The Conversation Calculus: a Model of Service-Oriented Computation

Hugo T. Vieira, Luís Caires, and João C. Seco

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

We present a process-calculus model for expressing and analyzing service-based systems. Our approach addresses central features of the service-oriented computational model such as distribution, process delegation, communication and context sensitiveness, and loose coupling. Distinguishing aspects of our model are the notion of conversation context, the adoption of a context sensitive, message-passing-based communication, and of a simple yet expressive mechanism for handling exceptional behavior. We instantiate our model by extending a fragment of the π-calculus, illustrate its expressiveness by means of many examples, and study its basic behavioral theory; in particular, we establish that bisimilarity is a congruence.

1 Introduction

Web services have emerged mainly as a toolkit of technological and methodological solutions for building open-ended collaborative software systems on the Internet. Many concepts that are frequently put forward as distinctive of service-oriented computing, namely, object-oriented distributed programming, long duration transactions and compensations, separation of workflow from service instances, late binding and discovery of functionalities, are certainly not new, at least when considered in isolation. What is certainly new about services is that they are contributing to physically realize (on the Internet) a global, interaction-based, loosely-coupled, model of computation. We would like to better understand in what sense service orientation is to be seen as a new paradigm to build and reason about distributed systems.

The main contributions of this work are the development of a process calculus for service-oriented computing based on a novel notion of conversation context, and the study of its basic behavioral theory. In particular, we establish that bisimilarity is a congruence, thus asserting the proper status of the proposed constructions as operators at the level of the behavioral semantics; we believe that such a result has not yet been provided for other related service calculi. Our starting point is an attempt to isolate and clarify essential characteristics of the service-oriented model, in order to propose a motivation from "first principles" of a reduced set of general abstractions for expressing and analyzing service-based systems. We then instantiate our model by modularly extending the static fragment of the π-calculus with conversation contexts, message-passing communication primitives, and an exception handling mechanism.

1.1 Some Key Aspects of Service-Oriented Computing

We identify as key aspects of the service-oriented computational model: *distribution*, process *delegation*, communication and *context* sensitiveness, and *loose coupling*.

Distribution The purpose of a service relationship is to allow the incorporation of certain activities in a given system, without having to engage *local* resources and capabilities to support or implement such activities. By delegating activities to an external service provider, which will perform them using its own *remote* resources and capabilities, a computing system may concentrate on those tasks for which it may autonomously provide convenient solutions. Thus, the notion of service makes particular sense when the service provider and the service client are separate entities, with access to separate sets of resources and capabilities. This understanding 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 sites of a network.

Process Delegation versus Operation Invocation The primitive remote communication mechanism in distributed computing is message passing. On top of this basic mechanism, the only one really implementable, more sophisticated abstractions may be represented, namely remote procedure call (passing first-order data) and remote method invocation (also passing remote object references). Along these lines, we see service invocation as a still higher level mechanism, allowing the service client to delegate to a remote server not just a single operation or task, but the execution of a whole interactive activity (technically, a process). This emphasis on the remote delegation of *interactive processes* is, in our view, a distinguishing feature of service-oriented computing, as opposed to the remote delegation of individual operations.

Invocation of a service by a client results in the creation of a new service instance. 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, we do not consider an endpoint to be a name, a port address, or channel, but an interactive process. Dual endpoints work together in a tightly coordinated way, by exchanging data and control information through a private communication tunnel.

Contexts and Context Sensitiveness A context is a space where 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 a *context of conversation* between two or more parties. In the latter situation, remote parties may well talk under the same context of conversation, so that contexts of conversation need not be localized, but accessible at different points. Moreover, 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 context of conversation seems to be a convenient abstraction mechanism to structure the interactions between several entities collaborating in a service-oriented system.

A context is also a natural abstraction to publish together closely related services. 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: a "small" object, where the service definitions are the methods and the shared context is the object internal state, and an ISP such as, e.g., Amazon, that pub-

lishes many services for many different purposes; such services certainly share internal resources in the Amazon context, such as databases, payment gateways, and so on.

Loose Coupling A service-based computation usually consists in an collection of remote partner service instances, in which functionality is to be delegated, some locally implemented processes, and one or more control (or orchestration) processes. The flexibility and openness of a service-based design, or at least an aimed feature, results from a loose coupling between these various components. For instance, an orchestration describing a "business process", should be specified in a quite independent way of the particular subsidiary service instances used, paving the way for dynamic binding and dynamic discovery of service providers. 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 [11], the binding between service providers and clients is mediated by "wires", which describe plugging constraints between otherwise hard to match interfaces. These are two instances of the same general principle.

To avoid tight coupling of services, the interface between a service instance (at each of its several endpoints) and the context of instantiation should be mediated by appropriate connecting processes, in order to hide and/or adapt the endpoint communication protocol (which is in some sense dependent of the particular implementation or service provider chosen) to the abstract behavioral interface expected by the context of instantiation. All computational entities cooperating in a service task should then be encapsulated (delimited inside a conversation context), and able to communicate between themselves and the outer context only via some general message passing mechanism.

Communication Computations interacting in a context may offer essentially three forms of communication capabilities. First, they may communicate within the context, corresponding to regular internal computations in the context. Second, an endpoint must be able to send messages to and receive messages from the other (dual) endpoint of the context, reflecting interactions between the client and the server roles of a service instance. Third, internally to a context it must be possible to send messages to and receive messages from the enclosing context, thus allowing for a context to be seen as a regular process by its peers at the upper level. Contexts as the one described may be nested at many levels, corresponding to subsidiary service instances, processes, etc.

