Conversation Types

Luís Caires and Hugo Torres Vieira

CITI / Departamento de Informática, FCT Universidade Nova de Lisboa, Portugal

Abstract. We present a type theory for analyzing concurrent multiparty interactions as found in service-oriented computing. Our theory introduces a novel and flexible type structure, able to uniformly describe both the internal and the interface behavior of systems, referred respectively as choreographies and contracts in web-services terminology. The notion of conversation builds on the fundamental concept of session, but generalizes it along directions up to now unexplored; in particular, conversation types discipline interactions in conversations while accounting for dynamical join and leave of an unanticipated number of participants. We prove that well-typed systems never violate the prescribed conversation constraints. We also present techniques to ensure progress of systems involving several interleaved conversations, a previously open problem.

1 Introduction

While most issues arising in the context of communication-based software systems do not appear to be new when considered in isolation, the analysis of loosely-coupled distributed systems involving type based discovery, and multiparty collaborations such as those supported by web-services technology raises many challenges and calls for new concepts, specially crafted models, and formal analysis techniques (e.g., [1,2,3,6,7,12]). In previous work [19] we introduced the Conversation Calculus (CC), a π-calculus based model for service-oriented computing that builds on the concepts of process delegation, loose-coupling, and, crucially, conversation contexts.

A key concept for the organization of service-oriented computing systems is the notion of conversation. A conversation is a structured, not centrally coordinated, possibly concurrent, set of interactions between several participants. Then, a conversation context is a medium where partners may interact in a conversation. It can be distributed in many pieces, and processes in any piece may seamlessly talk to processes in the same or any other piece of the same conversation context. Intuitively a conversation context may be seen as a virtual chat room where remote participants exchange messages according to some discipline, while simultaneously engaged in other conversations. Conversation context identities can be passed around, allowing participants to dynamically join conversations. To join an ongoing conversation, a process may perform a remote conversation access using the conversation context identifier. It is then able to participate in the conversation to which it has joined, while being able to interact back with the caller context through the access point. To discipline multiparty conversations we introduce conversation types, a novel and flexible type structure, able to uniformly describe both the internal and the interface behavior of systems, referred respectively as choreographies and contracts in web-services terminology.

We give substantial evidence that our minimal extension to the π-calculus is already effective enough to model and type sophisticated service-based systems, at a fairly high

level of abstraction. Examples include challenging scenarios involving simultaneous multiparty conversations, with concurrency and access to local resources, and conversations with a dynamically changing and unanticipated number of participants, that fall out of scope of other approaches for modeling and typing of service-based systems.

1.1   Conversation Contexts and Conversation Types

We explain the key ideas of our development by going through a motivating example. Consider the following composition of two conversation contexts, named $Buyer$ and $Seller$, modeling a typical service collaboration:

$Buyer$ ◀ [**new** $Seller$ · startBuy $\Leftarrow$ buy!$(prod)$.price?(v)] |
$Seller$ ◀ [$PriceDB$ | **def** startBuy $\Rightarrow$ buy?$(prod)$.askPrice$^\uparrow$!$(prod)$.
readVal$^\uparrow$?(v).price!(v)]

Notice that in the core CC, the bounded communication medium provided by a conversation context may also be used to model a partner local context, avoiding the introduction of a primitive notion of site. The code in $Buyer$ starts a new conversation by calling service startBuy located at $Seller$ using the service instantiation idiom **new** $Seller$ · startBuy $\Leftarrow$ buy!$(prod)$.price?(v). The code buy!$(prod)$.price?(v) describes the role of $Buyer$ in the conversation: a buy message is sent, and afterwards a price message should be received. Upon service instantiation, the system evolves to

$(\boldsymbol{\nu}c)($ $Buyer$ ◀ [c ◀ [buy!$(prod)$.price?(v)]] |
$Seller$ ◀ [$PriceDB$ | c ◀ [buy?$(prod)$.askPrice$^\uparrow$!$(prod)$.
readVal$^\uparrow$?(v).price!(v)])

where c is the fresh name of the newly created conversation (with two pieces). The code

$$\text{buy?}(prod).\text{askPrice}^\uparrow!(prod).\text{readVal}^\uparrow?(v).\text{price!}(v)$$

describes the participation of $Seller$ in the conversation c: a buy message is received, and in the end, price message should be sent. In between, database $PriceDB$ located in the $Seller$ context is consulted through a pair of $\uparrow$ directed message exchanges (askPrice and readVal). Such messages are targeted to the parent conversation ($Seller$), rather than to the current conversation (c).

In our theory, message exchanges *inside* and *at* the interface of subsystems are captured by conversation types, which describe both internal and external participation of processes in conversations. The $Buyer$ and $Seller$ conversation is described by type

$$BSChat \triangleq \boldsymbol{\tau}\text{buy}(Tp).\boldsymbol{\tau}\text{price}(Tm)$$

specifying the two interactions that occur sequentially within the conversation c, first a message buy and after a message price (Tp and Tm represent basic value types).

The τ in, e.g., $\boldsymbol{\tau}\text{buy}(Tp)$ means that the interaction is internal. A declaration such as $\boldsymbol{\tau}\text{buy}(Tp)$ is like an assertion such as buy(Tp) : $Buyer \rightarrow Seller$ in a message sequence chart, or in the global types of [12], except that in our case participant identities are abstracted away, increasing flexibility. In general, the interactions described by a type such as $BSChat$ may be realized in several ways, by different participants. Technically, we specify the several possibilities by a (ternary) merge relation between types, noted $B = B_1 \bowtie B_2$, stating how a behavior B may be projected in two independent matching behaviors B_1 and B_2. In particular, we have (among others) the projection

$$BSChat = ?\,\mathtt{buy}(Tp).!\,\mathtt{price}(Tm) \bowtie !\,\mathtt{buy}(Tp).?\,\mathtt{price}(Tm)$$

The type $?\,\mathtt{buy}(Tp).!\,\mathtt{price}(Tm)$ will be used to type the *Buyer* participation, and the type $!\,\mathtt{buy}(Tp).?\,\mathtt{price}(Tm)$ will be used to type the *Seller* participation (in conversation *BSChat*). Thus, in our first example, the conversation type *BSChat* is decomposed in a pair of "dual" conversation types, as in classical session types [10,11]; this does not need to be always the case, however. In fact, the notion of conversation builds on the fundamental concept of session but extends it along unexplored directions, as we now discuss. Consider a three-party variation (from [6]) of the example above:

$Buyer \blacktriangleleft [$ **new** $Seller \cdot \mathtt{startBuy} \Leftarrow \mathtt{buy}!(prod).\mathtt{price}?(p).\mathtt{details}?(d)\,]$ $\quad|$
$Seller \blacktriangleleft [\ PriceDB\ |$
$\qquad\quad$ **def** $\mathtt{startBuy} \Rightarrow \mathtt{buy}?(prod).\mathtt{askPrice}^\uparrow!(prod).$
$\qquad\qquad\qquad\qquad \mathtt{readVal}^\uparrow?(p).\mathtt{price}!(p).$
$\qquad\qquad\qquad\qquad$ **join** $Shipper \cdot \mathtt{newDelivery} \Leftarrow \mathtt{product}!(prod)\,]$ $\quad|$
$Shipper \blacktriangleleft [$ **def** $\mathtt{newDelivery} \Rightarrow \mathtt{product}?(p).\mathtt{details}!(data)\,]$

