Analysis of Service Oriented Software Systems with the Conversation Calculus

Luís Caires and Hugo Torres Vieira

CITI and Departamento de Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa

Abstract. We overview some perspectives on the concept of service-based computing, and discuss the motivation of a small set of modeling abstractions for expressing and analyzing service based systems, which have led to the design of the Conversation Calculus. Distinguishing aspects of the Conversation Calculus are the adoption of a very simple, context sensitive, local message-passing communication mechanism, natural support for modeling multi-party conversations, and a novel mechanism for handling exceptional behavior. In this paper, written in a tutorial style, we review some Conversation Calculus based analysis techniques for reasoning about properties of service-based systems, mainly by going through a sequence of illustrating examples.

1 Introduction

Web and service-based systems have emerged mainly as a toolkit of technological and methodological solutions for building open-ended collaborative applications on the internet, leading to the recent trend towards the SaaS (Software as a Service) distribution model. Many concepts frequently advanced as particular to service-oriented computing systems, namely, interface-oriented distributed programming, long duration transactions and compensations, separation of workflow from service instances, late binding and discovery of functionalities, are not new. However, it must be acknowledged that the idea of service based computing is definitely contributing to physically realize an emerging model of computation, which is global (encompassing the internet as a whole), interaction-based (subsystems communicate via message passing), and loosely coupled (connections are established dynamically, and on demand). It is then very important to better understand in what sense service orientation may be exploited as a new paradigm to build and reason about distributed systems.

Of course, the global computing infrastructure is bound to remain highly heterogeneous and dynamic, so it does not seem reasonable to foresee the premature emergence of comprehensive theories and technological artifacts, well suited for everyone and every application. This suggests that one should focus not only on particular systems and theories themselves, but also on general systems, their properties, and their interfaces. In a recent line of work, mainly developed in the context of the EU IP project SENSORIA [21] and extended in the context of the

CMU-PT INTERFACES project [15], we have proposed a new model for service-oriented computation, based on a process calculus, with the aim of providing a foundation for rigorous modeling, analysis and verification. Our starting point was an attempt to isolate the essential characteristics of the service-oriented model of computation, in order to propose a motivation from "first principles" of a reduced set of general abstractions for expressing and analyzing service based systems. To focus on a set of independent primitives, we have developed our model by modularly adapting the synchronous π-calculus as follows

- introducing the general notion of *conversation context*;
- replacing channel communication by labeled message-passing primitives;
- adding a canonical exception handling mechanism

We have striven to keep our realization fairly general, so to achieve simplicity and clarity, and to ensure orthogonality and semantic independence of the chosen primitives. The proposed model, the Conversation Calculus, is a minimalistic, yet very expressive formalism, able to express and support reasoning about complex, dynamic, multiparty service based conversations, where partners may dynamically join and leave interactions, as we often find in "real-world" service based systems.

In this paper, we show how the Conversation Calculus can be used to reason about properties of service-based systems. In the spirit of a tutorial, no really new concepts are introduced, instead the focus is on a more detailed discussion of the underlying principles that guided the development of the language, accompanied by new examples that illustrate its expressiveness, and the kind of analyses that may be performed in the framework.

2 Aspects of Services

In this section, we attempt to identify some essential characteristics of the service-oriented model of computation, in order to justify a motivation from "first principles" of a reduced set of general abstractions for expressing and analyzing service based systems. Following [26], we identify as main features *distribution*, communication and *context* sensitiveness, and *loose coupling*.

2.1 Distribution

The purpose of a service relationship is to allow the incorporation of extra activities in a software system, without having to engage *local* resources and capabilities to support such activities. By delegating activities to an external provider, which will perform them using their remote resources and capabilities, a computing system may concentrate on those tasks for which it may autonomously provide adequate solutions. Thus, the notion of service makes particular sense when the service provider and the service client are separate entities, with access to separate resources and capabilities. This notion of the service relationship between provider and client assumes an underlying distributed computational

model, where client and server are located at least in distinct (operating system) processes, more frequently in distinct network sites.

The invocation of a service by a client results in the creation of a new service instance. Initially, a service instance is composed by a pair of endpoints, one endpoint located in the server site, where the service is defined, the other endpoint in the client site, where the request for instantiation took place. From the viewpoint of each partner, the respective endpoint acts as a local process, with potential direct access to local resources and capabilities. Thus, for us an endpoint is not a name, a port address, or channel, but an interactive process. Endpoints work together in a tightly coordinated way, by exchanging data and control information through a dedicated communication medium. In general, a service relationship may be initiated by a pair of remote endpoints, but may later on be enlarged with extra endpoints, located in other sites, developing a multiparty conversation.

2.2 Communication, Contexts, and Context Sensitiveness

The invocation of a service by an initiator client causes the creation of a new communication medium to host the interactions of the particular service instance. The service client (initiator) and client (responder) are immediately given access to this freshly created communication medium (see Figure 1), which will host a new conversation, and which might later on be joined in by other parties. To interact in the conversation medium the client establishes an endpoint or access point to the medium in its local context, and the process located in the endpoint is able to interact remotely in the service medium and locally in the client context. Likewise, the service responder establishes an endpoint to interact in the service conversation, and the process located in the endpoint is able to communicate with the server context (e.g., to access server resources).

For example, consider the scenario where the endpoint realizing an archiving functionality in the client context communicates with the other subsystems of the client, e.g., to receive document archiving requests and document retrieval requests, while the remote endpoint in the server site communicates with other subsystems in the service provider context, e.g., the database, the indexing infrastructure, and other resources needed for the provider task.

Access to a conversation may be shared with other parties. In particular, access may be given to other service providers that may then contribute to ongoing service tasks. At any moment, any party may then interact with any other party that shares access to the same conversation, as depicted in Figure 2 for the case of three parties. It is important to notice that in general, the distinction client/server may get blurred in a multiparty conversation, and we essentially need to host a delimited conversation between several symmetric partners.

For instance, in the archiving example, the server may decide to share its work load with other providers, allowing each one to establish an endpoint of the shared medium. In such way, a request to store data may be picked up by any of the service providers listening on the shared communication medium.
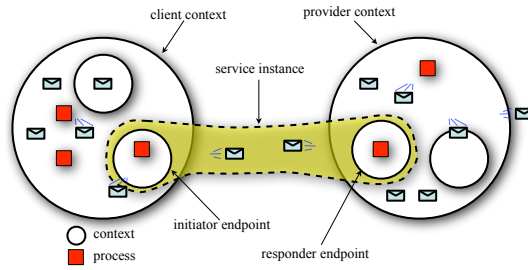
Fig. 1. Conversation initiation.
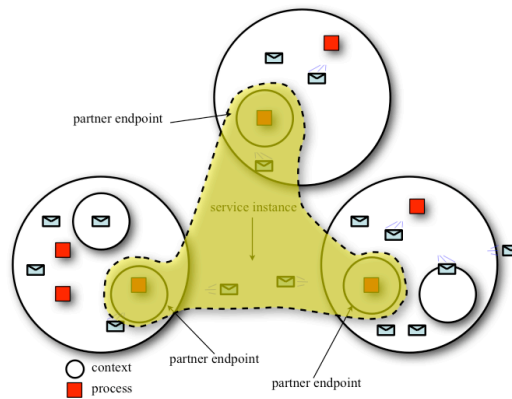
Fig. 2. Ongoing Conversation.

We understand an endpoint just as a particular case of a delimited context or scope of interaction. More generally, a context is a delimited space were computation and communication happens. A context may have a spatial meaning, e.g., as a *site* in a distributed system, but also a behavioral meaning, e.g., as *context of conversation*, or a *session*, between two or more partners. For example, the same message may appear in two different contexts, with different meanings (web services technology has introduced artifacts such as "correlation" to determine the appropriate context for otherwise indistinguishable messages).

Thus, the notion of conversation as a medium of communication which may be accessed from anywhere in the system and shared between several parties seems to be a convenient abstraction mechanism to structure the several service interactions in a service-oriented system.

The description above suggests two forms of communication capabilities. First, processes may interact if they are located in the same endpoint or in two endpoints of the same conversation. Second, interaction may occur between immediately nested endpoints. Endpoints as the one described may be nested at many levels, corresponding to subsidiary service instances, processes, etc. Notice
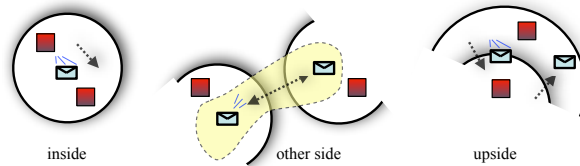
Fig. 3. Contexts and Communication Pathways.

that we do not expect communication to happen between arbitrary contexts, but rather to always fall in one of the two special cases described above: interaction inside a given conversation, and external interaction (with the immediately external context). In Figure 3 we illustrate our intended context dependent communication model, and the various forms of interaction it admits.

A context is also a natural abstraction to group and publish together closely related services, including when such services are provided by several sites. Typically, services published by the same entity are expected to share common resources; we notice that such sharing is common at several scales of granularity. Extreme examples are an object, where the service definitions are the methods and the shared context is the object internal state, and an entity such as, e.g., Amazon, that publishes several services for many different purposes; such services certainly share many internal resources of the Amazon context, such as databases, payment gateways, and so on.

Delimited contexts are also natural candidates for typing, in terms of the messages interchange patterns that may happen at its border. We would thus expect types (or logical formulas) specifying various properties of interfaces, of service contracts, of endpoint session protocols, of security policies, of resource usage, and of service level agreements, to be in general assigned to context boundaries. Enforcing boundaries between subsystems is also instrumental to achieve loose coupling of systems.