In the next Section, we present the conversation calculus, a process model crafted to incorporate the several key aspects just discussed; we explain the various primitives of the calculus, and define its syntax and operational semantics. In Section 3 we further motivate our model and calculus by means of several examples. In Section 4 we define the behavioral semantics and present related technical results. We compare our approach with related work in Section 5 and conclude in Section 6.

2 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.

Context A key contribution of this paper is the notion of 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), and is partitioned in two endpoints, which we will refer by "initiator" (◀), or "responder" (▶). We use the endpoint access construct $n \blacktriangleleft [P]$ to say that the process P is placed at the initiator endpoint of context n, and the (dual) construct $n \blacktriangleright [P]$ to say that the process P is placed at the responder endpoint of context n. Potentially, each endpoint access will be placed at a different enclosing context. On the other hand, any such endpoint access will necessarily be placed at a single enclosing context. The relationship between the enclosing context and such an endpoint may be seen as a call/callee relationship, but where both entities may interact continuously.

Communication Communication between subsystems is realized by means of message passing. Internal computation is related to communications between subsystems inside a given context. First, we denote the output and the input of messages to/from the current context by the constructs $\texttt{out} \downarrow label(\widetilde{v}).P$ and $\texttt{in} \downarrow label(\widetilde{x}).P$. In the output case, the terms v_i represent message arguments, values to be sent, as expected. In the input case, the variables x_i represent message parameters and are bound in P, as expected. The direction symbol $\downarrow$ (read "here") says that the corresponding communication actions must interact in the current endpoint.

Second, we denote the output and the input of messages to/from the enclosing endpoint by the constructs $\texttt{out} \uparrow label(\widetilde{v}).P$ and $\texttt{in} \uparrow label(\widetilde{x}).P$. The direction symbol $\uparrow$ (read "up") says that the corresponding communication actions must interact in the (uniquely determined) enclosing endpoint.

Third, we denote the output and the input of messages to/from the dual endpoint by the constructs $\texttt{out} \leftarrow label(\widetilde{v}).P$ and $\texttt{in} \leftarrow label(\widetilde{x}).P$ The direction symbol $\leftarrow$ (read "other") says that the corresponding communication action must interact with the dual endpoint, relative to the context where the $\texttt{out} \leftarrow$ or $\texttt{in} \leftarrow$ process is running.

Service Publication and Service Instantiation A context 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 the construct $\texttt{def}\ serviceName \Rightarrow ServiceBody$ where $serviceName$ is the service name, and $ServiceBody$ is the process that is to be executed at the service endpoint (responder) 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.,

$$serviceProvider \blacktriangleright [\texttt{def}\ serviceName \Rightarrow ServiceBody \mid \cdots]$$

Such a published service may be instantiated by means of the construct

$$\texttt{instance}\ n\,\rho\ serviceName \Leftarrow ClientProtocol$$

where $n\,\rho$ describes the context (n) and the endpoint role (ρ) where the service is published. For instance, the service defined above may be instantiated by

$$\texttt{instance}\ serviceProvider \blacktriangleright serviceName \Leftarrow ClientProtocol$$

The $ClientProtocol$ describes the process that will run inside the initiator endpoint. The outcome of a service instantiation is the creation of a new globally fresh context

identity (a hidden name), and the creation of two dual endpoints of a context named by this fresh identity. The responder endpoint will contain the $ServiceBody$ process and will be placed at the $serviceProvider$ context. The initiator endpoint will contain the $ClientProtocol$ process and will be placed at the same context as the **instance** expression that requested the service instantiation. The newly created endpoints appear to their enclosing contexts as a local process, and may interact continuously by means of $\uparrow$ communication.

Context Awareness A process running inside a given context is able to dynamically access its identity, by means of the construct **here**$(x).P$. The variable x will be replaced inside the process P by the name n of the current context. The computation will proceed as $P\{x \leftarrow n\}$. This primitive bears some similarity with the **self** or **this** of object-oriented languages, even if it has a different semantics.

Exception Handling We introduce primitives to model exceptional behavior, in particular fault signaling, fault detection, and resource disposal. These aspects are orthogonal to the introduced communication mechanisms, but need to be tackled in any model of service-oriented computation. The primitive to signal exceptional behavior is **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 **try** $-$ **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 **try** P **catch** $Handler$ actively allows a process P to run until some exception is thrown inside P. At that moment, all of P is terminated, and the $Handler$ handler process, which is guarded by **try**$-$**catch**, is activated, concurrently with the continuation $Exception$ of the **throw**.$Exception$ that originated the exception, in the context of a given **try** $-$ **catch**$-$ block. By exploiting the interaction potential of the $Handler$ and $Exception$ processes, one may represent many adequate recovery and resource disposal protocols.

2.1 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 dir for the communication directions, α for directed message labels, and ρ for the endpoint roles ($\rho = \blacktriangleleft$, the initiator role, or $\rho = \blacktriangleright$, the responder role). We denote by $\overline{\rho}$ the dual role of ρ, for instance $\overline{\blacktriangleright} = \blacktriangleleft$. Notice that message and service identifiers (from $\mathcal{L}$) are plain labels, not subject to restriction or binding. The syntax of the calculus is defined in Fig. 1.