The role of *Shipper* is to inform the client on the delivery details. The code is composed of three conversation contexts, representing the three partners *Buyer*, *Seller* and *Shipper*. The system progresses as in the first example: messages buy and price are exchanged between *Buyer* and *Seller* in the fresh conversation. After that, *Shipper* is asked by *Seller*, using idiom **join** $Shipper \cdot \mathtt{newDelivery} \Leftarrow \cdots$, to join the ongoing conversation (till then involving only *Buyer* and *Seller*). The system then evolves to

$(\boldsymbol{\nu} a)(\quad Buyer \blacktriangleleft [\ a \blacktriangleleft [\ \mathtt{details}?(d)\,]\ ]\,|$
$\qquad\qquad Seller \blacktriangleleft [\ a \blacktriangleleft [\ \mathtt{product}!(prod)\,]\ |\ \ldots\ ]\,|$
$\qquad\qquad Shipper \blacktriangleleft [\ a \blacktriangleleft [\ \mathtt{product}?(p).\mathtt{details}!(data)\,]\ ]\ )$

Notice that *Seller does not lose access* to the conversation after asking service $Shipper \cdot \mathtt{newDelivery}$ to join in the current conversation a (partial session delegation). In fact, *Seller* and *Shipper* will interact later on in the very same conversation, by exchanging a product message. Finally, *Shipper* sends a message details directly to *Buyer*. In this case, the global conversation a is initially assigned type

$$BSSChat \triangleq \boldsymbol{\tau}\,\mathtt{buy}(Tp).\boldsymbol{\tau}\,\mathtt{price}(Tm).\boldsymbol{\tau}\,\mathtt{product}(Tp).\boldsymbol{\tau}\,\mathtt{details}(Td)$$

We decompose type *BSSChat* in three "projections" (B_{bu}, B_{se}, and B_{sh}), by means of the *merge* $\bowtie$, first by $BSSChat = B_{bu} \bowtie B_{ss}$, and then by $B_{ss} = B_{se} \bowtie B_{sh}$, where

$$B_{bu} \triangleq ?\,\mathtt{buy}(Tp).!\,\mathtt{price}(Tm).!\,\mathtt{details}(Td)$$
$$B_{ss} \triangleq !\,\mathtt{buy}(Tp).?\,\mathtt{price}(Tm).\boldsymbol{\tau}\,\mathtt{product}(Tp).?\,\mathtt{details}(Td)$$
$$B_{se} \triangleq !\,\mathtt{buy}(Tp).?\,\mathtt{price}(Tm).?\,\mathtt{product}(Tp)$$
$$B_{sh} \triangleq !\,\mathtt{product}(Tp).?\,\mathtt{details}(Td)$$

These various "local" types are merged by our type system in a compositional way, allowing e.g., service startBuy to be assigned type $!\mathtt{startBuy}([B_{ss}])$, and the contribution of each partner in the conversation to be properly determined. At the point where **join** operation above gets typed, the (residual) conversation type corresponding to the participation of *Seller* is typed $\boldsymbol{\tau}\,\mathtt{product}(Tp).?\,\mathtt{details}(Td)$. At this stage, extrusion of the conversation name a to service $Seller \cdot \mathtt{newDelivery}$ will occur, to enable *Shipper* to join in. Notice that the global conversation *BSSChat* discipline will

nevertheless be respected, since the conversation fragment delegated to $Shipper$ is typed
! product(Tp).? details(Td) while the conversation fragment retained by $Seller$ is
typed ? product(Tp). Also notice that since conversation types abstract away from participant identities, the overall conversation type can be projected into the types of the
individual roles in several ways, allowing for different implementations of the roles of a
given conversation (cf. loose-coupling). It is even possible to type systems with an unbounded number of different participants, as needed to type, e.g., a service broker.

Our type system combines techniques from linear, behavioral, session and spatial
types (see [4,11,13,14]): the type structure features prefix $M.B$, parallel composition
$B_1 \mid B_2$, and other operators. Messages M describe external (receive ? / send !) exchanges in two views: with the *caller / parent* conversation ($\uparrow$), and in the *current*
conversation ($\downarrow$). They also describe internal message exchanges (τ). Key technical
ingredients in our approach to conversation types are the amalgamation of global types
and of local types (in the general sense of [12]) in the same type language, and the
definition of a merge relation ensuring, by construction, that participants typed by the
projected views of a type will behave well under composition. Merge subsumes duality,
in the sense that for each τ-free B there are types $\overline{B}, B'$ such that $B \bowtie \overline{B} = \tau(B')$,
so sessions are special cases of conversations. But merge of types allows for extra flexibility on the manipulation of projections of conversation types, in an open-ended way,
as illustrated above. In particular, our approach allows fragments of a conversation type
(e.g., a choreography) to be dynamically distributed among participants, while statically
ensuring that interactions follow the prescribed discipline.

The technical contributions of this work may be summarized as follows. First, we
define the new notion of conversation type. Conversation types are a generalization of
session types to loosely-coupled, possibly concurrent, multiparty conversations, allowing mixed global / local behavioral descriptions to be expressed at the same level, while
supporting the analysis of systems with dynamic delegation of fragments of ongoing
conversations. Second, we advance new techniques to certify safety and liveness properties of service-based systems. We propose a type system for assigning conversation
types to core CC systems. Processes that get past our typing rules are ensured to be free
of communication errors, and races on plain messages (Corollary 3.6): this also implies
that well-typed systems enjoy a conversation fidelity property (i.e., all conversations
follow the prescribed protocols). Finally, we present techniques to establish progress of
systems with several interleaved conversations (Theorem 4.4), exploiting the combination of conversation names with message labels in event orderings, and, more crucially,
propagation of orderings in communications, solving a previously open problem.

Additional examples, complete definitions and detailed proofs can be found in [5].

2   The Core Conversation Calculus

In this section, we present the syntax and operational semantics of the core Conversation Calculus (CC) [19]. The core CC extends the π-calculus [16] static fragment with
the conversation construct $n \blacktriangleleft [P]$, and replaces channel based communication with
context-sensitive message based communication. For simplicity, we use here a monadic
version. The syntax of the calculus is defined in Fig. 1. We assume given an infinite set
of names Λ, an infinite set of variables $\mathcal{V}$, an infinite set of labels $\mathcal{L}$, and an infinite set

$$a, b, c, \ldots \in \Lambda \quad \textit{(Names)} \qquad\qquad d \quad ::= \downarrow \mid \uparrow \quad \textit{(Directions)}$$
$$x, y, z, \ldots \in \mathcal{V} \quad \textit{(Variables)}$$
$$n, m, o \ldots \in \Lambda \cup \mathcal{V} \qquad\qquad \alpha, \beta ::= l^d!(n) \quad \textit{(Output)}$$
$$l, s \ldots \quad \in \mathcal{L} \quad \textit{(Labels)} \qquad\qquad \mid \quad l^d?(x) \quad \textit{(Input)}$$
$$\mathcal{X}, \mathcal{Y}, \ldots \in \chi \quad \textit{(Process Vars)} \qquad\qquad \mid \quad \textbf{this}(x) \ \textit{(Conversation Awareness)}$$

$$P, Q ::= \mathbf{0} \qquad\qquad \textit{(Inaction)} \qquad\quad \mid \textbf{rec}\, \mathcal{X}.P \qquad \textit{(Recursion)}$$
$$\mid \quad P \mid Q \qquad \textit{(Parallel Composition)} \quad \mid \mathcal{X} \qquad\qquad \textit{(Variable)}$$
$$\mid \quad (\boldsymbol{\nu} a)P \qquad \textit{(Name Restriction)} \qquad \mid \Sigma_{i \in I}\, \alpha_i.P_i \quad \textit{(Prefix Guarded Choice)}$$
$$\mid \quad n \blacktriangleleft [P] \qquad \textit{(Conversation Access)}$$