2.3 Loose Coupling

A service task may rely on several subsidiary services, where each one may involve a number of collaborating partners, and some local processes that control (orchestrate) the several subsidiary tasks and carry out some local functionality. Crucially to the service-oriented design, the several pieces that form the service implementation should be composed in a loosely coupled way, so as to support aimed features of service-oriented systems such as dynamic binding and dynamic discovery of partner service providers. For instance, an orchestration describing a "business process", should be specified in a quite independent way of the particular subsidiary service instances used. In the orchestration language `WSBPEL` [2], loose coupling to external services is enforced to some extent by the separate declaration of "partner links" and "partner roles" in processes. In the modeling language SRML [18] (inspired by the Service Component Architecture [4]),

the binding between service providers and clients is mediated by "wires", which describe plugging constraints between otherwise hard to match interfaces.

To support loose coupling, the specific details of a given service implementation should not be visible to external processes, so there must be a boundary between service instances and processes using them. Such boundary may be imposed by mediating processes that adapt the (implementation specific) service communication protocols to the abstract behavioral interface expected by the external context. It is then instrumental to encapsulate all computational entities cooperating in a service task in a conversation context, and allow them to communicate between themselves and the outer context only via some general message passing mechanism.

2.4   Other Aspects

There are many other aspects that must be addressed in a general model of service-oriented computation. The most obvious ones include failure handling and resource disposal, security (in particular access control, authentication and secrecy), time awareness, and a clean mechanism of inter-operation. This last aspect seems particularly relevant, and certainly suggests an important evaluation criteria for any service-oriented computation model.

3   The Conversation Calculus

In this section, we motivate and present in detail the primitives of our calculus. After that, we present the syntax of our calculus, and formally define its operational semantics, by means of a labeled transition system.

Conversation Context   A conversation context is a medium where related interactions can take place. A conversation context can be distributed in many pieces, and processes inside any piece can seamlessly talk to any other piece of the same context. Each context has a unique name (cf., a URI). We use the conversation access construct

$$n \blacktriangleleft [P]$$

to say that the process P is interacting in conversation n. Potentially, each conversation access will be placed at a different enclosing context. On the other hand, any such conversation access will necessarily be placed at a single enclosing context. The relationship between the enclosing context and such an access point may be seen as a caller/callee relationship, but where both entities may interact continuously.

Context Awareness   A process running inside a given context should be able to dynamically become aware of the identity of the former. This capability may be realized by the construct

$$\texttt{this}(x).P$$

The variable x will be replaced inside the process P by the name n of the current context. The computation will proceed as $P\{x\leftarrow n\}$. For instance the process $c \blacktriangleleft [\texttt{this}(x).P]$ evolves in one computation step to $c \blacktriangleleft [P\{x\leftarrow c\}]$. Our context awareness primitive bears some similarity with the $\texttt{self}$ or $\texttt{this}$ of object-oriented languages, although of course it has a different semantics.

3.1 Communication

Communication between subsystems is realized by message passing. We denote the input/output of messages from/to the current context by the constructs

$$\texttt{label}^\downarrow?(x_1,\ldots,x_n).P$$
$$\texttt{label}^\downarrow!(v_1,\ldots,v_n).P$$

Messages are $\texttt{label}$ed. In the output case (!), the terms v_i represent message arguments, that is, values to be sent, as expected. In the input case (?), the variables x_i represent the message parameters and are bound in P, as expected. The target symbol $\downarrow$ (read "here") says that the corresponding communication actions must interact in the current conversation conversation context, where the messages are being sent. Second, we denote the input/output of messages from/to the outer context by the constructs

$$\texttt{label}^\uparrow?(x_1,\ldots,x_n).P$$
$$\texttt{label}^\uparrow!(v_1,\ldots,v_n).P$$

The target symbol $\uparrow$ (read "up") says that the corresponding communication actions must interact in the (uniquely determined) outer context, where "outer" is understood relatively to the context where the process exercising the action is running.

3.2 Service Oriented Idioms

Although we do not introduce them natively in the language, we present some service-oriented idioms which capture typical service-oriented interaction: service definition, service instantiation and service join.

A context (a.k.a. a site) may publish one or more service definitions. Service definitions are stateless entities, pretty much as function definitions in a functional programming language. A service definition may be expressed by

$$\texttt{def } ServiceName \Rightarrow ServiceBody$$

where $ServiceName$ is the service name, and $ServiceBody$ is the process that should be executed at the service endpoint for each service instance, in other words the service body. In order to be published, such a definition must be inserted into a context, e.g.,

$$ServiceProvider \blacktriangleleft [\texttt{def } ServiceName \Rightarrow ServiceBody \,\cdots\,]$$

Such a published service may be instantiated by means of a instantiation idiom

$$\texttt{new}\ n \cdot ServiceName \Leftarrow ClientProtocol$$

where n identifies the conversation where the service is published. For instance, the service defined above may be instantiated by

$$\texttt{new}\ ServiceProvider \cdot ServiceName \Leftarrow ClientProtocol$$

The *ClientProtocol* describes the process that will run inside the endpoint held by the client. The outcome of a service instantiation is the creation of a new globally fresh context identity (a hidden name), and the creation of two access pieces of a context named by this fresh identity. One will contain the *ServiceBody* process and will be located inside the *ServiceProvider* context. The other will contain the *ClientProtocol* process and will be located in the same context as the `instance` expression that requested the service instantiation.

In our model conversation identifiers may be manipulated by processes if needed (accessed via the $\texttt{this}(x).P$), passed around in messages and subject to scope extrusion: this allows us to model collaboration between multiple parties in a single service conversation, realized by the progressive access of dynamically determined partners to an ongoing conversation. Joining of another partner to an ongoing conversation is a frequent programming idiom, that may be abstracted by:

$$\texttt{join}\ ServiceProvider \cdot \texttt{ServiceName} \Leftarrow ContinuationProcess$$

The `join` and the `new` expression are implemented in a similar way, both relying on name passing. The key difference is that while `new` creates a fresh *new conversation*, `join` allows a service `ServiceName` defined at *ServiceProvider* to join in the *current conversation*, while the calling party continues interacting in the current conversation as specified by *ContinuationProcess*. So, even if the `new` and `join` are represented in a similar way, the abstract notion they realize is actually very different: `new` is used to start a fresh conversation between two parties (e.g., used by a client that instantiates a service) while the `join` is used to allow another service provider to join an ongoing conversation (e.g., used by a participant in a service collaboration to dynamically delegate a task to some remote partner). At a very high level of description the two primitives can be unified as primitives that support the dynamic delegation of tasks (either in a unary conversation or in a n-ary conversation).

3.3 Exception Handling

We introduce two primitives to model exceptional behavior, in particular fault signaling, fault detection, and resource disposal, these aspects are certainly orthogonal to the previously introduced communication mechanisms. We adapt the classical $\texttt{try} - \texttt{catch}-$ and $\texttt{throw}-$ to a concurrent setting. The primitive to raise an exception is

$$\texttt{throw}.Exception$$

This construct throws an exception with continuation the process *Exception*, and has the effect of forcing the termination of all other processes running in all enclosing contexts, up to the point where a $\mathtt{try}-\mathtt{catch}$ block is found (if any). The continuation *Exception* will be activated when (and if) the exception is caught by such an exception handler. The exception handler construct

$$\mathtt{try}\ P\ \mathtt{catch}\ Handler$$

allows a process P to execute normally until some exception is thrown inside P. At that moment, all of P is terminated, and the $Handler$ handler process, which is guarded by $\mathtt{try}-\mathtt{catch}$, is activated, concurrently with the continuation *Exception* of the $\mathtt{throw}.Exception$ that originated the exception, in the context of a given $\mathtt{try}-\mathtt{catch}-$ block. By exploiting the interaction of the $Handler$ and *Exception* processes, it is possible to represent many recovery and resource disposal protocols, including compensable transactions [12].

3.4 Syntax and Semantics of the Calculus

We may now formally introduce the syntax and semantics of the conversation calculus. We assume given an infinite set of names Λ, an infinite set of variables $\mathcal{V}$, and an infinite set of labels $\mathcal{L}$. We abbreviate $a_1, \ldots, a_k$ by $\widetilde{a}$. We use d for communication directions, α for action prefixes and P, Q for processes. Notice that message and service identifiers (from $\mathcal{L}$) are plain labels, not subject to restriction or binding. The syntax of the calculus is given in Figure 4.

The static core of our language is derived from the π-calculus [24]. We thus have $\mathbf{0}$ for the inactive process, $P \mid Q$ for the parallel composition, $(\boldsymbol{\nu}a)P$ for name restriction, and $\mathtt{rec}\,\mathcal{X}.P$ and $\mathcal{X}$ for recursion. The prefix guarded choice $\Sigma_{i \in I}\,\alpha_i.P_i$ specifies a process which may exhibit any one of the α_i actions and evolve to the respective continuation P_i. Processes also specify conversation accesses: $n \blacktriangleleft [P]$ represents a process that is accessing conversation n and interacting in it according to what P specifies.

Context-oriented actions prefixes include the output $\mathtt{l}^d!(\widetilde{n})$ — send names $\widetilde{n}$ in a message labeled $\mathtt{l}$, to either the current or enclosing conversation (depending on d); the input $\mathtt{l}^d?(\widetilde{x})$ — receive names and instantiate variables $\widetilde{x}$ in a message labeled $\mathtt{l}$ from either the current or enclosing conversation (d); and the context awareness primitive $\mathtt{this}(x)$ which allows a process to dynamically gain access to the identity of its "current" ($\downarrow$) conversation.