The static core of our language is derived from the π-calculus [19]. We thus have **stop** for the inactive process, $P \mid Q$ for the parallel composition, (**new** a)P for name restriction, and $!P$ for replication. Then we have context-oriented polyadic communication primitives: **out** $\alpha(\widetilde{v}).P$ for output and **in** $\alpha(\widetilde{x}).P$ for input. In the communication primitives, α denotes a pair of name and direction, as explained before. We then have the context endpoint access construct $n\,\rho\,[P]$, the context awareness primitive **here**$(x).P$, the service invocation and service definition primitives **instance** $n\,\rho\,s \Leftarrow P$ and **def** $s \Rightarrow P$, respectively. The primitives for exception handling are the **try** P **catch** Q

$$
\begin{array}{llll}
a,b,c,\ldots & \in & \Lambda & (\textit{Names}) \\
x,y,z,\ldots & \in & \mathcal{V} & (\textit{Variables}) \\
n,v,\ldots & \in & \Lambda \cup \mathcal{V} & \\
l,s\ldots & \in & \mathcal{L} & (\textit{Labels}) \\
\mathit{dir} & ::= & \downarrow \mid \leftarrow \mid \uparrow & (\textit{Directions}) \\
\alpha & ::= & \mathit{dir}\ l & \\
\rho & ::= & \blacktriangleright \mid \blacktriangleleft & (\textit{Endpoint Roles})
\end{array}
\qquad
\begin{array}{lll}
P,Q ::= & & \\
 & \mathtt{stop} & \mid n\,\rho\,[P] \\
\mid & P \mid Q & \mid \mathtt{here}(x).P \\
\mid & (\mathtt{new}\ a)P & \mid \mathtt{instance}\ n\,\rho\,s \Leftarrow P \\
\mid & \mathtt{out}\ \alpha(\widetilde{v}).P & \mid \mathtt{def}\ s \Rightarrow P \\
\mid & \mathtt{in}\ \alpha(\widetilde{x}).P & \mid \mathtt{try}\ P\ \mathtt{catch}\ Q \\
\mid & !P & \mid \mathtt{throw}.P
\end{array}
$$

Fig. 1. The Conversation Calculus

and the $\mathtt{throw}.P$. The distinguished occurrences of a, $\widetilde{x}$, and x are binding occurrences in $(\mathtt{new}\ a)P$, $\mathtt{in}\ \alpha(\widetilde{x}).P$, and $\mathtt{here}(x).P$, 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.

We define the semantics of the conversation calculus using a labeled transition system. We introduce transition labels λ. We use act to range over actions, defined as

$$act ::= \boldsymbol{\tau} \mid \alpha(\widetilde{a}) \mid \mathtt{here} \mid \mathtt{throw} \mid \mathtt{def}\ s$$

Then, a transition label λ is an expression as given by $\lambda ::= c\,\rho\,act \mid act \mid (\nu a)\lambda$. 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 $c\,\rho$ is said to be *located at* $c\,\rho$ (or just *located*), otherwise is said to be *unlocated*. We write $(\widetilde{\nu a})$ to abbreviate a (possibly empty) sequence $(\nu a_1)\ldots(\nu a_k)$.

We adopt a few conventions and notations. We note by λ^{dir} a transition label λ^{dir} containing the direction dir $(\uparrow, \leftarrow, \downarrow)$. Then we denote by $\lambda^{\mathit{dir}'}$ the label obtained by replacing dir by dir' in λ^{dir}. Given an unlocated label λ, we represent by $c\,\rho\,\cdot\,\lambda$ the label obtained by locating λ at $c\,\rho$, so that e.g., $c\,\rho\,\cdot\,(\widetilde{\nu a})act = (\widetilde{\nu a})c\,\rho\,act$. We assert $loc(\lambda)$ if λ is not located and does not contain $\mathtt{here}$.

The set of transition labels is polarized and equipped with an injective involution $\overline{\lambda}$ (such that $\overline{\overline{\lambda}} = \lambda$). The involution, used to define synchronizing (matching) transition labels, is defined such that $\overline{act} \neq act'$ for all act, act', and

$$\overline{c\ \rho\ \mathtt{def}\ s} \triangleq c\ \rho\ \overline{\mathtt{def}\ s} \qquad \overline{c\ \rho\ \downarrow \alpha} \triangleq c\ \rho\ \overline{\downarrow \alpha} \qquad \overline{c\ \rho\ \leftarrow \alpha} \triangleq c\ \overline{\rho}\ \overline{\leftarrow \alpha}$$

We define $out(\lambda)$ as $\widetilde{a} \setminus (\widetilde{b} \cup \{c\})$, if $\lambda = (\widetilde{\nu b})\overline{c\,\rho\,\alpha(\widetilde{a})}$ or $\lambda = (\widetilde{\nu b})\overline{\alpha(\widetilde{a})}$. We use *fn*$(\lambda)$ and *bn*(λ) to denote (respectively) the free and bound names of a transition label.

In Figs. 2, 3 and 4 we present the labeled transition system for the calculus. The rules presented in Fig. 2 closely follow the π-calculus labeled transition system (see [20]). In (vii) the unlocated $\leftarrow$ label is excluded (to synchronize it must first get located in some context). We omit the rule symmetric to (vi).