Fig. 1. The core Conversation Calculus

of process variables χ. The static fragment is defined by the inaction $\mathbf{0}$, parallel composition $P \mid Q$, name restriction $(\boldsymbol{\nu} a)P$ and recursion $\textbf{rec}\, \mathcal{X}.P$. The conversation access construct $n \blacktriangleleft [P]$, allows a process to interact, as specified by P, in conversation n.

Communication is expressed by the guarded choice construct $\Sigma_{i \in I}\, \alpha_i.P_i$, meaning that the process may select some initial action α_i and then progress as P_i. Communication actions are of two forms: $l^d!(n)$ for sending messages and $l^d?(x)$ for receiving messages. Message communication is defined by the label l and the direction d. There are two message directions: $\downarrow$ (read "here") meaning that the interaction should take place in the current conversation or $\uparrow$ (read "up") meaning that the interaction should take place in the enclosing (caller) conversation. To lighten notation we omit the $\downarrow$ in the $\downarrow$-directed messages without any ambiguity. A basic action may also be of the form $\textbf{this}(x)$, allowing a process to dynamically access the name of the current conversation. Notice that message labels (from $l \in \mathcal{L}$) are not names but free identifiers (cf. record labels or XML tags), and therefore are not subject to fresh generation, restriction or binding. Only conversation names (in Λ) may be subject to binding, and freshly generated via $(\boldsymbol{\nu} a)P$. The distinguished occurrences of a, x, x and $\mathcal{X}$ are binding occurrences in $(\boldsymbol{\nu} a)P$, $l^d?(x).P$, $\textbf{this}(x).P$ and $\textbf{rec}\, \mathcal{X}.P$, respectively. The sets of free ($\mathit{fn}(P)$) and bound ($\mathit{bn}(P)$) names, free variables ($\mathit{fv}(P)$), and free process variables ($\mathit{fpv}(P)$) in a process P are defined as expected. We implicitly identify α-equivalent processes.

The operational semantics of the core CC is defined by a labeled transition system. For clarity, we split the presentation in two sets of rules, one (in Fig. 3) containing the rules for the basic operators, another (in Fig. 4) grouping the rules specific to the conversations. A transition $P \xrightarrow{\lambda} Q$ states that process P may evolve to process Q by performing the action represented by the transition label λ. Transition labels (λ) and actions (σ) are defined in Fig. 2. An action τ denotes an internal communication, actions $l^d!(a)$ and $l^d?(a)$ represent communications with the environment, and $\texttt{this}$ represents a conversation identity access; these correspond to the basic actions a process may perform in the context of a given conversation. To capture the observational semantics of processes [19], transition labels register not only the action but also the conversation where the action takes place. So, a transition label λ containing $c\, \sigma$ is said to be *located at* conversation c (or just *located*), otherwise is said to be *unlocated*. In $(\boldsymbol{\nu} a)\lambda$ the distinguished occurrence of a is bound with scope λ (cf., the π-calculus bound output actions). For a communication label λ we denote by $\overline{\lambda}$ the dual matching label obtaining

$$\sigma ::= \tau \mid l^d!(a) \mid l^d?(a) \mid \texttt{this} \quad \text{(Transition Labels)} \qquad \lambda ::= c\,\sigma \mid \sigma \mid (\nu a)\lambda \quad \text{(Actions)}$$

Fig. 2. Transition Labels and Actions

$$l^d!(a).P \xrightarrow{l^d!(a)} P \; (out) \qquad l^d?(x).P \xrightarrow{l^d?(a)} P\{x/a\} \; (inp) \qquad \frac{\alpha_j.P_j \xrightarrow{\lambda} Q \quad j \in I}{\Sigma_{i \in I}\, \alpha_i.P_i \xrightarrow{\lambda} Q}(sum)$$

$$\frac{P \xrightarrow{\lambda} Q \quad a = out(\lambda)}{(\nu a)P \xrightarrow{(\nu a)\lambda} Q}(opn) \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}(com) \qquad \frac{P \xrightarrow{(\nu a)\overline{\lambda}} P' \quad Q \xrightarrow{\lambda} Q'}{P \mid Q \xrightarrow{\tau} (\nu a)(P' \mid Q')}(clo)$$

$$\frac{P \xrightarrow{\lambda} Q \quad a \notin na(\lambda)}{(\nu a)P \xrightarrow{\lambda} (\nu a)Q}(res) \qquad \frac{P \xrightarrow{\lambda} Q}{P \mid R \xrightarrow{\lambda} Q \mid R}(par) \qquad \frac{P\{\mathcal{X}/\textbf{rec}\,\mathcal{X}.P\} \xrightarrow{\lambda} Q}{\textbf{rec}\,\mathcal{X}.P \xrightarrow{\lambda} Q}(rec)$$

Fig. 3. Operational Semantics: Basic Operators (π-calculus)

$$\frac{P \xrightarrow{\lambda^\uparrow} Q}{c \blacktriangleleft [P] \xrightarrow{\lambda^\downarrow} c \blacktriangleleft [Q]}(her) \qquad \frac{P \xrightarrow{\lambda^\downarrow} Q}{c \blacktriangleleft [P] \xrightarrow{c \cdot \lambda^\downarrow} c \blacktriangleleft [Q]}(loc) \qquad \frac{P \xrightarrow{a\,\lambda^\downarrow} Q}{c \blacktriangleleft [P] \xrightarrow{a\,\lambda^\downarrow} c \blacktriangleleft [Q]}(thr)$$

$$\frac{P \xrightarrow{\tau} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]}(tau) \qquad \textbf{this}(x).P \xrightarrow{c\,\texttt{this}} P\{x/c\} \; (thi) \qquad \frac{P \xrightarrow{c\,\texttt{this}} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]}(thl)$$

$$\frac{P \xrightarrow{\sigma} P' \quad Q \xrightarrow{c\,\overline{\sigma}} Q'}{P \mid Q \xrightarrow{c\,\texttt{this}} P' \mid Q'}(tco) \qquad \frac{P \xrightarrow{\sigma} P' \quad Q \xrightarrow{(\nu a)c\,\overline{\sigma}} Q'}{P \mid Q \xrightarrow{c\,\texttt{this}} (\nu a)(P' \mid Q')}(tcl)$$

Fig. 4. Operational Semantics: Conversation Operators

by swapping inputs with outputs, such that $\overline{l^d!(a)} = l^d?(a)$ and $\overline{l^d?(a)} = l^d!(a)$. By $na(\lambda)$ we denote the free and bound names and by $bn(\lambda)$ the bound names of λ.