The Conversation Calculus includes two primitives for exception handling: the $\mathtt{try}\ P\ \mathtt{catch}\ Q$ and the $\mathtt{throw}.P$. The $\mathtt{throw}.P$ signals an exception and causes the termination of every process up to an enclosing $\mathtt{try}\ P\ \mathtt{catch}\ Q$, in which case P is activated. The $\mathtt{try}\ P\ \mathtt{catch}\ Q$ behaves as process P up to the point an exception is thrown (if any), in which case process Q is activated.

The distinguished occurrences of a, $\widetilde{x}$, and x are binding occurrences in $(\boldsymbol{\nu}a)P$, $\mathtt{l}^d?(\widetilde{x}).P$, and $\mathtt{this}(x).P$, respectively. The sets of free ($fn(P)$) and bound ($bn(P)$) names and variables in a process P are defined as usual, and we implicitly identify α-equivalent processes.

$$
\begin{array}{lll}
a, b, c, \ldots & \in \quad \Lambda & \text{(Names)} \\
x, y, z, \ldots & \in \quad \mathcal{V} & \text{(Variables)} \\
n, v, \ldots & \in \quad \Lambda \cup \mathcal{V} & \text{(Identifiers)} \\
\mathtt{l}, \mathtt{s} \ldots & \in \quad \mathcal{L} & \text{(Labels)}
\end{array}
$$

$$
d \quad ::= \downarrow \mid \uparrow \quad \text{(Directions)}
$$

$$
\begin{array}{llll}
\alpha & ::= & \mathtt{l}^d!(\widetilde{n}) & \text{(Output)} \\
& \mid & \mathtt{l}^d?(\widetilde{x}) & \text{(Input)} \\
& \mid & \mathtt{this}(x) & \text{(Context awareness)}
\end{array}
$$

$$
\begin{array}{llll}
P, Q & ::= & \mathbf{0} & \text{(Inaction)} \\
& \mid & P \mid Q & \text{(Parallel)} \\
& \mid & (\boldsymbol{\nu} a)P & \text{(Restriction)} \\
& \mid & \mathtt{rec}\, \mathcal{X}.P & \text{(Recursion)} \\
& \mid & \mathcal{X} & \text{(Variable)} \\
& \mid & \Sigma_{i \in I}\, \alpha_i.P_i & \text{(Prefix Guarded Choice)} \\
& \mid & n \blacktriangleleft [P] & \text{(Conversation Access)} \\
& \mid & \mathtt{try}\, P \,\mathtt{catch}\, Q & \text{(Try-catch)} \\
& \mid & \mathtt{throw}.P & \text{(Throw)}
\end{array}
$$

Fig. 4. The Conversation Calculus

We define the semantics of the conversation calculus via a labeled transition system. We introduce transition labels λ and actions act:

$$
\begin{array}{ll}
act ::= \tau \mid \mathtt{l}^d!(\widetilde{a}) \mid \mathtt{l}^d?(\widetilde{a}) \mid \mathtt{this}^d \mid \mathtt{throw} & \text{(Actions)} \\
\lambda \quad ::= c\ act \mid act \mid (\nu a)\lambda & \text{(Transitions)}
\end{array}
$$

Actions capture internal actions τ, message outputs $\mathtt{l}^d!(\widetilde{a})$ and inputs $\mathtt{l}^d?(\widetilde{a})$, context identity accesses $\mathtt{this}^d$ and exception signals $\mathtt{throw}$. Transition labels tag actions with the conversation identifier they respect to $c\ act$ and with bound names which are emitted in the action $(\nu a)\lambda$. In $(\nu a)\lambda$ the distinguished occurrence of a is bound with scope λ (cf., the π-calculus bound output and bound input actions). A transition label containing $c\ act$ is said to be *located at c* (or just *located*), otherwise is said to be *unlocated*. We write $(\nu\widetilde{a})$ to abbreviate a (possibly empty) sequence $(\nu a_1) \ldots (\nu a_k)$, where the order is not important.

We adopt a few conventions and notations. We note by λ^d a transition label λ^d containing the direction d ($\uparrow, \downarrow$). Then we denote by $\lambda^{d'}$ the label obtained by replacing d by d' in λ^d. Given an unlocated label λ, we represent by $c \cdot \lambda$ the label obtained by locating λ at c, so that e.g., $c \cdot (\nu\widetilde{a})act = (\nu\widetilde{a})c\ act$. We assert $unloc(\lambda)$ if λ is not located and is either an input or an output, and $loc(\lambda)$ if λ is located and is either an input or an output. We define $out(\lambda)$ as follows:

$$
out((\nu\widetilde{a})b\ \mathtt{l}^d!(\widetilde{c})) \triangleq \widetilde{c} \setminus (\widetilde{a} \cup \{b\}) \quad out((\nu\widetilde{a})\mathtt{l}^d!(\widetilde{c})) \triangleq \widetilde{c} \setminus (\widetilde{a})
$$

and $out(\lambda) = \emptyset$ otherwise. We use $n(\lambda)$ and $bn(\lambda)$ to denote (respectively) all names and the bound names of a transition label.

The labeled transition system relies on a synchronization algebra which we now introduce. Essentially, the synchronization algebra describes how two parallel processes may synchronize, specifying how any pair of transitions may be combined and what is the result of such combination. Since not all pairs of transitions represent a valid synchronization we use the $\circ$ as the result of an invalid synchronization. We then denote by $\lambda_1 \bullet \lambda_2$ the result of combining λ_1 and λ_2 via function $\bullet$, resulting in either a transition (in case λ_1 and λ_2 may synchronize) or $\circ$ (otherwise). $\bullet$ is defined such that $\lambda_1 \bullet \lambda_2 = \lambda_2 \bullet \lambda_1$ and:

$$
\begin{aligned}
\mathtt{l}^\downarrow!(\widetilde{a}) \bullet \mathtt{l}^\downarrow?(\widetilde{a}) &\triangleq \tau \\
\mathtt{l}^\uparrow!(\widetilde{a}) \bullet \mathtt{l}^\uparrow?(\widetilde{a}) &\triangleq \tau \\
c\,\mathtt{l}^\downarrow!(\widetilde{a}) \bullet c\,\mathtt{l}^\downarrow?(\widetilde{a}) &\triangleq \tau \\
\mathtt{l}^\downarrow!(\widetilde{a}) \bullet \mathtt{l}^\uparrow?(\widetilde{a}) &\triangleq \mathtt{this}^\downarrow \\
\mathtt{l}^\downarrow!(\widetilde{a}) \bullet c\,\mathtt{l}^\downarrow?(\widetilde{a}) &\triangleq c\,\mathtt{this}^\downarrow \\
\mathtt{l}^\uparrow!(\widetilde{a}) \bullet c\,\mathtt{l}^\downarrow?(\widetilde{a}) &\triangleq c\,\mathtt{this}^\uparrow \\
\mathtt{l}^\downarrow?(\widetilde{a}) \bullet \mathtt{l}^\uparrow!(\widetilde{a}) &\triangleq \mathtt{this}^\downarrow \\
\mathtt{l}^\downarrow?(\widetilde{a}) \bullet c\,\mathtt{l}^\downarrow!(\widetilde{a}) &\triangleq c\,\mathtt{this}^\downarrow \\
\mathtt{l}^\uparrow?(\widetilde{a}) \bullet c\,\mathtt{l}^\downarrow!(\widetilde{a}) &\triangleq c\,\mathtt{this}^\uparrow
\end{aligned}
$$

for some $\mathtt{l}, \widetilde{a}, c$, and $\lambda_1 \bullet \lambda_2 = \circ$ otherwise. Function $\bullet$ resolves direct synchronizations (e.g., $c\,\mathtt{l}^\downarrow?(\widetilde{a}) \bullet c\,\mathtt{l}^\downarrow!(\widetilde{a})$) in an internal action τ. However, since messages are context dependent, synchronizations of unlocated transitions require contextual information. So, for instance, transition labels $\mathtt{l}^\downarrow!(\widetilde{a})$ and $c\,\mathtt{l}^\downarrow?(\widetilde{a})$ may synchronize, provided the current conversation is c — the resulting label $c\,\mathtt{this}^\downarrow$ will read the identity of the current conversation, and progress only if the identity is c. Intuitively, label $c\,\mathtt{this}^\downarrow$ captures a silent action of a process which may occur *provided* the process is placed in conversation c. Labels $\mathtt{l}^\downarrow?(\widetilde{a})$ and $\mathtt{l}^\uparrow!(\widetilde{a})$ synchronize, provided the current and enclosing conversations have the same identity — tested via label $\mathtt{this}^\downarrow$. Also, labels $c\,\mathtt{l}^\downarrow?(\widetilde{a})$ and $\mathtt{l}^\uparrow!(\widetilde{a})$ synchronize provided the enclosing conversation has identity c — checked via label $c\,\mathtt{this}^\uparrow$.

We may now present the labeled transition system. In Figs. 5, 6 and 7 we present the labeled transition system for the calculus. The rules presented in Figure 5 closely follow the π-calculus labeled transition system (see [25]). We omit the rules symmetric to (*Par*) and (*Close*).