We briefly review the rules presented in Fig. 3: (i) service instantiation request; (ii) service instantiation; (iii) after going through a context boundary, an $\uparrow$ message becomes $\downarrow$; (iv) an unlocated $\downarrow$ message gets located at the context identity in which it originates, analogously (v) for a $\leftarrow$ message and (vi) for service instantiation; (vii) a $\mathtt{here}$ label matches the enclosing context; $(viii)$ a $\mathtt{here}$ label reads the context identity; (ix) a non-$\mathtt{here}$ located label transparently crosses the context boundary, likewise (x)

$$\mathtt{out}\ \alpha(\widetilde{v}).P \xrightarrow{\overline{\alpha(\widetilde{v})}} P \quad (i) \qquad \mathtt{in}\ \alpha(\widetilde{x}).P \xrightarrow{(\widetilde{\nu n})\alpha(\widetilde{v})} P\{\widetilde{x}\leftarrow\widetilde{v}\} \quad (\widetilde{n} \subseteq \widetilde{v}) \quad (ii)$$

$$\frac{P \xrightarrow{\lambda} Q \qquad n \notin \mathit{fn}(\lambda)}{(\mathtt{new}\ n)P \xrightarrow{\lambda} (\mathtt{new}\ n)Q}(iii) \qquad \frac{P \xrightarrow{\lambda} Q \quad n \in \mathit{out}(\lambda)}{(\mathtt{new}\ n)P \xrightarrow{(\nu n)\lambda} Q}(iv) \qquad \frac{P \mid !P \xrightarrow{\lambda} Q}{!P \xrightarrow{\lambda} Q}(v)$$

$$\frac{P \xrightarrow{\lambda} Q \qquad \lambda \neq \mathtt{throw}}{P \mid R \xrightarrow{\lambda} Q \mid R}(vi) \qquad \frac{P \xrightarrow{(\widetilde{\nu n})\lambda} P' \quad Q \xrightarrow{(\widetilde{\nu n})\overline{\lambda}} Q' \qquad \lambda \neq \leftarrow l(\widetilde{a})}{P \mid Q \xrightarrow{\tau} (\mathtt{new}\ \widetilde{n})(P' \mid Q')}(vii)$$

Fig. 2. Basic Operators

$$\mathtt{instance}\ n\,\rho\,s \Leftarrow P \xrightarrow{\overline{(\nu c)n\rho\,\mathtt{def}\ s}} c \blacktriangleleft [P] \quad (i) \qquad \mathtt{def}\ s \Rightarrow P \xrightarrow{(\nu c)\mathtt{def}\ s} c \blacktriangleright [P] \quad (ii)$$

$$\frac{P \xrightarrow{\lambda^{\uparrow}} Q}{n\,\rho\,[P] \xrightarrow{\lambda^{\downarrow}} n\,\rho\,[Q]}(iii) \qquad \frac{P \xrightarrow{\lambda^{\downarrow}} Q}{n\,\rho\,[P] \xrightarrow{n\rho\cdot\lambda^{\downarrow}} n\,\rho\,[Q]}(iv) \qquad \frac{P \xrightarrow{\lambda^{\leftarrow}} Q}{n\,\rho\,[P] \xrightarrow{n\rho\cdot\lambda^{\leftarrow}} n\,\rho\,[Q]}(v)$$

$$\frac{P \xrightarrow{(\nu c)\mathtt{def}\ s} Q}{n\,\rho\,[P] \xrightarrow{(\nu c)n\rho\,\mathtt{def}\ s} n\,\rho\,[Q]}(vi) \qquad \frac{P \xrightarrow{n\rho\,\mathtt{here}} Q}{n\,\rho\,[P] \xrightarrow{\tau} n\,\rho\,[Q]}(vii) \qquad \mathtt{here}(x).P \xrightarrow{n\rho\,\mathtt{here}} P\{x\leftarrow n\} \quad (viii)$$

$$\frac{P \xrightarrow{\lambda} Q \quad \mathit{loc}(\lambda)}{n\,\rho\,[P] \xrightarrow{\lambda} n\,\rho\,[Q]}(ix) \qquad \frac{P \xrightarrow{\tau} Q}{n\,\rho\,[P] \xrightarrow{\tau} n\,\rho\,[Q]}(x) \qquad \frac{P \xrightarrow{(\widetilde{\nu n})act} P' \qquad Q \xrightarrow{(\widetilde{\nu n})\overline{c\rho\,act}} Q'}{P \mid Q \xrightarrow{c\rho\,\mathtt{here}} (\mathtt{new}\ \widetilde{n})(P' \mid Q')}(xi)$$

Fig. 3. Service and Context Operators

$$\mathtt{throw}.P \xrightarrow{\mathtt{throw}} P \quad (i) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{P \mid Q \xrightarrow{\mathtt{throw}} R}(ii) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{n\,\rho\,[P] \xrightarrow{\mathtt{throw}} R}(iii)$$

$$\frac{P \xrightarrow{\lambda} Q \qquad \lambda \neq \mathtt{throw}}{\mathtt{try}\ P\ \mathtt{catch}\ R \xrightarrow{\lambda} \mathtt{try}\ Q\ \mathtt{catch}\ R}(iv) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{\mathtt{try}\ P\ \mathtt{catch}\ Q \xrightarrow{\tau} Q \mid R}(v)$$

Fig. 4. Exception Handling Operators

for a τ label; (xi) an unlocated label synchronizes with a part (the unlocated part) of a located label, originating a `here` label, thus requiring the interaction to occur inside the given context. We omit the rule symmetric to (xi).

