \in I}{\rho\{M_i.B_i\}_{i\in I} \bowtie B \Rightarrow \rho\{M_i.\mathbf{end}\}_{i\in I} \mid B'} \tag{BRK}$$

Fig. 9. Merge operation

Now, from the typing of subprocess

$$\mathit{Shipper} \triangleright_{\mathbf{Shipper}} \mathit{shipService}(x).x \triangleright_{\mathbf{Shipper}} \mathit{product}.x \triangleleft_{\mathbf{Shipper}} \mathit{details}.\mathbf{0},$$

we will infer that B is in fact $!\mathbf{sh}\{\mathit{product}.?\mathbf{sh}\{\mathit{details}.\mathbf{end}\}\}$. Hence, in the above derivation, the type of *chat* will be resolved to $\mathbf{s} \to \mathbf{sh}\{\mathit{product}.!\mathbf{sh}\{\mathit{details}.\mathbf{end}\}\}$. Therefore, this fragment illustrates the use of the merge operation $\bowtie$, which step by step matches the possible communications. In this case, by matching the communication between **Seller** and **Shipper** with label *product*.

The semantics of conversations types can be extended to type environments.

Definition 1. *We write $\Gamma_1 \xrightarrow{x:r\to s:l} \Gamma_2$ to denote that*

$$\Gamma_1 = \Gamma, x : B_1,\ \Gamma_2 = \Gamma, x : B_2 \ \text{ and } B_1 \xrightarrow{r\to s:l} B_2.$$

Finally, we state the type-preservation propriety for conversation processes and types.

Theorem 1 (type preservation). *Suppose $\Gamma \vdash P$ and $P \xrightarrow{\lambda} P'$. Then,*

- *if $\lambda = \tau$, then $\Gamma \vdash P'$;*
- *if $\lambda = x : r \to s : l$, then $\Gamma' \vdash P'$ for some Γ' such that $\Gamma \xrightarrow{\lambda} \Gamma'$.*

The type preservation if a safety result. We know that the typing system yields conversation types of valid conversations. Therefore, the previous theorem asserts that if a conversation process is typed, then it is simulated by a valid type. Hence, assuring the correctness of the overall conversation.

$$x_1 : \mathbf{end}, \ldots, x_k : \mathbf{end} \vdash \mathbf{0} \qquad \text{(T-END)}$$

$$\frac{\forall i \in I \qquad \Gamma \bowtie x : B_i, y_i : B_i' \vdash P_i}{\Gamma, x : ?r\{l_i(B_i').B_i\}_{i \in I} \vdash x \triangleright_r \{l_i(y_i).P_i\}_{i \in I}} \qquad \text{(T-IN)}$$

$$\frac{k \in I \qquad \Gamma \bowtie x : B_k \vdash P}{\Gamma \bowtie x : !r\{l(B_i').B_i\}_{i \in I} \bowtie y : B_k' \vdash x \triangleleft_r l(y).P} \qquad \text{(T-OUT)}$$

$$\frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \bowtie \Gamma_2 \vdash P_1 \mid P_2} \qquad \text{(T-PAR)}$$

$$\frac{\Gamma, x : B \vdash P \qquad \mathbf{matched}(B)}{\Gamma \vdash (\mathbf{new}\ x)P} \qquad \text{(T-NEW)}$$

Fig. 10. Typing rules

4 Conclusion and Future Work

We have presented a type-based analysis which ensures that systems follow the prescribed role-based protocol specifications. New to our approach are the notions that a role may be implemented by several processes, and that a single process may change its role at different stages of the protocol.

Our development builds on conversation types, a flexible type structure that generalizes binary session types [5, 6] to address multiparty conversations, including when conversations are dynamically established and have an unanticipated number of participants. Conversation types were originally developed for the Conversation Calculus [10], a specialization of the π-calculus for service-oriented systems. In this paper, conversation types are used in a more canonical model, that essentially consists in the π-calculus extended with labeled communication. Our extension with role-based protocol specifications retains the simplicity of the conversation type theory, and contrasts with related approaches in the dynamic and flexible nature of roles. In [1, 7, 11, 4] roles are structurally mapped to processes in a way that roles and threads end up in a one to one relation.

We are already considering immediate further developments of this work. For example, we will introduce recursion, in order to model some infinite behaviors. We will also combine linear (race-free) interaction with shared (race allowed) interaction, allowing to model service definitions that are globally available and may have concurrent requests. Another interesting problem to be addressed is the dynamic delegation of roles. In our current setting roles are statically annotated in processes. Extending the language with role delegation would allow parties to dynamically assume unforeseen roles at runtime.

Acknowledgments. This work was supported by FCT/MCTES via projects PTD-C/EIA–CCO/105359/2008 and INTERFACES CMU–PT/NGN44–2009–12. Luís Caires participated in the discussions that have led to this work.

References

1. E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In G. Barthe and C. Fournet, editors, *TGC 2007, Third International Symposium on Trustworthy Global Computing, Revised Selected Papers*, volume 4912 of *LNCS*, pages 240–256. Springer, 2008.
2. L. Caires and H. Vieira. Conversation Types. In G. Castagna, editor, *ESOP 2009, 18th European Symposium on Programming, Proceedings*, volume 5502 of *LNCS*, pages 285–300. Springer-Verlag, 2009.
3. L. Caires and H. Vieira. Conversation Types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.
4. P.-M. Deniélou and N. Yoshida. Dynamic Multirole Session Types. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pages 435–446. ACM, 2011.
5. K. Honda. Types for Dyadic Interaction. In E. Best, editor, *CONCUR 1993, 4th International Conference on Concurrency Theory, Proceedings*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.
6. K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In C. Hankin, editor, *ESOP 1998, 7th European Symposium on Programming, Proceedings*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
7. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In G. Necula and P. Wadler, editors, *POPL 2008, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Proceedings*, pages 273–284. ACM Press, 2008.
8. D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
9. V. T. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume 472 of *LNCS*, pages 460–474. Springer, Nov. 1993.
10. H. Vieira, L. Caires, and J. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In S. Drossopoulou, editor, *ESOP 2008, 17th European Symposium on Programming, Proceedings*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.
11. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised Multiparty Session Types. In C.-H. L. Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Proceedings*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.