We briefly review the rules presented in Fig. 6: in rule (*Here*), after going through a context boundary, an $\uparrow$ message becomes $\downarrow$; in (*Loc*) an unlocated $\downarrow$ message gets located at the context identity in which it originates; in (*Through*) a non-$\mathtt{this}$ located label transparently crosses the context boundary, and likewise in (*Internal*) for a τ label; in (*ThisHere*) a $\mathtt{this}$ label reads the identity of the enclosing context (and matches it with the current conversation identity); in (*ThisLoc*) a $c\,\mathtt{this}$ label matches the enclosing context; in (*This*) a $\mathtt{this}$ label reads the current conversation identity.

As for the rules in Figure 7: in (*Throw*) an exception is signaled; in (*ThrowPar*) and (*ThrowConv*) enclosing computations are terminated; in (*Try*) a non-$\mathtt{throw}$

$$\mathbf{1}^d!(\widetilde{a}).P \xrightarrow{\mathbf{1}^d!(\widetilde{a})} P \quad (Out) \qquad\qquad \mathbf{1}^d?(\widetilde{x}).P \xrightarrow{\mathbf{1}^d?(\widetilde{a})} P\{\widetilde{x}{\leftarrow}\widetilde{a}\} \quad (In)$$

$$\frac{\alpha_j.P_j \xrightarrow{\lambda} Q \quad (j \in I)}{\Sigma_{i \in I}\, \alpha_i.P_i \xrightarrow{\lambda} Q}(Sum) \qquad\qquad \frac{P\{\mathcal{X}{\leftarrow}\mathtt{rec}\ \mathcal{X}.P\} \xrightarrow{\lambda} Q}{\mathtt{rec}\ \mathcal{X}.P \xrightarrow{\lambda} Q}(Rec)$$

$$\frac{P \xrightarrow{\lambda} Q \quad (a \notin n(\lambda))}{(\boldsymbol{\nu}a)P \xrightarrow{\lambda} (\boldsymbol{\nu}a)Q}(Res) \qquad\qquad \frac{P \xrightarrow{\lambda} Q \quad (a \in out(\lambda))}{(\boldsymbol{\nu}a)P \xrightarrow{(\boldsymbol{\nu}a)\lambda} Q}(Open)$$

$$\frac{P \xrightarrow{\lambda} Q \quad (\lambda \neq \mathtt{throw})}{P \mid R \xrightarrow{\lambda} Q \mid R}(Par) \quad \frac{P \xrightarrow{\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad (\lambda_1 \bullet \lambda_2 \neq \circ)}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} P' \mid Q'}(Com)$$

$$\frac{P \xrightarrow{(\nu\widetilde{a})\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad (\widetilde{a} \cap fn(P \mid Q) = \emptyset, \lambda_1 \bullet \lambda_2 \neq \circ)}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\widetilde{a})(P' \mid Q')}(Close)$$

Fig. 5. Transition Rules for Basic Operators.

$$\frac{P \xrightarrow{\lambda^\uparrow} Q \quad (c \notin bn(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda^\downarrow} c \blacktriangleleft [Q]}(Here) \qquad\qquad \frac{P \xrightarrow{\lambda} Q \quad (unloc(\lambda))}{c \blacktriangleleft [P] \xrightarrow{c \cdot \lambda} c \blacktriangleleft [Q]}(Loc)$$

$$\frac{P \xrightarrow{\lambda} Q \quad (loc(\lambda), c \notin bn(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda} c \blacktriangleleft [Q]}(Through) \quad \frac{P \xrightarrow{\tau} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]}(Internal)$$

$$\frac{P \xrightarrow{\mathtt{this}^\downarrow} Q}{c \blacktriangleleft [P] \xrightarrow{c\,\mathtt{this}^\downarrow} c \blacktriangleleft [Q]}(ThisHere) \qquad \frac{P \xrightarrow{c\,\mathtt{this}^\downarrow} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]}(ThisLoc)$$

$$\mathtt{this}(x).P \xrightarrow{c\,\mathtt{this}^\downarrow} P\{x{\leftarrow}c\} \quad (This)$$

Fig. 6. Transition Rules for Conversation Operators.

$$\mathtt{throw}.P \xrightarrow{\mathtt{throw}} P \quad (Throw) \qquad\qquad \frac{P \xrightarrow{\mathtt{throw}} R}{P \mid Q \xrightarrow{\mathtt{throw}} R}(ThrowPar)$$

$$\frac{P \xrightarrow{\mathtt{throw}} R}{n \blacktriangleleft [P] \xrightarrow{\mathtt{throw}} R}(ThrowConv) \quad \frac{P \xrightarrow{\mathtt{throw}} R}{\mathtt{try}\ P\ \mathtt{catch}\ Q \xrightarrow{\tau} Q \mid R}(Catch)$$

$$\frac{P \xrightarrow{\lambda} Q \quad \lambda \neq \mathtt{throw}}{\mathtt{try}\ P\ \mathtt{catch}\ R \xrightarrow{\lambda} \mathtt{try}\ Q\ \mathtt{catch}\ R}(Try)$$

Fig. 7. Transition Rules for Exception Handling Operators.

$$\begin{aligned}
\texttt{def } s \Rightarrow P \quad &\triangleq \mathsf{s}^{\downarrow}?(x).x \blacktriangleleft [P] \\
\texttt{new } n \cdot s \Leftarrow Q \quad &\triangleq (\boldsymbol{\nu}c)(n \blacktriangleleft [\mathsf{s}^{\downarrow}!(c)] \mid c \blacktriangleleft [Q]) \\
\texttt{join } n \cdot s \Leftarrow Q \quad &\triangleq \texttt{this}(x).(n \blacktriangleleft [\mathsf{s}^{\downarrow}!(x)] \mid Q) \\
\star\texttt{def } s \Rightarrow P \quad &\triangleq \texttt{rec } \mathcal{X}.\mathsf{s}^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P])
\end{aligned}$$

Fig. 8. Service Idioms.

transition crosses the handler block; in (*Catch*) an exception is caught by the handler block, activating (in parallel) the continuation process and the handler.

Notice that the presentation of the transition system is fully modular: the rules for each operator are independent, so that one may easily consider several fragments of the calculus (e.g., without exception handling primitives). The operational semantics of closed systems, usually represented by a reduction relation, is here specified by $\xrightarrow{\tau}$.

3.5 Representing Service-Oriented Idioms

Our core model focuses on the fundamental notions of conversation context and message-based communication. From these basic mechanisms, useful programming abstractions for service-oriented systems may be idiomatically defined, namely service definition and instantiation constructs (defined as primitives in [26]), and the conversation join construct (introduced in [13]). These constructs may be embedded in a simple way in the minimal calculus, without hindering the flexibility of modeling and analysis.

The service-oriented idioms along with their translation in lower level communication primitives is shown in Fig. 8. We specify the service definition idiom by $\texttt{def } s \Rightarrow P$, which publishes a service named s in the current conversation. Process P specifies the code that will run in the service conversation, upon service instantiation, implementing the service provider role in the conversation. The service definition is implemented in basic communication primitives as:

$$\texttt{def } s \Rightarrow P \quad \triangleq \quad \mathsf{s}^{\downarrow}?(x).x \blacktriangleleft [P] \qquad (x \notin fv(P))$$

Essentially, the service definition specifies a message — labeled by the name of the service s — is received, carrying the identity of the service conversation. Then, code P will run in such received conversation. Service definitions must be placed in appropriate conversation contexts (cf., methods in objects). For instance, to specify `BuyService` is published in the *Seller* context we write:

$$Seller \blacktriangleleft [\texttt{def BuyService} \Rightarrow SellerCode]$$

Typically, services once published are persistent in the sense they can be instantiated several times. To model such persistent services we introduce the recursive variant of service definition.

$$\star\texttt{def } s \Rightarrow P \quad \triangleq \quad \texttt{rec } \mathcal{X}.\mathsf{s}^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P]) \qquad (x \notin fv(P))$$

Persistent service definitions are specified so as to always be ready to receive a service instantiation request, handling each request in the conversation received in each service instantiation message.

The idiom that supports the instantiation of a published service is noted by `new` $n \cdot s \Leftarrow Q$. The `new` idiom specifies the conversation where the service is published at (n), the name of the service (s) and the code that will run on the service client side (Q). A service instantiation resulting from a synchronization from a published service `def` and an instantiation `new` results in the creation of a fresh conversation that is shared between service provider and service caller. We translate the `new` idiom in the basic primitives of the CC as follows:

$$\texttt{new } n \cdot s \Leftarrow Q \quad \triangleq \quad (\boldsymbol{\nu}c)(n \blacktriangleleft \left[\texttt{s}^{\downarrow}!(c)\right] \mid c \blacktriangleleft [Q]) \qquad (c \notin (fn(Q) \cup \{n\}))$$

The service instantiation is then realized by means of a message exchange in conversation n, where the service is published at, being the message labeled by the name of the service s and carrying a newly created name c that identifies the conversation where the service interaction is to take place. In parallel to the message output that instantiates the service, we find the code of the client role Q, running in the freshly created conversation c. Notice that Q is already active, although it has to wait for the server side to pick up the service conversation identity to start interacting in conversation c, by means of $\downarrow$ directed messages. Notice also that process Q can interact in the conversation where the service instantiation request lies, using $\uparrow$ directed messages.