As for the rules in Fig. 4: (i) signals an exception; (ii) and (iii) terminate enclosing computations, (iv) a non-`throw` transition crosses the handler block, (v) an exception is caught by the handler block. We omit the rule symmetric to (ii).

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 Examples

In this section, we illustrate the expressiveness of our calculus through a sequence of simple, yet illuminating examples. For the sake of commodity, we informally extend the language with some auxiliary primitives, e.g., $\texttt{if} - \texttt{then} - \texttt{else}$, etc, and recursion $\texttt{rec}\ X.P$ (that may be represented using replication).

3.1 Reading a Remotely Generated Value

A provider *antarctica* provides a service *temperature*. Whenever invoked, such service reads the current value of a sensor at the provider site, and sends it to the caller endpoint.

$$antarctica \blacktriangleright [Sensor \mid \texttt{def}\ temperature \Rightarrow \texttt{in} \uparrow measure(x).\texttt{out} \leftarrow value(x)]$$

By *Sensor* we denote some process running in the $antarctica \blacktriangleright [\cdots]$ context, and that is able to send $measure(t)$ messages inside that context, where t is the current temperature. To use the service in "one shot", a remote client may use the code

$$\texttt{instance}\ antarctica \blacktriangleright temperature \Leftarrow \texttt{in} \leftarrow value(x).\texttt{out} \uparrow temp(x)$$

The effect of this code would be to send a $temp(t)$ message to the client context, where t is the temperature as read at the *antarctica* site. A service delegation as the one just shown resembles a plain remote method call in a distributed object system.

3.2 Service Composition and Orchestration

Our next example, depicted in in Fig. 5, illustrates a familiar service composition and orchestration scenario (inspired by a tutorial example on BPEL published in the Oracle website [15]). Any instance of the $travelApproval$ service is expected to receive a $TravelRequest$ message and return a $clientCallBack$ message after finding a suitable flight. The implementation of the service relies on subsidiary services provided by $americanAirlines$ and $deltaAirlines$ in order to identify the most favorable price.

Notice how the service instance interacts with service side resources in order to find the *travelClass* associated to each *employee*, by means of the *employeeTravelStatusRequest* and *employeeTravelStatusResponse* messages to and from the server context.

Notice also that the service endpoint is used to pass around control messages with the requests and responses to and from the two airline services involved – *flightRequestAA*, *flightRequestDA* and *flightResponseAA*, *flightResponseDA*, respectively. These message exchanges form a loosely-coupled interaction between the orchestration code and the subsidiary service endpoints. There is thus a clear separation between the partner service instances, that adapt the remote endpoint functionalities (or protocols) to the particular roles performed by the instances in this local process, and the orchestration script, that is a process communicating with the several instances via messages. In our view, this separation captures the essence of BPEL's partner links and partner roles, introduced with the motivation of decoupling the description of the business process (the workflow) from the identification and binding to the actual partners involved in the particular service instances.

We discuss an interesting variation of the previous example. We would now like to instantiate the *flightAvailability* services independently (e.g., at site setup time), in the

```
def travelApproval ⇒ (
    instance americanAirlines ▶ flightAvailability ⇐        % Partner americanAirlines
        in ↑ flightRequestAA(flightData, travelClass).
        out ← flightDetails(flightData, travelClass).
        in ← flightTicketCallBack(response, price).
        out ↑ flightResponseAA(response, price)
    |
    instance deltaAirlines ▶ flightAvailability ⇐           % Partner deltaAirlines
        in ↑ flightRequestDA(flightData, travelClass).
        out ← flightDetails(flightData, travelClass).
        in ← flightTicketCallBack(response, price).
        out ↑ flightResponseDA(response, price)
    |
    in ← travelRequest(employee, flightData).               % Orchestration
    out ↑ employeeTravelStatusRequest(employee).
    in ↑ employeeTravelStatusResponse(travelClass).(
        out ↓ flightRequestAA(flightData, travelClass) |
        out ↓ flightRequestDA(flightData, travelClass))
    |
    in ↓ flightResponseAA(flightAA, priceAA).
    in ↓ flightResponseDA(flightDA, priceDA).
    if (priceAA < priceDA) then
        out ← clientCallBack(flightAA)
    else
        out ← clientCallBack(flightDA)
)
```

Fig. 5. The Travel Approval Service

service provider context, rather than creating new instances for each instantiation of the $travelApproval$ service. In other words, the service $deltaAirlines \blacktriangleright flightAvailability$ and the service $americanAirlines \blacktriangleright flightAvailability$ will be used by the orchestration script in the same way as the $employeeTravelStatus$ already was, by means of loosely coupled message exchanges. We depict the solution in Fig. 6. Since many concurrent instantiations of the $travelApproval$ service may be outstanding at any given moment, the need arises to explicitly keep track of the messages relative to each instance (establish a correlation mechanism, in web services terminology). Correlation is achieved by passing the name of the current context (accessed by the **here**$(context)$ primitive) in the request messages to the services instantiated in the shared context (e.g., as in the message $flightRequestAA(context, \cdots)$), allowing the replies associated with the requests to be placed directly in the corresponding contexts.