Transition rules presented in Fig. 3 closely follow the ones for the π-calculus [18] and should be fairly clear to a reader familiar with mobile process calculi. For example, rule (opn) corresponds to the bound output or extrusion rule, in which a bound name a is extruded to the environment in an output message λ: we define $out(\lambda) = a$ if $\lambda = l^d!(a)$ or $\lambda = c\,l^d!(a)$ and $c \neq a$. We discuss the intuitions behind the rules for conversation contexts (Fig. 4). In (her) an $\uparrow$ directed message (to the caller conversation) becomes $\downarrow$ (in the current conversation), after passing through the conversation access boundary. We note by λ^d a transition label λ^d containing the direction d ($\uparrow, \downarrow$), and by $\lambda^{d'}$ the label obtained by replacing d by d' in λ^d (e.g., if $\lambda^\uparrow$ is $\texttt{askPrice}^\uparrow?(a)$ then $\lambda^\downarrow$ is $\texttt{askPrice}^\downarrow?(a)$). In (loc) an unlocated $\downarrow$ message (in the current conversation) gets explicitly located at the conversation c in which it originates. Given an unlocated label λ, we represent by $c \cdot \lambda$ the label obtained by locating λ at c (e.g., if $\lambda^\downarrow$ is $\texttt{askPrice}^\downarrow?(p)$ then $c \cdot \lambda^\downarrow$ is $c\,\texttt{askPrice}^\downarrow?(p)$). In (thr) an already located communication label transparently crosses some conversation boundary, and likewise for a τ label in (tau). In (thi) a $\texttt{this}$ label reads the conversation identity, and originates a $c\,\texttt{this}$ label. A $c\,\texttt{this}$ labeled transition may only progress inside the c conversation, as expressed in (thl), where a $\texttt{this}$ label matches the enclosing conversation. In (tco)

$$\textbf{def } s \Rightarrow P \triangleq s?(x).x \blacktriangleleft [P] \qquad \textbf{new } n \cdot s \Leftarrow Q \triangleq (\boldsymbol{\nu}c)(n \blacktriangleleft [s!(c)] \mid c \blacktriangleleft [Q])$$

$$\textbf{join } n \cdot s \Leftarrow Q \triangleq \textbf{this}(x).(n \blacktriangleleft [s!(x)] \mid Q)$$

Fig. 5. Service Idioms

and (tcl) an unlocated communication matches a communication located at c, originating a c this label, thus ensuring the interaction occurs in the given conversation c, as required. The reduction relation of the core CC, noted $P \to Q$, is defined as $P \xrightarrow{\boldsymbol{\tau}} Q$.

Using conversation contexts and the basic message based communication mechanisms, useful programming abstractions for service-oriented systems may be idiomatically defined in the core CC, namely service definition and instantiation constructs (redundantly introduced as primitives in [19]) and also a new conversation join construct, as shown in Fig. 5. The **def** form publishes a service definition. There are two ways of using such a service definition: either by the **new** form, which establishes a fresh conversation between client and server; or by the **join** form which instead passes to the service provider the identity of the current conversation, allowing parties to ask other service providers to join in on ongoing conversations. Both usages refer the service name s and the conversation n where the service is available at, thus service definitions must be located in order to be instantiated (as e.g., methods must reside in objects).

3 Type System

In this section we formally present our type system for the core CC. As already motivated in the Introduction, our types specify the message protocols that flow between and within conversations. The syntax for the types is shown in Fig. 6. Typing judgments have the form $P :: T$, where T is a process type. Intuitively, a type judgement $P :: T$ states that if process P is placed in an environment that complies with type T, then the resulting system is safe, in a sense to be made precise below (Corollary 3.6). In general, a process type T has the form $L \mid B$, where L is a *located type* and B is a *behavioral type*. An atomic located type associates a conversation type C to a conversation name n. Conversation types C are given by $[B]$, where B specifies the message interactions that may take place in the conversation. Behavioral types B include the branch and the choice constructs ($\&_{i \in I}\{M_i.B_i\}$ and $\oplus_{i \in I}\{M_i.B_i\}$, respectively), specifying processes that can branch in either of the $M_i.B_i$ behaviors and choose between one of the $M_i.B_i$ behaviors, respectively. Prefix $M.B$ specifies a process that sends, receives, or internally exchanges a message M before proceeding with behavior B. We also have parallel composition $B_1 \mid B_2$, inaction $\mathbf{0}$, and recursion. Message types M are specified by a polarity p (either output !, input ? or internal action $\boldsymbol{\tau}$), a pair label-direction l^d, and the type C of the name communicated in the message. For typing purposes, we split the set of message labels $\mathcal{L}$ into shared $\mathcal{L}_\star$ and plain $\mathcal{L}_p$ labels (plain labeled messages will be used linearly, and shared labeled messages will be used exponentially). Notice that a message M may refer to an *internal* exchange between two partners, if it is of the form $\boldsymbol{\tau}l^\downarrow(C)$. The unlocated part B of a process P type $L \mid B$ specifies the behavior of P in the current conversation (taking place in the context where P resides).

$$B ::= B_1 \mid B_2 \ \Big| \ \mathbf{0} \ \Big| \ \mathtt{rec}\,\mathcal{X}.B \ \Big| \ \mathcal{X} \ \Big| \oplus_{i \in I} \{M_i.B_i\} \ \Big| \ \&_{i \in I}\{M_i.B_i\} \quad \textit{(Behavioral)}$$

$$M ::= p\, l^d(C) \ \ \textit{(Message)} \quad p ::= ! \ \Big| \ ? \ \Big| \ \tau \ \ \textit{(Polarity)} \quad C ::= [B] \ \ \textit{(Conversation)}$$

$$L ::= n : C \ \Big| \ L_1 \mid L_2 \ \Big| \ \mathbf{0} \ \ \textit{(Located)} \qquad\qquad T ::= L \mid B \ \ \textit{(Process)}$$

Fig. 6. Syntax of Types

$$\mathtt{rec}\,\mathcal{X}.T \equiv T\{\mathcal{X}/\mathtt{rec}\,\mathcal{X}.T\} \ \ (1) \qquad\qquad n : [B_1 \mid B_2] \equiv n : [B_1] \mid n : [B_2] \ \ (2)$$

$$M.B_1 \mid B_2 <: M.(B_1 \mid B_2) \ \ (M \# B_2) \ \ (3) \qquad\qquad \downarrow B \mid \uparrow B <: B \ \ (4)$$

$$\frac{M_i.B_i <: M_i'.B_i' \ \ (i \in I)}{\oplus_{i \in I}\{M_i.B_i\} <: \oplus_{i \in I}\{M_i'.B_i'\}}(5) \qquad \frac{M_i.B_i <: M_i'.B_i' \ \ (i \in I) \quad I \subseteq J}{\&_{i \in J}\{M_i.B_i\} <: \&_{i \in I}\{M_i'.B_i'\}}(6)$$