The join idiom is implemented using the core CC primitives as follows:

$$\texttt{join } n \cdot s \Leftarrow Q \quad \triangleq \quad \texttt{this}(x).(n \blacktriangleleft \left[\texttt{s}^{\downarrow}!(x)\right] \mid Q) \qquad (x \notin (fv(Q) \cup \{n\}))$$

The current conversation identity is accessed via the `this` primitive, and passed along in service message s exchanged in the conversation n where s is published at. Process Q continues to interact in the current conversation (the same that was accessed in the `this`).

4 A Sequence of Examples

In this section, we illustrate the expressiveness of our calculus through a sequence of examples. For the sake of commodity, we informally extend the language with some auxiliary primitives, e.g., $\texttt{if} - \texttt{then} - \texttt{else}$, etc. We also use replication !, which may be simulated using recursion, anonymous contexts, defined as $[P] \triangleq (\boldsymbol{\nu}a)(a \blacktriangleleft [P])$ (where a is fresh) to isolate communication, and we omit $\downarrow$ message directions (e.g., $\texttt{read}?()$ abbreviates $\texttt{read}^{\downarrow}?()$).

4.1 Memory Cell

We discuss some simple examples of stateful service definition and invocation patterns, using memory cell implementations. Consider the following implemen-

tation of a memory cell service.

$$
\begin{aligned}
&\texttt{def Cell} \Rightarrow (\\
&\quad \texttt{!(read?().next!()} \\
&\quad\ + \\
&\quad\ \ \texttt{write?}(x).\texttt{stop!}().\texttt{rec}\,\mathcal{X}.(\texttt{stop?}() + \texttt{next?}().\texttt{value!}(x).\mathcal{X}) \\
&\quad\ \ | \\
&\quad\ \texttt{stop?()}))
\end{aligned}
$$

Intuitively, each time a value is written in the cell, a process that stores the value is spawned. This process is ready to repeatedly emit the $\texttt{value}$ upon request (message $\texttt{next}$), or to $\texttt{stop}$ giving out the value. To read the cell value, a request for the $\texttt{next}$ emission of the value is sent to the memory process. To write a new value, the installed memory process is $\texttt{stop}$ped, and another process which stores the new value is spawned (notice that since the first write does not have to stop any memory process, the respective stop message is collected separately).

We show how to instantiate the $\texttt{Cell}$ service so to create a delegate cell process in the current context. The delegate accepts $\texttt{put}$ and $\texttt{get}$ messages from the client context, and replies to each $\texttt{get}$ message with a $\texttt{reply}$ message to the context. It provides the memory cell functionality delegation to the remote service $\textit{FreeCellsInc} \blacktriangleleft \texttt{Cell}$.

$$
\begin{aligned}
&\texttt{new}\ \textit{FreeCellsInc} \cdot \texttt{Cell} \Leftarrow (\\
&\quad \texttt{!(put}^\uparrow?(x).\texttt{write!}(x) \\
&\quad\ + \\
&\quad\ \ \texttt{get}^\uparrow?().\texttt{read!}().\texttt{value?}(x).\texttt{reply}^\uparrow!(x))\ )
\end{aligned}
$$

A process in the context may then use the created service instance as follows:

$$
\texttt{put!}(value).\texttt{get!}().\texttt{reply?}(x).\texttt{proceed!}(x)
$$

To show how the system evolves, let us consider the composition of the $\texttt{Cell}$ service provider and user, placing the provider in the $\textit{FreeCellsInc}$ context:

$$
\begin{aligned}
&\textit{FreeCellsInc} \blacktriangleleft [\\
&\quad \texttt{def Cell} \Rightarrow (\ \texttt{!(read?().next!()} \\
&\qquad\qquad\quad\ + \\
&\qquad\qquad\quad\ \ \texttt{write?}(x).\texttt{stop!}().\texttt{rec}\,\mathcal{X}.(\texttt{stop?}() + \texttt{next?}().\texttt{value!}(x).\mathcal{X}) \\
&\qquad\qquad\quad\ \ | \\
&\qquad\qquad\quad\ \texttt{stop?()}\ ] \\
&\ | \\
&\texttt{new}\ \textit{FreeCellsInc} \cdot \texttt{Cell} \Leftarrow (\texttt{!(put}^\uparrow?(x).\texttt{write!}(x) \\
&\qquad\qquad\qquad\qquad\qquad\ + \\
&\qquad\qquad\qquad\qquad\qquad\ \ \texttt{get}^\uparrow?().\texttt{read!}().\texttt{value?}(x).\texttt{reply}^\uparrow!(x))) \\
&\ | \\
&\texttt{put!}(value).\texttt{get!}().\texttt{reply?}(x).\texttt{proceed!}(x)
\end{aligned}
$$

By translating the service idioms into their lower level representation we obtain:

$FreeCellsInc \blacktriangleleft [$
     $\texttt{Cell?}(y).y \blacktriangleleft [\, !(\texttt{read?}().\texttt{next!}()$
                          $+$
                          $\texttt{write?}(x).\texttt{stop!}().\texttt{rec}\, \mathcal{X}.(\texttt{stop?}() + \texttt{next?}().\texttt{value!}(x).\mathcal{X})$
                          $|$
                          $\texttt{stop?}()\,]\,]$
$|$
$(\boldsymbol{\nu}c)(FreeCellsInc \blacktriangleleft [\texttt{Cell!}(c)]$
     $|$
     $c \blacktriangleleft [\, !(\texttt{put}^\uparrow\texttt{?}(x).\texttt{write!}(x)$
           $+$
           $\texttt{get}^\uparrow\texttt{?}().\texttt{read!}().\texttt{value?}(x).\texttt{reply}^\uparrow\texttt{!}(x))\,]\,)$
 $|$
$\texttt{put!}(value).\texttt{get!}().\texttt{reply?}(x).\texttt{proceed!}(x)$

Service provider and client may synchronize in the Cell service message, being
(fresh) name c passed in the message which allows the service provider to gain
access to the conversation.

   $(\boldsymbol{\nu}c)(FreeCellsInc \blacktriangleleft [$
       $c \blacktriangleleft [\, !(\texttt{read?}().\texttt{next!}()$
             $+$
             $\texttt{write?}(x).\texttt{stop!}().\texttt{rec}\, \mathcal{X}.(\texttt{stop?}() + \texttt{next?}().\texttt{value!}(x).\mathcal{X})$
             $|$
             $\texttt{stop?}()\,]\,]$
    $|$
   $c \blacktriangleleft [\, !(\texttt{put}^\uparrow\texttt{?}(x).\texttt{write!}(x)$
         $+$
         $\texttt{get}^\uparrow\texttt{?}().\texttt{read!}().\texttt{value?}(x).\texttt{reply}^\uparrow\texttt{!}(x))\,]\,)$
    $|$
   $\texttt{put!}(value).\texttt{get!}().\texttt{reply?}(x).\texttt{proceed!}(x)$

The service client instance and the process using it interact in message put
(notice the ↑ direction), activating a cell write.

   $(\boldsymbol{\nu}c)(FreeCellsInc \blacktriangleleft [$
       $c \blacktriangleleft [\, !(\texttt{read?}().\texttt{next!}()$
             $+$
             $\texttt{write?}(x).\texttt{stop!}().\texttt{rec}\, \mathcal{X}.(\texttt{stop?}() + \texttt{next?}().\texttt{value!}(x).\mathcal{X})$
             $|$
             $\texttt{stop?}()\,]\,]$
    $|$
   $c \blacktriangleleft [\, !(\texttt{put}^\uparrow\texttt{?}(x).\texttt{write!}(x) + \texttt{get}^\uparrow\texttt{?}().\texttt{read!}().\texttt{value?}(x).\texttt{reply}^\uparrow\texttt{!}(x))$
         $|$
         $\texttt{write!}(value)\,]\,)$
    $|$
   $\texttt{get!}().\texttt{reply?}(x).\texttt{proceed!}(x)$

At this point, service provider and client instances exchange message write in the service conversation c, after which message stop is exchanged.

$(\boldsymbol{\nu}c)(\mathit{FreeCellsInc} \blacktriangleleft [$
$\quad c \blacktriangleleft [\, !(\texttt{read?().next!()}$
$\qquad\quad +$
$\qquad\quad \texttt{write?}(x).\texttt{stop!().rec}\,\mathcal{X}.(\texttt{stop?}() + \texttt{next?().value!}(x).\mathcal{X})$
$\qquad\quad |$
$\qquad\quad \texttt{rec}\,\mathcal{X}.(\texttt{stop?}() + \texttt{next?().value!}(\mathit{value}).\mathcal{X})\,]\,]$
$\quad |$
$c \blacktriangleleft [\, !(\texttt{put}^{\uparrow}\texttt{?}(x).\texttt{write!}(x)$
$\qquad\quad +$
$\qquad\quad \texttt{get}^{\uparrow}\texttt{?().read!().value?}(x).\texttt{reply}^{\uparrow}\texttt{!}(x))\,]\,)$
$\quad |$
$\texttt{get!().reply?}(x).\texttt{proceed!}(x)$

Then, the get message is exchanged between service client instance and its user process, activating a cell read.