3.3 Orc

The Orc language [16] is frequently cited as an interesting general model of service orchestration. This example is also relevant to our discussion because Orc also seems to present a mechanism of process delegation, although in a more restricted sense than we are introducing here. In fact, calling a site in Orc causes a persistent process to be

```
instance americanAirlines ▶ flightAvailability ⇐
    ! in ↑ flightRequestAA(r, flightData, travelClass).
      out ← flightDetails(flightData, travelClass).
      in ← flightTicketCallBack(response, price).
      r ▶ [out ↓ flightResponseAA(response, price)]
|
instance deltaAirlines ▶ flightAvailability ⇐
    ! in ↑ flightRequestDA(r, flightData, travelClass).
      out ← flightDetails(flightData, travelClass).
      in ← flightTicketCallBack(response, price).
      r ▶ [out ↓ flightResponseDA(response, price)]
|
! def travelApproval ⇒ (
    in ← travelRequest(employee, flightData).
    here(context).
    out ↑ employeeTravelStatusRequest(context, employee).
    in ↓ employeeTravelStatusResponse(travelClass).(
        out ↑ flightRequestAA(context, flightData, travelClass) |
        out ↑ flightRequestDA(context, flightData, travelClass))
    |
    in ↓ flightResponseAA(flightAA, priceAA).
    in ↓ flightResponseDA(flightDA, priceDA).
    ··· % respond to client as before)
```

Fig. 6. Correlating concurrent conversations.

spawned, consisting the observable behavior of such a process in streaming a sequence of values to the caller context.

We present an encoding of Orc in Fig. 7. To simplify presentation, we introduce anonymous contexts defined as $[P] \triangleq (\mathtt{new}\ n)(n \blacktriangleright [P])$ where n is not used in P. We denote by $[\![O]\!]_{out}$ the encoding of an Orc process O into a conversation calculus process. The out parameter identifies the message label used to output the stream of values generated by the Orc process. So, for instance, in the encoding of Orc's sequential composition $f \gg x \gg g$ each value produced by f (and hence emitted by $[\![f]\!]_{out_1}$ in out_1) will replace x in a new copy of g. The anonymous context guarantees non interference, being the values produced by g forwarded to the upper environment as values produced by $f \gg x \gg g$.

The operational correspondence property between the encoding presented in Fig. 7 and the formal semantics presented in [16] is shown in the technical report [8], where an encoding of a distributed object calculus [7] is also developed.

3.4 Exceptions

We illustrate a few usage idioms for our exception handling primitives in Fig. 8. In Fig. 8 (a) and (b) we show how exceptions can be used to program conversation interruption. As shown in (a) any remote endpoint instance of the *interruptible* service may be interrupted by the service protocol *ServiceProto* by dropping a *stop*() message inside

$$
\begin{array}{ll}
[\![n.S(x)]\!]_{out} & \triangleq \mathtt{instance}\ n \blacktriangleright S \Leftarrow \\
 & \quad (\mathtt{out} \leftarrow args(x).!\mathtt{in} \leftarrow result(x).\mathtt{out} \uparrow out(x)) \\
[\![n.S(x) = e]\!] & \triangleq n \blacktriangleright [!\ \mathtt{def}\ S \Rightarrow (\mathtt{in} \leftarrow args(x).[\![e]\!]_{out} \mid \\
 & \quad\quad !\mathtt{in} \downarrow out(x).\mathtt{out} \leftarrow result(x))] \\
[\![f \gg x \gg g]\!]_{out} & \triangleq [[\![f]\!]_{out_1} \mid \\
 & \quad\quad !\mathtt{in} \downarrow out_1(x).([\![g]\!]_{out_2} \mid \mathtt{in} \downarrow out_2(x).\mathtt{out} \uparrow out(x))] \\
[\![f\ \mathtt{where}\ x :\in g]\!]_{out} & \triangleq [(\mathtt{new}\ x)(\\
 & \quad\quad [\![f]\!]_{out} \mid \\
 & \quad\quad !\mathtt{in} \downarrow out(x).\mathtt{out} \uparrow out(x) \mid \\
 & \quad\quad \mathtt{try} \\
 & \quad\quad\quad [\![g]\!]_{out_2} \mid \mathtt{in} \downarrow out_2(y).\mathtt{throw}\ x \blacktriangleright [\mathtt{out} \leftarrow val(y)] \\
 & \quad\quad \mathtt{catch}\ \mathbf{0})] \\
[\![x]\!]_{out} & \triangleq x \blacktriangleleft [\mathtt{in} \leftarrow val(y).\mathtt{out} \uparrow out(y)] \\
[\![f \mid g]\!]_{out} & \triangleq [\![f]\!]_{out} \mid [\![g]\!]_{out} \\
[\![\mathbf{0}]\!]_{out} & \triangleq \mathbf{0}
\end{array}
$$

Fig. 7. An embedding of Orc.

(a)

```
server ▶ [
  def interruptible ⇒
    in ↓ stop().out ← stop().throw
    | ServiceProto ]
```

(b)

```
instance
    server ▶ interruptible ⇐
        in ← stop().throw
        | ClientProto
```

(c)

```
rec Restart.
   try
      instance
          server ▶ interruptible ⇐ ...
   catch Restart
```

(d)

```
server ▶ [
   def timeBound ⇒
       in ↑ timeAllowed(delay).
       wait(delay).throw
       | ServiceProto ]
```

Fig. 8. Exception handling

the endpoint context. Such a message causes the endpoint to send a *stop*() message to the other (client side) endpoint, and then throwing an exception, which will cause abortion of the service endpoint. On the other hand, the service invocation protocol, shown in (b), will throw an exception at the client endpoint upon reception of *stop*(). Notice that this behavior will possibly happen concurrently with ongoing interactions between *ServiceProto* and *ClientProto*. In Fig. 8 (c) we show a pattern for a client that allows for the recovery of a failure by repeatedly re-launching the service. In Fig. 8 (d) we show a time-aware service definition. 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 a possible extension of our sample language with a $\mathtt{wait}(t)$ primitive).