Fig. 7. Selected Subtyping Rules

$$\frac{P :: L \mid B}{n \blacktriangleleft [P] :: (L \bowtie n : [\downarrow B]) \mid loc(\uparrow B)}(piece) \qquad \frac{P :: L \mid B_1 \mid x : [B_2] \ \ (x \notin dom(L))}{\mathbf{this}(x).P :: L \mid (B_1 \bowtie B_2)}(this)$$

$$\frac{P_i :: L \mid B_i \mid x_i : C_i \ \ (x_i \notin dom(L))}{\Sigma_{i \in I}\, l_i^d?(x_i).P_i :: L \mid \oplus_{i \in I}\{!l_i^d(C_i).B_i\}}(inp) \qquad \frac{P :: L \mid B}{l^d!(n).P :: (L \bowtie n : C) \mid ?l^d(C).B}(out)$$

$$\frac{P :: T \mid a : [B] \ \ (closed(B), a \notin dom(T))}{(\boldsymbol{\nu}a)P :: T}(res) \qquad \frac{P :: T_1 \quad Q :: T_2}{P \mid Q :: T_1 \bowtie T_2}(par) \qquad \mathbf{0} :: \boldsymbol{\tau}(L)\ (stop)$$

$$\frac{P :: L_M \mid B\langle\mathcal{X}\rangle}{\mathbf{rec}\,\mathcal{X}.P :: \star L_M \mid \mathtt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle}(rec) \qquad \overline{\mathcal{X} :: \mathcal{X}}(var) \qquad \frac{T <: T' \quad P :: T'}{P :: T}(sub)$$

Fig. 8. Typing Rules

We introduce some auxiliary notations and notions. We abbreviate both $\oplus\{M.B\}$ and $\&\{M.B\}$ with $M.B$. We write M for $M.\mathbf{0}$, and $p\,l(C)$ for $p\,l^\downarrow(C)$. An important auxiliary notion is the *projection $d(B)$ in direction d of a behavioral type B*. It consists in the selection of all messages that have the given direction d while filtering out ones in the other direction, offering a partial view of behavior B from the d viewpoint. For instance, if $B \triangleq\ !\,\mathtt{buy}(Tp).?\,\mathtt{askPrice}^\uparrow(Tp).!\,\mathtt{readVal}^\uparrow(Tm).?\,\mathtt{price}(Tm)$ then $\downarrow(B) =\ !\,\mathtt{buy}(Tp).?\,\mathtt{price}(Tm)$ and $\uparrow(B) =\ ?\,\mathtt{askPrice}^\uparrow(Tp).!\,\mathtt{readVal}^\uparrow(Tm)$. We also write, e.g., $\uparrow B$ for $\uparrow(B)$, to lighten notation. Informally, we refer to $\downarrow B$ as the "here interface" of B, and likewise for $\uparrow B$ as the "up interface". If p is a polarity (!, ?, τ), we denote by $p(B)$ the projection type that selects all messages that have polarity p.

Types are related by the subtyping relation $<:$, for which we depict a selection of rules in Fig. 7. The subtyping rules express expected relationships of types, such as the commutative monoid rules for $(-\mid -, \mathbf{0})$, congruence principles, and the split rule (2). For types T_1 and T_2 we write $T_1 \equiv T_2$ if $T_1 <: T_2$ and $T_2 <: T_1$. A key subtyping principle is (4), that allows a behavioral type to be decomposed (in the subtype) in its two projections according to the message directions $\downarrow$ and $\uparrow$. Another important subtyping principle is (3), that allows a message to be serialized (in the supertype). Notice we do not allow width subtyping in choice type (5). Essentially we can not forget some choices in the choice type, as this would allow undesired matches between choice and

branch types: if the environment expected by a process does not fully reveal the choices it may take, then placing the process in such environment may lead to unexpected (not described by the type) behaviors (cf., [7], where a related issue is addressed).

We may now present our typing rules in Fig. 8. They rely on several auxiliary operations and predicates on types. The key ones are predicate *apartness* $T_1 \# T_2$ and relation *merge* $T = T_1 \bowtie T_2$. Intuitively, two types are apart when they may type subsystems that may be safely composed without undesirable interferences. Apartness is defined by checking disjointness of sets of message labels, more precisely it asserts disjointness of plain ("linear") types, and consistency of shared ("exponential") types, w.r.t. conversations. The merge relation relates two types T_1 and T_2 to some composition, so that if $T = T_1 \bowtie T_2$ then T is a particular behavioral combination of the types T_1 and T_2. Merge is defined not only in terms of spatial separation, but also, and crucially, in terms of synchronization / shuffle of behavioral traces. Notice that there might not be T such that $T = T_1 \bowtie T_2$. On the other hand, if such T exists, we use $T_1 \bowtie T_2$ to non-deterministically denote any such T (e.g., in conclusions of type rules). Intuitively, $T = T_1 \bowtie T_2$ holds if T_1 and T_2 may safely synchronize or interleave so as to produce behavioral type T. We formally define merge and apartness in the Appendix, but this informal understanding already allows us to explain the key typing rules.

Rule $(piece)$ types a (piece of a) conversation. Process P expects some located behavior L, and some unlocated behavior B in the current conversation. The type in the conclusion is obtained by merging the type L with a type that describes the behavior of the new conversation piece, in parallel with the type of the toplevel conversation, the now current conversation. Essentially, the type of the projections in the two directions is collected appropriately: the "here" projection $\downarrow B$ is the behavior in conversation n, and the "up" projection $\uparrow$ of P becomes the "here" behavior at the toplevel conversation, via $loc(\uparrow B)$ which sets the direction of all messages to $\downarrow$. Rule $(this)$ types the conversation awareness primitive, requiring behavior B_2 of conversation x to be a separate (in general, just partial) view of the current conversation. This allows to bind the current conversation to name x, and possibly sent to other parties that may need to join it.

In rule (inp) the premise states that processes P_i require some located behavior L, some current conversation behavior B_i, and some behavior at conversation x_i ($dom(L)$ denotes the set of conversation identifiers of located type L). Then, the conclusion states that the input summation process is well-typed under type L, with the behavior interface becoming the choice of the types of the continuations prefixed by the messages $! \, l_i^d(C_i)$, where the output capability $!$ corresponds to the message capability expected from the external environment (as well as the choice that also refers to the capability of performing a choice expected from the external environment). In rule (out) notice that the context type is a separate $\bowtie$ view of the context, which means that the type being sent may actually be some separate part of the type of some conversation, which will be (partially) delegated away. This mechanism is crucial to allow external partners to join in on ongoing conversations in a disciplined way. The behavioral interface of the output prefixed process is an input type, as an input is expected from the external environment.