$(\boldsymbol{\nu}c)(\mathit{FreeCellsInc} \blacktriangleleft [$
$\quad c \blacktriangleleft [\, !(\texttt{read?().next!()}$
$\qquad\quad +$
$\qquad\quad \texttt{write?}(x).\texttt{stop!().rec}\,\mathcal{X}.(\texttt{stop?}() + \texttt{next?().value!}(x).\mathcal{X})$
$\qquad\quad |$
$\qquad\quad \texttt{rec}\,\mathcal{X}.(\texttt{stop?}() + \texttt{next?().value!}(\mathit{value}).\mathcal{X})\,]\,]$
$\quad |$
$c \blacktriangleleft [!(\texttt{put}^{\uparrow}\texttt{?}(x).\texttt{write!}(x) + \texttt{get}^{\uparrow}\texttt{?().read!().value?}(x).\texttt{reply}^{\uparrow}\texttt{!}(x))$
$\qquad\quad |$
$\qquad\quad \texttt{read!().value?}(x).\texttt{reply}^{\uparrow}\texttt{!}(x))\,]\,)$
$\quad |$
$\texttt{reply?}(x).\texttt{proceed!}(x)$

At this point, service provider and client exchange message read, after which message next is exchanged and the value emission is activated.

$(\boldsymbol{\nu}c)(\mathit{FreeCellsInc} \blacktriangleleft [$
$\quad c \blacktriangleleft [\, !(\texttt{read?().next!()}$
$\qquad\quad +$
$\qquad\quad \texttt{write?}(x).\texttt{stop!().rec}\,\mathcal{X}.(\texttt{stop?}() + \texttt{next?().value!}(x).\mathcal{X})$
$\qquad\quad |$
$\qquad\quad \texttt{value!}(\mathit{value}).\texttt{rec}\,\mathcal{X}.(\texttt{stop?}() + \texttt{next?().value!}(\mathit{value}).\mathcal{X})\,]\,]$
$\quad |$
$c \blacktriangleleft [!(\texttt{put}^{\uparrow}\texttt{?}(x).\texttt{write!}(x)$
$\qquad\quad +$
$\qquad\quad \texttt{get}^{\uparrow}\texttt{?().read!().value?}(x).\texttt{reply}^{\uparrow}\texttt{!}(x))$
$\qquad\quad |$
$\qquad\quad \texttt{value?}(x).\texttt{reply}^{\uparrow}\texttt{!}(x))\,]\,)$
$\quad |$
$\texttt{reply?}(x).\texttt{proceed!}(x)$

Now, the `value` message is exchanged, after which message `reply` carrying the initially written value is picked up by the user process allowing it to `proceed`.

4.2 Dictionary

In the next example, we use a toy dictionary service to discuss the possible need of correlating messages belonging to different interaction contexts. A possible instantiation of such a service may be expressed thus:

$$
\begin{aligned}
&\texttt{new } \mathit{FreeBagsCo} \cdot \texttt{Dict} \Leftarrow (\\
&\quad !(\texttt{put}^{\uparrow}?(\mathit{key}, x).\texttt{store}!(\mathit{key}, x) \\
&\quad + \\
&\quad \texttt{get}^{\uparrow}?(\mathit{key}).\texttt{get}!(\mathit{key}).\texttt{value}?(x).\texttt{reply}^{\uparrow}!(x) \\
&)
\end{aligned}
$$

If the generated instance is to be solicited by several concurrent `get` requests, some form of correlation may be needed, in order to route the associated `reply` answers to the appropriate contexts. In this case, we set the `get` message to play the role of an initiator message, now receiving also a reference r to the context of interaction (associated to getting the dictionary entry associated to the indicated key).

$$
\begin{aligned}
&\texttt{new } \mathit{FreeBagsCo} \cdot \texttt{Dict} \Leftarrow (\\
&\quad !(\texttt{put}^{\uparrow}?(\mathit{key}, x).\texttt{store}!(\mathit{key}, x) \\
&\quad + \\
&\quad \texttt{get}^{\uparrow}?(r, \mathit{key}).\texttt{get}!(\mathit{key}).\texttt{value}?(x).r \blacktriangleleft [\texttt{reply}!(x)] \\
&)
\end{aligned}
$$

Now, the `reply` message is sent inside the appropriate conversation context r, the one that relates to the initial `get`. A process in the context may then use the service instance by following the appropriate intended protocol, e.g.:

$$
\texttt{put}!(\mathit{key}, \mathit{value}).(\boldsymbol{\nu} r)(\texttt{get}(r, \mathit{key}).r \blacktriangleleft [\texttt{reply}?(x).\texttt{proceed}^{\uparrow}!(x)])
$$

Here, we are essentially in presence of a familiar form of continuation passing.

In this case, we have generated a new special context r in order to carry out the appropriate conversation. In many situations we would like just to correlate the subsidiary conversation with the current context, without having to introduce a new special context. In this case, we may write the (perhaps more natural) code, that will have the same effect as the code above:

$$
\texttt{put}!(\mathit{key}, \mathit{value}).\texttt{this}(\mathit{currentC}).\texttt{get}(\mathit{currentC}, \mathit{key}).\texttt{reply}?(x).\texttt{proceed}!(x)
$$

Remember that the $\texttt{this}(x).P$ (context-awareness) primitive binds x in P to the identity of the current context.

4.3 Service Provider Factory

We revisit the memory cell example, and provide a different realization. In this case, we would like to represent each cell as a specific service provider, such that the `Read` and `Write` operations are now services, rather than operations of a particular service as shown above. A cell (named n) may be be represented by the context:

$$Cell(n) \triangleq n \blacktriangleleft [\, \texttt{def Read} \Rightarrow \texttt{value}^\uparrow?(x).\texttt{value!}(x) \mid$$
$$\texttt{def Write} \Rightarrow \texttt{value}?(x).\texttt{value}^\uparrow!(x) \,]$$

We may now specify a memory cell factory service.

$$CellFactoryService \triangleq \texttt{def NewCell} \Rightarrow (\boldsymbol{\nu}a)(Cell(a) \mid \texttt{replyCell!}(a))$$

To instantiate the cell factory service, and drop a `theCell` message with a fresh cell reference (c) in the current context, we may write:

$$\texttt{new } FreeCellsInc \cdot \texttt{NewCell} \Leftarrow \texttt{replyCell}?(x).\texttt{theCell}^\uparrow!(x)$$

The newly allocated cell service provider is allocated in the *FreeCellsInc* context, as expected. To use the cell one may then write, e.g.,

$$\texttt{theCell}(c).($$
$$\cdots$$
$$\texttt{new } c \cdot Read \Leftarrow \cdots$$
$$\mid \cdots$$
$$\texttt{new } c \cdot Write \Leftarrow \cdots$$
$$\cdots)$$

This usage pattern for services, where service instantiation corresponds to some form of task delegation rather that process delegation, is closer to a distributed object model than to a service-oriented model. In any case, it is interesting to be able to accommodate this usage pattern as a special case, not only for the sake of abstract generality, but also because it will certainly turn out useful in appropriate scenarios.

4.4 Exceptions

We illustrate a few usage idioms for our exception handling primitives. In the first example, the service `Service` is instantiated on site *Server*, and repeatedly re-launched on each failure – failure will be signaled by exception throwing within the local protocol *ClientProto*, possibly as a result of a remote message.

$$\texttt{rec } Restart.$$
$$\texttt{try}$$
$$\texttt{new } Server \cdot \texttt{Service} \Leftarrow ClientProto$$
$$\texttt{catch } Restart$$

A possible scenario of remote exception throwing is illustrated below.

$$Server \blacktriangleleft [$$
$$\text{def Interruptible} \Rightarrow$$
$$\text{stop?().urgentStop!().throw} \mid$$
$$\dots ServiceProto \dots]$$

$$\text{new } Server \cdot \text{Interruptible} \Leftarrow$$
$$\text{urgentStop?().throw} \mid$$
$$\dots ClientProto \dots$$

Here, any remote endpoint instance of the `Interruptible` service may be interrupted by the service protocol *ServiceProto* by dropping a message `stop` inside the endpoint context. In this example, such a message causes the endpoint to send an `urgentStop` message to the client side, and then throwing an exception, which will cause abortion of the service endpoint. On the other hand, the service invocation protocol will throw an exception at the client endpoint upon reception of `urgentStop`. Notice that this behavior will happen concurrently with ongoing interactions between *ServiceProto* and *ClientProto*. In this example, the exception issued in the server and client endpoints will have to be managed by appropriate handlers in both sites. In the next example, no exception will be propagated to the service site, but only to the remote client endpoint, and as a result of any exception thrown in the service protocol *ServiceProto*.

$$Server \blacktriangleleft [$$
$$\text{def Interruptible} \Rightarrow$$
$$\text{try}$$
$$ServiceProto$$
$$\text{catch urgentStop!().throw}]$$

In the examples discussed above, the decision to terminate the ongoing remote interactions is triggered by the service code. In the next example, we show a simple variation of the idioms above, where the decision to kill the ongoing service instance is responsibility of the service context. Here, any instance of the `Interruptible` service may be terminated by the service provider by means of dropping a message `killRequest` in the endpoint external context.

$$Server \blacktriangleleft [$$
$$\text{def Interruptible} \Rightarrow$$
$$\text{try}$$
$$\text{killRequest}^\uparrow\text{?().throw} \mid ServiceProto$$
$$\text{catch urgentStop().throw}]$$

A simple example of a similar pattern in our last example on exceptions.

$$Server \blacktriangleleft [$$
$$\text{def TimeBound} \Rightarrow$$
$$\text{timeAllowed}^\uparrow\text{?}(delay)\text{.wait}(delay)\text{.throw} \mid$$
$$ServiceProto]$$

Here, any invocation of the `TimeBound` service will be allocated no more than *delay* time units before being interrupted, where *delay* is a dynamic parameter value read from the current server side context (we assume some extension of our sample language with a `wait`(t) primitive, with the expected semantics).