Somehow related to exceptional behavior is the notion of compensation (see [12]), of particular relevance to service-oriented computing. In the technical report [8] we exhibit an encoding into the conversation calculus of a core fragment of the Compensating CSP calculus [6].

4 Behavioral Semantics

We define a compositional behavioral semantics of the conversation calculus by means of strong bisimulation. The main technical result of this section is a proof that strong bisimilarity is a congruence for all the primitives of our calculus. This further ensures that our syntactically defined constructions induce properly defined behavioral operators at the semantic level. Detailed proofs may be found in the technical report [8].

Definition 4.1. *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 $bn(\lambda) \cap fn(Q) = \emptyset$ then there is Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\mathcal{R}Q'$.

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

Theorem 4.2. *Strong bisimilarity is a congruence for all operators.*

N.B. Here we consider for input prefix the universal instantiation congruence principle: if $P\{x{\leftarrow}n\} \sim Q\{x{\leftarrow}n\}$ for all n then $\mathtt{in}\ \alpha(x).P \sim \mathtt{in}\ \alpha(x).Q$ (cf., [20] Theorem 2.2.8(2)). We may also prove several other behavioral equations of interest.

Proposition 4.3. *The following equations hold up to strong bisimilarity.*

1. $n \blacktriangleright [P] \mid n \blacktriangleright [Q] \sim n \blacktriangleright [P \mid Q]$.
2. $m \blacktriangleright [n \blacktriangleright [o \blacktriangleright [P]]] \sim n \blacktriangleright [o \blacktriangleright [P]]$.
3. $n \blacktriangleright [\mathtt{out} \uparrow m(\widetilde{v}).R] \sim \mathtt{out} \downarrow m(\widetilde{v}).n \blacktriangleright [R]$.
4. $m \blacktriangleright [n \blacktriangleright [\mathtt{out} \downarrow l(\widetilde{v}).P]] \sim n \blacktriangleright [\mathtt{out} \downarrow l(\widetilde{v}).m \blacktriangleright [n \blacktriangleright [P]]]$.
5. $m \blacktriangleright [n \blacktriangleright [\mathtt{out} \leftarrow l(\widetilde{v}).P]] \sim n \blacktriangleright [\mathtt{out} \leftarrow l(\widetilde{v}).m \blacktriangleright [n \blacktriangleright [P]]]$.
6. $m \blacktriangleright [n \blacktriangleright [\mathtt{def}\ s \Rightarrow P]] \sim n \blacktriangleright [\mathtt{def}\ s \Rightarrow P]$
7. $m \blacktriangleright [n \blacktriangleright [\mathtt{instance}\ n\rho\, s \Leftarrow P]] \sim n \blacktriangleright [\mathtt{instance}\ n\rho\, s \Leftarrow P]$

For instance, Proposition 4.3(2) captures the local character of message-based communication in our model. The behavioral identities stated in Proposition 4.3 allow us to prove an perhaps surprising normal form property, that contributes to illuminate the spatial structure of conversation calculus systems. A guarded process is a process of the form $\mathtt{out}\ \alpha(\widetilde{v}).P$ or $\mathtt{in}\ \alpha(\widetilde{x}).P$, $\mathtt{here}(x).P$, $\mathtt{instance}\ n\,\rho\, s \Leftarrow P$, or $\mathtt{def}\ s \Rightarrow P$. We use G to range over parallel compositions of guarded processes. We then have the following

Proposition 4.4. *Let P be a process in the finite exception-free fragment. Then there exist sets of guarded processes $\widetilde{G}, \widetilde{G'}, \widetilde{G''}$, sets of names $\widetilde{a}, \widetilde{b}, \widetilde{c}, \widetilde{d}$, and roles $\widetilde{\rho}, \widetilde{\rho'}, \widetilde{\rho''}$ such that*

$$P \sim (\mathtt{new}\ \widetilde{a})(\ G_1 \mid \ldots \mid G_t \mid b_1\,\rho_1\,[G'_1] \mid \ldots \mid b_j\,\rho_j\,\left[G'_j\right] \\ \mid c_1\,\rho'_1\,[d_1\,\rho''_1\,[G''_1]] \mid \ldots \mid c_k\,\rho'_k\,[d_k\,\rho''_k\,[G''_k]])$$

and where the sequences $b_i\rho_i$ and $c_i\rho'_i d_i \rho''_i$ are all pairwise distinct.

Intuitively, Proposition 4.4 states that any process (of the finite exception-free fragment of the calculus) is behaviorally equivalent to a process where the maximum nesting of contexts is two. The restriction to finite (replication-free) and exception-free processes is sensible, if one just wants to focus on the communication topology.

We may interpret the normal form existence result as follows. A system is composed by several conversation contexts. The set of upward (↑) communication paths of a system may be seen as a graph, where the nodes are processes and contexts, and arcs connect processes to their call-ancestor contexts. As each such arc is uniquely defined by its two terminal nodes, so is the communication structure of an arbitrary process defined (up to bisimilarity) by a system where the (syntactic) nesting of contexts is of at most depth two (see [8]). Intuitively, the structure suggested here represents the join-subconversation relation of concurrently ongoing conversations. Then, the normal form of Proposition 4.4 is analogous to a flattened representation of such a graph.