In rule (res) we use $closed(B)$, to avoid hiding conversation names where unmatched communications still persist (necessary to ensure deadlock absence). *closed* behavioral types characterize processes that have matching receives for all sends.

$$\tau\, l^{\downarrow}(C).B \to B \qquad \frac{T_1 \to T_2}{T_1 \mid T_3 \to T_2 \mid T_3} \qquad \frac{T_1 \to T_2}{n : [T_1] \to n : [T_2]}$$

Fig. 9. Type Reduction Selected Rules

Definition 3.1. *A behavioral type B is closed, noted $closed(B)$, if the polarities of message types in B are only τ messages or outputs ! on shared labels.*

In rule (rec) we denote by $B\langle \mathcal{X} \rangle$ a behavioral type with a single occurrence of $\mathcal{X}$. We use $\star M$ as an abbreviation of $\texttt{rec}\ \mathcal{X}.M.\mathcal{X}$. Then, by L_M we denote a located type of the form $n_1 : [M_1] \mid \ldots \mid n_k : [M_k]$, and by $\star L_M$ we denote $n_1 : [\star M_1] \mid \ldots \mid n_k : [\star M_k]$. The rule states that the process is well typed under an environment that persistently offers messages M_i under conversations n_i, and persistently offers behavior B in the current conversation. The message types M_i must be defined with shared labels with polarity ?. We now present our main soundness results. Subject reduction is defined using a notion of reduction on types, since a reduction step at the process level may require a modification in the type. Fig. 9 shows a selection of type reduction rules.

Theorem 3.2 (Subject Reduction). *Let P be a process and T a type such that $P :: T$. If $P \to Q$ then there is T' such that $T \to T'$ and $Q :: T'$.*

Our safety result asserts that certain error processes are unreachable from well-typed processes. To define error processes we introduce static process contexts.

Definition 3.3 (Static context). *Static process contexts, noted $\mathcal{C}[\cdot]$, are defined as:*
$$\mathcal{C}[\cdot] ::= (\boldsymbol{\nu}a)\mathcal{C}[\cdot] \mid P \mid \mathcal{C}[\cdot] \mid c \blacktriangleleft [\mathcal{C}[\cdot]] \mid \textbf{rec}\ \mathcal{X}.\mathcal{C}[\cdot] \mid \cdot$$

We also use $w(\lambda)$ to denote the sequence $c\, l^d$ of elements in the action label λ, for example $w((\nu a)c\, l^d!(a)) = c\, l^d$ and $w((\nu a)l^d!(a)) = l^d$.

Definition 3.4 (Error Process). *P is an error process if there is a static context $\mathcal{C}$ with $P = \mathcal{C}[Q \mid R]$ and there are $Q', R', \lambda, \lambda'$ such that $Q \xrightarrow{\lambda} Q'$, $R \xrightarrow{\lambda'} R'$ and $w(\lambda) = w(\lambda')$, $\overline{\lambda} \neq \lambda'$ and $w(\lambda)$ is not a shared label.*

A process is not an error only if for each possible immediate interaction in a plain message there is at most a single sender and a single receiver.

Proposition 3.5 (Error Freeness). *Let P be such that $P :: T$. Then P is not an error.*

By subject reduction (Theorem 3.2), we conclude that any process reachable from a well-typed process is not an error. We note by $\xrightarrow{*}$ the reflexive transitive closure of $\to$.

Corollary 3.6 (Type Safety). *Let P be a process such that $P :: T$ for some T. If there is Q such that $P \xrightarrow{*} Q$, then Q is not an error process.*

Our type safety result ensures that, in any reduction sequence arising from a well-typed process, for each plain-labeled message ready to communicate there is always at most a unique input/output outstanding synchronization. More: arbitrary interactions in shared labels do not invalidate this invariant. Another consequence of subject reduction is that

any message exchange inside the process must be explained by a τM prefix in the related conversation type (via type reduction), thus implying conversation fidelity, i.e., all conversations follow the protocols prescribed by their types. In the expected polyadic extension of core CC and type system we would also exclude arity mismatch errors.

Before closing this section, we show the types of the *Buyer-Seller-Shipper* example (*BuySys*) of the Introduction, assuming the expected typing for process *PriceDB*.

$$B_{ss} \triangleq \, ! \, \texttt{buy}(Tp).? \, \texttt{price}(Tm).\tau \, \texttt{product}(Tp).? \, \texttt{details}(Td)$$

$$B_{sh} \triangleq \, ! \, \texttt{product}(Tp).? \, \texttt{details}(Td) \quad B_{db} \triangleq \tau \, \texttt{askPrice}(Tp).\tau \, \texttt{readVal}(Tm)$$

$$BuySys :: Seller : [\tau \, \texttt{startBuy}([B_{ss}]).B_{db}] \mid Shipper : [\tau \, \texttt{newDelivery}([B_{sh}])]$$

4   Progress

In this section, we develop an auxiliary proof system to enforce progress properties on systems. As most traditional deadlock detection methods (e.g., see [9,15,17]), we build on the construction of a well-founded ordering on events. In our case, events are message synchronizations occurring under conversations. Thus the ordering must relate pairs (conversation identifier,message label), which allows us to cope with systems with multiple interleaved conversations, and back and forth communications between two or more conversations in the same thread. Since references to conversations can be passed in message synchronization, the ordering also considers for each message the ordering associated to the conversation which is communicated in the message. These ingredients allow us to check that all events in the continuation of a prefix are of greater rank than the event of the prefix, thus guaranteeing the event dependencies are acyclic.

The proof system, for which we depict a selection of rules in Fig. 10, is presented by means of judgments of the form $\Gamma \vdash_\ell P$. The judgment $\Gamma \vdash_\ell P$ states that the communications of process P follow a well determined order, specified by Γ. In such a judgment we note by Γ an event ordering: a well-founded partial order of events. Events consist of both a pair (name,label) $((\Lambda \cup \mathcal{V}) \times \mathcal{L})$ and an event ordering abstraction, i.e., a parameterized event ordering, noted $(x)\Gamma$ (where x is a binding occurrence with scope Γ), which represents the ordering of the conversation which is to be communicated in the message. We range over events with $e, e_1, \dots$ and denote by $n.l.(x)\Gamma$ an event where n is the conversation name, l is the message label and $(x)\Gamma$ is the event ordering abstraction. In $\Gamma \vdash_\ell P$, we use ℓ to keep track of the names of the current conversation ($\ell(\downarrow)$) and of the enclosing conversation ($\ell(\uparrow)$); if $\ell = (n, m)$ then $\ell(\uparrow) = n$ and $\ell(\downarrow) = m$. We define some operations over event orderings Γ. The event ordering $\Gamma \setminus n$ is obtained from Γ by removing all events that have n as conversation name, while

$$\frac{(\ell(d).l_i.(y)\Gamma'_i \perp \Gamma) \cup \Gamma'_i\{y/x_i\} \vdash_\ell P_i}{\Gamma \vdash_\ell \Sigma_{i \in I} \, l_i^{d}?(x_i).P_i} (inp) \qquad\qquad \frac{\Gamma \vdash_\ell P}{\Gamma \setminus a \vdash_\ell (\nu a)P} (res)$$

$$\frac{(\ell(d).l.(x)\Gamma' \perp \Gamma) \vdash_\ell P \quad \Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \perp \Gamma)}{\Gamma \vdash_\ell l^d!(n).P} (out) \qquad \frac{\Gamma \vdash_{(\ell(\downarrow),n)} P}{\Gamma \vdash_\ell n \blacktriangleleft [P]} (piece)$$

Fig. 10. Selection of Proof Rules for Progress

keeping the overall ordering. By $e_1 \prec_\Gamma e_2$ we denote that e_1 is smaller than e_2 under Γ, and by $dom(\Gamma)$ we denote the set of events which are related by Γ.