4.5 Programming a Finance Portal

In this section we show the implementation of a Finance portal (inspired in a SENSORIA Project [21] case study) which illustrates how the several primitives and idioms of the language can be combined, allowing to model complex interaction patterns in a rather simple way. We model a credit request scenario, where a bank client, a bank clerk and a bank manager participate, mediated through a bank portal. The client starts by invoking a service available in the bank portal and places the credit request, providing his identification and the desired amount. The implementation of such client in CC is then:

$$Client \blacktriangleleft [\ ClientTerminal$$
$$|$$
$$\texttt{new}\ BankPortal \cdot \texttt{CreditRequest} \Leftarrow$$
$$\texttt{request!}(myId, amount).$$
$$(\texttt{requestApproved?()}.\texttt{transferDate!}(date).\texttt{approved}^{\uparrow}\texttt{!()}$$
$$+$$
$$\texttt{requestDenied?()}.\texttt{denied}^{\uparrow}\texttt{!())}\ ]$$

The client code for the service instantiation specifies the messages that are to be exchanged in the service conversation by using ↓ messages. First a message `request` is sent, after which one of two messages (either `requestApproved` or `requestDenied`) informing on the decision is received. Only after receiving one of such messages is the *ClientTerminal* informed (correspondingly) of the final decision. Notice that the service code interacts with the *ClientTerminal* process by means of ↑ messages `approved` or `denied`. In fact, from the point of view of *ClientTerminal* the external interface of the service instance can be characterized by the process `approved!() + denied!()`.

Next we show the code of the `CreditRequest` service published in conversation *BankPortal*, and persistently available (as indicated by the $\star$ annotation).

$$BankPortal \blacktriangleleft [\star\ \texttt{def}\ \texttt{CreditRequest} \Rightarrow$$
$$\texttt{request?}(uid, amount).$$
$$\texttt{join}\ Clerk \cdot \texttt{RiskAssessment} \Leftarrow$$
$$\texttt{assessRisk!}(uid, amount).$$
$$\texttt{riskVal?}(risk).$$
$$\texttt{if}\ risk\ =\ \texttt{HIGH}\ \texttt{then}\ \texttt{requestDenied!}()$$
$$\texttt{else}\ \texttt{this}(clientC).$$
$$\texttt{new}\ Manager \cdot \texttt{CreditApproval} \Leftarrow$$
$$\texttt{requestApproval!}(clientC, uid, amount, risk)\ ]$$

The server code specifies that, in each `CreditRequest` service conversation, a message `request` is received, then message `assessRisk` is sent and then message

riskVal is received. The first will be exchanged with the service client, while the latter two will be exchanged with the clerk, that is asked to join the ongoing conversation through service **RiskAssessment**. After that, depending on the risk rate the clerk determined for the request, the bank portal is either able to automatically reject the request, in which case it informs the client of such decision by sending message **requestDenied**, or it has to consult the bank manager, creating a new instance of the **CreditApproval** service to that end — notice that a **new** instance is created in this case. However, since the bank manager will reply directly back to the client, the name of the client service conversation is accessed, via the **this**($clientC$), and passed along to the manager (in the first argument of message **requestApproval**). This pattern is similar to a **join**: the name of the current conversation is sent to the remote service provider, allowing for it to join in the conversation. The difference with respect to a join is that the remote service will only join the client conversation to reply back to the client. In some sense, it is as if we only delegate a basic fragment of the client conversation (e.g., the final reply), instead of incorporating the whole functionality provided by **CreditApproval** in the **CreditRequest** service collaboration.

We now show the code for the **CreditApproval** service, assuming there is a *ManagerTerminal* process able to interact with the manager, similarly to the *ClientTerminal* process.

$$
\begin{aligned}
&Manager \blacktriangleleft [\ ManagerTerminal \\
&\quad | \\
&\quad \star\, \texttt{def CreditApproval} \Rightarrow \\
&\qquad \texttt{requestApproval}?(clientC, uid, amount, risk). \\
&\qquad \texttt{this}(managerC). \\
&\qquad \texttt{showRequest}^{\uparrow}!(managerC, uid, amount, risk). \\
&\qquad\quad (\texttt{reject}?().clientC \blacktriangleleft [\ \texttt{requestDenied}!()\ ] \\
&\qquad\quad + \\
&\qquad\quad \texttt{accept}?().clientC \blacktriangleleft [\ \texttt{requestApproved}!(). \\
&\qquad\qquad\qquad\qquad\qquad \texttt{join}\ BankATM \cdot \texttt{CreditTransfer} \Leftarrow \\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{orderTransfer}!(uid, amount)\ ]\ ]
\end{aligned}
$$

The **CreditApproval** server code specifies the reception of a **requestApproval** message, carrying the name of the conversation where the final answer is to be given in, after which the identity of the current conversation is accessed, and passed along to *ManagerTerminal* in message **showRequest** in conversation *Manager*. This allows *ManagerTerminal* to reply directly to the "right" conversation, since several copies of the **CreditApproval** service may be running in parallel, and therefore several **showRequest** messages may have to be concurrently handled and replied to by the **ManagerTerminal**: if the replies were to be placed in the *Manager* conversation then they would also compete and be at risk of being picked up by the wrong (unrelated) service instance. The *ManagerTerminal* thus replies in the **CreditApproval** service conversation with either a **reject** message or an **accept** message. After that the credit request client is notified accordingly in the respective conversation. Also, in the case that the credit is approved, the manager asks service **CreditTransfer** published at

BankATM to join the client conversation (the current conversation for the `join` is the client conversation), so as to place the transfer order.

We now specify the code for the *CreditTransfer* service.

$$
\begin{aligned}
&BankATM \blacktriangleleft [\\
&\quad BankATMProcess \\
&\quad | \\
&\quad \star \, \texttt{def CreditTransfer} \Rightarrow \\
&\qquad \texttt{orderTransfer}?(uid, amount). \\
&\qquad \texttt{transferDate}?(date). \\
&\qquad \texttt{scheduleTransfer}^{\uparrow}!(uid, amount, date) \ ]
\end{aligned}
$$

The `CreditTransfer` service code specifies the reception of the transfer order and of the desired date of the transfer, after which forwards the information to a local *BankATMProcess*, which will then schedule the necessary procedure. Notice that the *BankATM* party is only asked to join in the conversation under some circumstances, in such case interacting with the bank manager in message `orderTransfer` and with the credit request client in message `transferDate`, while otherwise it does not participate at all in the service collaboration.

The system obtained by composing the described processes captures an interesting scenario where, not only the set of multiple participants in the collaboration is dynamically determined, but also the actual maximum number of participants depends on a runtime condition.

5 Analysis Techniques

In several works, we have studied the dynamic and static semantics of the CC, and illustrated their use to the analysis of service-based systems. Namely, we have investigated behavioral equivalences, and type systems for conversation fidelity and deadlock absence. In this tutorial note, we focus on basic results of the observational semantics; in Section 6, we give further pointers to the type based analysis.

We define a compositional behavioral semantics of the conversation calculus by means of the standard notion of strong bisimulation. We prove that strong and weak bisimilarity are congruences for all the primitives of our calculus. This further ensures that our syntactically defined constructions induce properly defined behavioral operators at the semantic level.

Definition 1. *A (strong) bisimulation is a symmetric binary relation $\mathcal{R}$ on processes such that, for all processes P and Q, if $P\mathcal{R}Q$, we have:*

If $P \xrightarrow{\lambda} P'$ and $bn(\lambda) \cap fn(P \mid Q) = \emptyset$ then there is a process Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\mathcal{R}Q'$.

We denote by $\sim$ (strong bisimilarity) the largest strong bisimulation.

Strong bisimilarity is an equivalence relation. We also have:

Theorem 1. *Strong bisimilarity is a congruence.*

We consider weak bisimilarity defined as usual, denoted by $\approx$.

Theorem 2. *Weak bisimilarity is a congruence.*

Notice Theorem 2 is not a direct consequence of Theorem 1. In fact, there are other languages where the latter holds while the former does not. Informally, the usual counter-example is given by processes $\tau.\alpha.P$ and $\alpha.P$ which are weakly bisimilar. Put in a summation context with process R we obtain $\tau.\alpha.P + R$ and $\alpha.P + R$ which are not weakly bisimilar (the former can do a silent action and lose the ability to do R and the latter cannot mimic such action).

We also prove other interesting behavioral equations, for instance, the following equations hold up to strong bisimilarity:

1. $n \blacktriangleleft [P] \mid n \blacktriangleleft [Q] \sim n \blacktriangleleft [P \mid Q]$.
2. $m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P]]] \sim n \blacktriangleleft [o \blacktriangleleft [P]]$.

(1) captures the local character of message-based communication in our model, while (2) illustrates the idea that processes located in different access pieces of the same conversation interact as if they where located in the same access piece. Using such behavioral identities, in [26] we show that processes admit a flat representation, where the nesting level of any active communication prefix is at most two.

As an example of the sort of specifications captured by our type analysis consider the `CreditApproval` service shown in Section 4.5. After being notified by of the credit approval decision, the `CreditApproval` service instance forwards the notification to the client and, in case of approval, asks service `CreditTransfer` to join the ongoing conversation. The type that captures the role of the `CreditApproval` service instance in the client conversation is characterized by the following type (assuming some basic types $\mathtt{T}_1, \ldots$):

$$
\begin{aligned}
managerR \; \triangleq \; \\
\oplus \{ \, &! \, \mathtt{requestApproved}().\boldsymbol{\tau} \, \mathtt{orderTransfer}(\mathtt{T}_1, \mathtt{T}_2).? \, \mathtt{transferDate}(\mathtt{T}_4); \\
&! \, \mathtt{requestDenied}() \}
\end{aligned}
$$

Type $managerR$ specifies a choice ($\oplus$) between two outputs (!): either the output of message `requestApproved` or of message `requestDenied`. In the case of the `requestApproved` choice, the type specifies that the process proceeds by internally exchanging ($\boldsymbol{\tau}$) message `orderTransfer` and by receiving message `transferDate`. This behavior results from the combination of the rest of the manager role in the client conversation (the output of message `orderTransfer`, typed $! \, \mathtt{orderTransfer}(\mathtt{T}_1, \mathtt{T}_2)$) with the type of the `CreditTransfer` service:

$$
creditTransferB \; \triangleq \; ? \, \mathtt{orderTransfer}(\mathtt{T}_1, \mathtt{T}_2).? \, \mathtt{transferDate}(\mathtt{T}_4)
$$

where the synchronization in message `orderTransfer` between the manager and the bank is recorded in $managerR$, using the $\boldsymbol{\tau}$ annotation. Such flexibility in

combining behavioral types is crucial to capture conversations between several parties, including scenarios where parties dynamically join and leave conversations, which is the case of the `CreditTransfer` service.

The `CreditApproval` service is then characterized by the type:

$$creditApprovalB \triangleq$$
$$? \texttt{requestApproval}(managerR, \texttt{T}_1, \texttt{T}_2, \texttt{T}_3). \oplus \{\boldsymbol{\tau}\,\texttt{reject}();\ \boldsymbol{\tau}\,\texttt{accept}()\}$$

which specifies the reception of a `requestApproval` message, carrying a conversation identifier in which the manager behaves as specified by $managerR$, after which proceeding as the internal choice between messages `reject` and `accept`.

The type of the manager process exposes the required and provided services:

$$ManagerProcess ::$$
$$Manager : [\star?\,\texttt{CreditApproval}(creditApprovalB)$$
$$|$$
$$BankATM : [\star!\,\texttt{CreditTransfer}(creditTransferB)]$$

service `CreditApproval` is published (?) in conversation $Manager$, and service `CreditTransfer` is expected (!) in conversation $BankATM$.

6 Further Reading and Closing Remarks

The Conversation Calculus was first introduced in [26], where we also presented a basic study of its behavioral semantics. In [13, 14] we introduced the conversation type theory, which provides analysis techniques for conversation fidelity and deadlock absence, while addressing challenging scenarios involving dynamically established conversations between several partners. Analysis techniques based on the CC where mainly developed by Hugo Vieira in his PhD thesis [27]. Several aspects of the Conversation Calculus have also been reported in several chapters of the book which collects the key results of IP SENSORIA Project [1, 3, 11, 17, 22]. More recently, in the context of the CMU-PT INTERFACES project [15], we have been using the Conversation Calculus/Types suite to model and analyze role based multiparty interactions in the setting of web business applications.

Our development of the concept of conversation was initially motivated by the concept of binary session [19]; most session-based approaches to service interactions only support binary interactions (simple client-server). Only recently proposals have appeared to support multiparty interaction [5, 6, 16, 20, 28]. To support multiparty interaction, [20] considers multiple session channels, while [5] considers indexed session channels, both resorting to multiple communication pathways. Our model follows an essentially different design, by letting a single medium of interaction support concurrent multiparty interaction via labeled messages. We base our approach on the notion of conversation context, and on plain message-passing communication, which allows us to introduce the conversation initiation and conversation join constructs as idioms. This contrasts with other session-based proposals for session and service-oriented models where

we find primitive service instantiation operations constructs (see, e.g., [7, 8, 23]). Comparing with such models, the Conversation Calculus seems to be the simplest extension to the pure π-calculus for modeling and analyzing complex multi-party service based systems.

Ad hoc primitives to deal with exceptional behavior are present in several service calculi. Perhaps surprisingly, our exception mechanism, although clearly based on the canonical construct for functional languages, does not seem to have been explored before us in process calculi (more recently, a similar mechanism is explored in [9]). In [12] we showed how a transactional model supporting compensations (the compensating CSP [10]) can be encoded in the Conversation Calculus by means of its exception mechanism.

In our discussion on the underlying principles of the service-oriented computational model, we left out some interesting features of distributed systems that we view as fairly alien to this setting. Forms of code migration (weak mobility) seem to require an homogeneous execution support infrastructure, and thus to run against the aim to get loose coupling, openness and independent evolution of subsystems. In general, any mechanism relying on centralized control or authority mechanisms, or that require a substantial degree on homogeneity in the runtime infrastructure (e.g., strong mobility) also seem hard to accommodate.

To summarize, we have reviewed the Conversation Calculus, a core model for service-oriented computation. The design of the Conversation Calculus, building on the identification and analysis of general aspects of service-based systems, was also discussed and justified in considerable detail. By means of a series of simple, yet hopefully illuminating examples, we have illustrated how our model may express many service-oriented idioms, and complex multi-party service based systems in a very natural way. Properties such as behavioral equivalence, conversation fidelity and deadlock absence may be verified on Conversation Calculus models by means of several available techniques. The aim of providing simpler, more expressive, and usable techniques for complex software systems is certainly a good justification for introducing yet another core model or language, in particular if such model is expressed as a tiny layer on top of a purest foundational model; the π calculus. We leave for others to judge the extent to which such aim was achieved by the Conversation Calculus and related techniques.

References

1. L. Acciai, C. Bodei, M. Boreale, R. Bruni, and H. Vieira. *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *Lecture Notes in Computer Science*, chapter Static Analysis Techniques for Session-Oriented Calculi. Springer-Verlag, 2011.
2. A. Alves et al. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2006.

3. M. Bartoletti, L. Caires, I. Lanese, F. Mazzanti, D. Sangiorgi, H. Vieira, and R. Zunino. *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *Lecture Notes in Computer Science*, chapter Tools and Verification. Springer-Verlag, 2011.

4. M. Beisiegel et al. Service Component Architecture: Building Systems using a Service-Oriented Architecture, version 0.9. Technical report, BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase Joint Whitepaper, 2005.

5. L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In F. van Breugel and M. Chechik, editors, *CONCUR 2008, 19th International Conference on Concurrency Theory, Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer-Verlag, 2008.

6. 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 *Lecture Notes in Computer Science*, pages 240–256. Springer-Verlag, 2008.

7. M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: a Service Centered Calculus. In *Proceedings of WS-FM 2006, 3rd International Workshop on Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer-Verlag, 2006.

8. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In G. Barthe and F. de Boer, editors, *FMOODS 2008, 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, Proceedings*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer-Verlag, 2008.

9. M. Bravetti and G. Zavattaro. On the Expressive Power of Process Interruption and Compensation. *Mathematical Structures in Computer Science*, 19(3):565–599, 2009.

10. M. Butler and C. Ferreira. A Process Compensation Language. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *IFM 2000, Second International Conference on Integrated Formal Methods, Proceedings*, volume 1945 of *Lecture Notes in Computer Science*, pages 61–76. Springer-Verlag, 2000.

11. L. Caires, R. De Nicola, R. Pugliese, V. Vasconcelos, and G. Zavattaro. *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *Lecture Notes in Computer Science*, chapter Core Calculi for Service-Oriented Computing. Springer-Verlag, 2011.

12. L. Caires, C. Ferreira, and H. Vieira. A Process Calculus Analysis of Compensations. In C. Kaklamanis and F. Nielson, editors, *TGC 2008, Fourth International Symposium on Trustworthy Global Computing, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*, pages 87–103. Springer-Verlag, 2009.

13. L. Caires and H. Vieira. Conversation Types. In G. Castagna, editor, *ESOP 2009, 18th European Symposium on Programming, Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 285–300. Springer-Verlag, 2009.

14. L. Caires and H. Vieira. Conversation Types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.

15. CMU-PT INTERFACES Project. Website: http://ctp.di.fct.unl.pt/interfaces/.

16. 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.

17. C. Ferreira, I. Lanese, A. Ravara, H. Vieira, and G. Zavattaro. *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *Lecture Notes in Computer Science*, chapter Advanced Mechanisms for Service Combination and Transactions. Springer-Verlag, 2011.

18. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A Formal Approach to Service Component Architecture. In M. Bravetti, M. N., and G. Zavattaro, editors, *Web Services and Formal Methods, Third International Workshop,WS-FM 2006*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer-Verlag, 2006.

19. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In C. Hankin, editor, *ESOP'98, 7th European Symposium on Programming,ETAPS'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

20. 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.

21. IP Sensoria Project. Website: http://www.sensoria-ist.eu/.

22. I. Lanese, A. Ravara, and H. Vieira. *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *Lecture Notes in Computer Science*, chapter Behavioral Theory for Session-Oriented Calculi. Springer-Verlag, 2011.

23. I. Lanese, V. T. Vasconcelos, F. Martins, and A. Ravara. Disciplining Orchestration and Conversation in Service-Oriented Computing. In *5th International Conference on Software Engineering and Formal Methods*, pages 305–314. IEEE Computer Society Press, 2007.

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

25. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

26. 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 *Lecture Notes in Computer Science*, pages 269–283. Springer-Verlag, 2008.

27. H. T. Vieira. *A Calculus for Modeling and Analyzing Conversations in Service-Oriented Computing*. PhD thesis, Universidade Nova de Lisboa, 2010.

28. 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 *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.