Definition 4.1. *Given event e and event ordering Γ such that $e \in dom(\Gamma)$ we define $e \perp \Gamma$ as the subrelation of Γ where all events are greater than e, as follows:*

$$e \perp \Gamma \triangleq \{(e_1 \prec e_2) \mid (e_1 \prec_\Gamma e_2) \wedge (e \prec_\Gamma e_1)\}$$

We briefly discuss the key proof rules of Fig. 10. Rules (inp) and (out) ensure communications originating in the continuations, including the ones in the conversation being received/sent, are of a greater order. In rule (inp) the event ordering considered in the premise is such that it contains elements greater than $\ell(d).l_i.(x)\Gamma'$, the event associated with the input, enlarged with the event ordering abstraction $(x)\Gamma'$ of the event associated with the input, where the bound x is replaced by the input parameter x_i. In rule (out) the event ordering considered in the premise is such that it contains elements greater than $\ell(d).l.(x)\Gamma'$, the event associated to the output. Also the premise states that the event ordering abstraction $(x)\Gamma'$ of the event associated to the output is a subrelation of the event ordering Γ, when the parameter x is replaced by the name to be sent in the output (n). We may now present our progress results.

Theorem 4.2 (Preservation of Event Ordering). *Let P be a well typed process $P :: T$ and Γ an event ordering such that $\Gamma \vdash_\ell P$. If there is Q such that $P \to Q$ then $\Gamma \vdash_\ell Q$.*

We define finished processes so to distinguish stable from stuck processes.

Definition 4.3 (Finished Process). *P is finished if for any static context $\mathcal{C}$ and process Q such that $P = \mathcal{C}[Q]$ then Q has no immediate output $(\lambda = l^d!(a))$ transitions.*

Finished processes have no reductions and also have no pending requests (outputs), hence are in a stable state, but may have some active inputs (e.g., persistent definitions).

Theorem 4.4 (Progress). *Let P be a well typed process such that $P :: T$, where $closed(T)$, and Γ an event ordering and a, b names $(a, b \notin fn(P))$ such that $\Gamma \vdash_{(a,b)} P$. If P is not a finished process then there is Q such that $P \to Q$.*

Theorem 4.4 ensures that well-typed and well-ordered processes never get stuck on an output that has no matching input. This property entails that services are always available upon request and protocols involving interleaving conversations never get stuck. In the light of these results, given we can show that the $Buyer$-$Seller$-$Shipper$ example of the Introduction has such an event ordering, we can assert it enjoys the progress property. Notice that $Seller$ leaves and reenters the received conversation, to consult $PriceDB$; such a scenario is not in the scope of other progress techniques for sessions.

5   Related Work

Behavioral Type Systems As most behavioral type systems (see [8,13]), we describe a conversation behavior by some kind of abstract process. However, fundamental ideas behind the conversation type structure, in particular the composition / decomposition of behaviors via merge, as captured, e.g., in the typing rule for $P \mid Q$, and used to model delegation of conversation fragments, have not been explored before.

Binary Sessions. The notion of conversation originates in that of session (introduced in [10,11]). Sessions are a medium for two-party interaction, where session participants access the session through a session endpoint. On the other hand conversations are also a single medium but for multiparty interaction, where any of the conversation participants accesses the conversation through a conversation endpoint (pieces). Session channels support single-threaded interaction protocols between the two session participants. Conversation contexts, on the other hand, support concurrent interaction protocols between multiple participants. Sessions always have two endpoints, created at session initialization. Participants can delegate their participation in a session, but the delegation is full as the delegating party loses access to the session. Conversations also initially have two endpoints. However the number of endpoints may increase (decrease) as participants join in on (leave) ongoing conversations. Participants can ask a party to join in on a conversation and not lose access to it (partial delegation). Since there are only two session participants, session types may describe the entire protocol by describing the behavior of just one of the participants (the type of the other participant is dual). Conversations types, on the other hand, describe the interactions between multiple parties so they specify the entire conversation protocol (a choreography description) that decomposes in the types of the several participants (e.g., $B_t = B_{bu} \bowtie B_{se} \bowtie B_{sh}$).

Multiparty Sessions. The goals of the works [2,12] are similar to ours. To support multiparty interaction, [12] considers multiple session channels, while [2] considers a multiple indexed session channel, both resorting to multiple communication pathways. We follow an essentially different approach, by letting a single medium of interaction support concurrent multiparty interaction via labeled messages. In [2,12] sessions are established simultaneously between several parties through a multicast session request. As in binary sessions, session delegation is full so the number of initial participants is kept invariant, unlike in conversations where parties can keep joining in. The approach of [2,12] builds on two-level descriptions of service collaborations (global and local types), first introduced in a theory of endpoint projection [6]. The global types mention the identities of the communicating partners, being the types of the individual participants projections of the global type with respect to these annotations. Our merge operation $\bowtie$ is inspired in the idea of projection [6], but we follow a different approach where "global" and "local" types are treated at the same level in the type language and types do not explicitly mention the participants identities, so that each given protocol may be realized by different sets of participants, provided that the composition of the types of the several participants produce (via $\bowtie$) the appropriate invariant. Our approach thus supports conversations with dynamically changing number of partners, ensuring a higher degree of loose-coupling. We do not see how this could be encoded in the approach of [12]. On the other hand, we believe that core CC with conversation types can express the same kind of systems as [12].

Progress in Session Types. There are a number of progress studies for binary sessions (e.g., [1,3,9]), and for multiparty sessions [2,12]. The techniques of [2,9] are nearer to ours as orderings on channels are imposed to guarantee the absence of cyclic dependencies. However they disallow processes that get back to interact in a session after interacting in another, and exclude interleaving on received sessions, while we allow processes that re-interact in a conversation and interleave received conversations.

6   Concluding Remarks

We have presented a core typed model for expressing and analyzing service and communication based systems, building on the notions of conversation, conversation context, and context-dependent communication. We believe that, operationally, the core CC can be seen as a specialized idiom of the π-calculus [18], if one considers π extended with labeled channels or pattern matching. However, for the purpose of studying communication disciplines for service-oriented computing and their typings, it is much more convenient to adopt a primitive conversation context construct, for it allows the conversation identity to be kept implicit until needed.

Conversation types elucidate the intended dynamic structure of conversations, in particular how freshly instantiated conversations may dynamically engage and dismiss participants, modeling in a fairly abstract way, the much lower level correlation mechanisms available in Web-Services technology. Conversation types also describe the information and control flow of general service-based collaborations, in particular they may describe the behavior of orchestrations and choreographies. We have established subject reduction and type safety theorems, which entail that well-typed systems follow the defined protocols. We also have studied a progress property, proving that well-ordered systems never get stuck, even when participants are engaged in multiple interleaved conversations, as is often the case in applications. Conversation types extend the notion of binary session types to multiple participants, but discipline their communication by exploiting distinctions between labeled messages in a single shared communication medium, rather than by introducing multiple or indexed more traditional session typed communication channels as, e.g., [12]. This approach allows us to unify the notions of global type and local type, and type highly dynamic scenarios of multiparty concurrent conversations not covered by other approaches. On the other hand, being more abstract and uniform, our type system does not explicitly keep track of participant identities. It would be interesting to investigate to what extent both approaches could be conciliated, for instance, by specializing our approach so as to consider extra constraints on projections on types and merges, restricting particular message exchanges to some roles.

References

1. Acciai, L., Boreale, M.: A Type System for Client Progress in a Service-Oriented Calculus. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 642–658. Springer, Heidelberg (2008)
2. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
3. Bruni, R., Mezzina, L.G.: Types and Deadlock Freedom in a Calculus of Services, Sessions and Pipelines. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 100–115. Springer, Heidelberg (2008)

4. Caires, L.: Spatial-Behavioral Types for Concurrency and Resource Control in Distributed Systems. Theoretical Computer Science 402(2-3), 120–141 (2008)

5. Caires, L., Vieira, H.T.: Conversation Types. UNL-DI-3-08, Departamento de Informática, Universidade Nova de Lisboa (2008)

6. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)

7. Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. In: 35th Symposium on Principles of Programming Languages, POPL 2008, pp. 261–272. ACM, New York (2008)

8. Chaki, S., Rajamani, S.K., Rehof, J.: Types as models: Model Checking Message-Passing Programs. In: POPL 2002, pp. 45–57. ACM, New York (2002)

9. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)

10. Honda, K.: Types for Dyadic Interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)

11. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)

12. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: 35th Symposium on Principles of Programming Languages, POPL 2008, pp. 273–284. ACM, New York (2008)

13. Igarashi, A., Kobayashi, N.: A Generic Type System for the Pi-Calculus. Theoretical Computer Science 311(1-3), 121–163 (2004)

14. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the Pi-Calculus. In: 23rd Symposium on Principles of Programming Languages, POPL 1996, pp. 358–371. ACM, New York (1996)

15. Lynch, N.: Fast Allocation of Nearby Resources in a Distributed System. In: 12th Symposium on Theory of Computing, STOC 1980, pp. 70–81. ACM, New York (1980)

16. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Part I + II. Information and Computation 100(1), 1–77 (1992)

17. Kobayashi, N.: A New Type System for Deadlock-Free Processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)

18. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)

19. Vieira, H.T., Caires, L., Seco, J.C.: The Conversation Calculus: A Model of Service-Oriented Computation. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008)

Appendix

In this appendix, we group the detailed definition of key technical notions, namely apartness, merge, and conformance. We denote by $Labels_{\mathcal{L}}(B)$ the set of message types with labels in $\mathcal{L}$ occurring in behavioral type B, and by $LLabels_{\mathcal{L}}(B)$ the set of directed labels (l^d) from $\mathcal{L}$ of a behavioral type B. For example, given some behavioral type B, $Labels_{\mathcal{L}_p}(B)$ is the set of all plain (from $\mathcal{L}_p$) message types ($p\,l^d(C)$) occurring in B.

Given behavioral types B_1 and B_2, we let $B_1 \asymp B_2$ state that message types with shared labels occur in both B_1 and B_2 with identical argument types.

Definition 6.1. *The conformance relation $B_1 \asymp B_2$ on behavioral types is defined as:*

$$B_1 \asymp B_2 \triangleq \text{if } (p_1 \ l^d(C_1)) \in \textit{Labels}_{\mathcal{L}_\star}(B_1) \text{ and } (p_2 \ l^d(C_2)) \in \textit{Labels}_{\mathcal{L}_\star}(B_2) \text{ then}$$
$$C_1 \equiv C_2 \text{ and either } p_1 = p_2 = ?, \ p_1 = p_2 = \boldsymbol{\tau} \text{ or } p_i = ! \text{ and } p_j = \boldsymbol{\tau}$$

Notice that two message types defined on shared labels and polarity ! are not conformant: this allows us to disallow composition of processes that are listening on the same shared message (expecting !), thus ensuring a unique handling principle.

Definition 6.2. *The apartness relation $B_1 \# B_2$ on behavioral types is defined as:*

$$B_1 \# B_2 \triangleq B_1 \asymp B_2 \text{ and } \textit{LLabels}_{\mathcal{L}_p}(B_1) \cap \textit{LLabels}_{\mathcal{L}_p}(B_2) = \emptyset$$

Definition 6.3. *The merge relation $B = B_1 \bowtie_u B_2$ on behavioral types is defined as:*

$$B\{? \ l^d(C)/\boldsymbol{\tau} \ l^d(C)\} \mid \star \ ! \ l^d(C) = B \bowtie_u \star \ ! \ l^d(C) \qquad\qquad \text{if } l \in \mathcal{L}_\star \ (1)$$

$$\Pi. \oplus_{i \in I} \{\boldsymbol{\tau} \ l_i^\downarrow(C_i).B_i\} = \Pi. \oplus_{i \in I} \{! \ l_i^\downarrow(C_i).B_i^+\} \bowtie_u \&_{i \in I} \{? \ l_i^\downarrow(C_i).B_i^-\} \ (2)$$
$$\text{if } l \in \mathcal{L}_p \text{ and } \Pi \# ?l_i^\downarrow(C_i).B_i^- \text{ and } B_i = B_i^- \bowtie_u B_i^+$$

$$\texttt{rec}\,\mathcal{X}.B = \texttt{rec}\,\mathcal{X}.B^+ \bowtie_u \texttt{rec}\,\mathcal{X}.B^- \qquad\qquad \text{if } B = B^- \bowtie_u B^+ \ (3)$$

$$B_1 \mid B_2 \ = B_1^+ \mid B_2^+ \bowtie_u B_1^- \mid B_2^- \quad \text{if } B_1 \# B_2 \text{ and } B_i = B_i^- \bowtie_u B_i^+ \ (4)$$

$$\mathcal{X} = \mathcal{X} \bowtie_u \mathcal{X} \ (5) \qquad\qquad B = B \bowtie_u \mathbf{0} \ (6) \qquad\quad B = \mathbf{0} \bowtie_u B \ (7)$$

We denote by $\bowtie$ the congruence closure extension of $\bowtie_u$ to both located and behavioral types. In (1) we denote by $B\{? \ l^d(C)/\boldsymbol{\tau} \ l^d(C)\}$ the type obtained by replacing all occurrences of $? \ l^d(C)$ by $\boldsymbol{\tau} \ l^d(C)$ in B. Shared labels synchronize and leave open the possibility for further synchronizations, expecting further outputs from the environment – rule (1). Instead, plain message synchronization captures the uniquely determined synchronization on that plain label – rule (2). Also, through (2), it is possible to hoist a sequence of messages Π, where Π abbreviates $M_1.(\ldots).M_k$, by interleaving with the continuation, if Π is apart from the behavior to be placed in parallel.