5 Related work

Various calculi have been recently proposed with the aim to capture aspects of service-oriented computation. At the root of each one, one finds different motivations and methodological approaches. Some intend to model artifacts of the web services technology, in order to develop applied verification techniques (e.g., COWS [18], SOCK [13]), others were introduced in order to demonstrate analysis techniques (e.g., [7, 9]), yet others have the goal of isolating primitives for formalizing and programming service-oriented applications (SCC [3], SSCC [17], CaSPiS [4]) just to refer a few.

The inspiration for the work presented here was motivated by previous developments around SCC [3], a process calculus designed to model service-oriented computing introduced within the Sensoria Project [1]. Our proposal inherits from [14] and SCC the presence of client-server session establishment primitives. However, we end up following a fresh approach, based on the notion of conversation context, and on a simple and flexible message-passing communication. Our development of the concept of conversation context was initially motivated by the concept of session (see [14]). We see conversation contexts as being more general than sessions, in the same sense that coroutining may be seen as a generalization of the stricter procedure (stack-oriented) call discipline. Moreover, the fact that in our model endpoint accesses may appear as arbitrary interacting processes to their enclosing contexts makes them quite different from the more familiar data streaming session endpoints.

Our up (↑) communication primitive was introduced with the aim of expressing the interaction between nested conversation contexts, in particular, between service instances endpoints and their callers, with loose-coupling in mind. Similar primitives have been already introduced in ambient calculi, namely Seal [10], Boxed Ambients [5] and Box π [21]. Our computation model is very different from those models (which are targeted at modeling migration and mobility), as witnessed by Proposition 4.4. Hence, even if formally related to some primitives introduced in [5, 10], at least when their reaction rules are considered in isolation, our communication primitives have very different consequences at the semantic level (for example, two ↑ messages can synchronize, just as long as they originate in subcontexts of the same context).

Primitives to deal with exceptional behavior (for example, closing sessions) are present in several service calculi. Perhaps surprisingly, our exception mechanism, although clearly based on the classical construct for functional languages, does not seem to have been much explored in process calculi; we believe that it allows us to express many interesting exceptional behavior situations.

We have demonstrated that our approach is expressive enough to capture Orc's composition operators; we expect that similar results may be established for calculi with related constructs, such as streams and pipelines [17, 4], at least in the absence of types.

6 Concluding Remarks

We have presented a model for service-oriented computation, building on the identification of some general aspects of service-based systems. We have instantiated our model by proposing the conversation calculus, which incorporates abstractions of the several aspects involved by means of carefully chosen programming language primitives. We have focused our presentation on a detailed justification of the concepts involved, on examples that illustrate the expressiveness of our model, and on the semantic theory for our calculus, based on a standard strong bisimilarity. Our examples demonstrate how our calculus may express many service-oriented idioms in a rather natural way. The behavioral semantics allowed us to prove several interesting behavioral identities. Some of these identities suggested a normal form result that clarifies the spatial communication topology of conversation calculus systems.

Conversation contexts are natural subjects for typing disciplines, in terms of the message interchange patterns that may happen at their borders. We expect types specifying various properties of interfaces, service contracts, endpoint session protocols, security policies, resource usage, and service level agreements, to be in general assigned to context boundaries. One of the most interesting challenges to be addressed by type systems for the conversation calculus is then to discipline the delegation of conversation contexts according to quite strict usage disciplines, allowing for the static verification of systems where several (not just two) partners join and leave dynamically a conversation in a coordinated way.

Acknowledgments We thank our colleagues of the Sensoria Project for many discussions about programming language concepts and core calculi for service based computing. We also acknowledge the anonymous referees for their detailed and useful comments and suggestions.

References

1. IP Sensoria Project, website: http://www.sensoria-ist.eu/.
2. A. Alves and et al. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2006.
3. M. Boreale, R. Bruni, L. Caires, R. De 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*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

4. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. A Service Oriented Process Calculus with Sessioning and Pipelining. Technical report, 2007. Draft.
5. M. Bugliesi, G. Castagna, and S. Crafa. Access Control for Mobile Agents: The Calculus of Boxed Ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.
6. M. J. Butler, C. A. R. Hoare, and C. Ferreira. A Trace Semantics for Long-Running Transactions. In A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors, *25 Years Communicating Sequential Processes*, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2004.
7. L. Caires. Spatial-Behavioral Types for Distributed Services and Resources. In U. Montanari and D. Sanella, editors, *Proceedings of the Second International Symposium on Trustworthy Global Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2006.
8. L. Caires, H. T. Vieira, and J. C. Seco. A Model of Service Oriented Computation. TR-DI/FCT/UNL 6/07, Universidade Nova de Lisboa, 2007.
9. M. Carbone, K. Honda, and N. Yoshida. Structured Global Programming for Communication Behavior. In R. De Nicola, editor, *Proceedings of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science. Springer, 2007.
10. G. Castagna, J. Vitek, and F. Z. Nardelli. The Seal Calculus. *Information and Computation*, 201(1):1–54, 2005.
11. 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, 2006*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer-Verlag, 2006.
12. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
13. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In M. Bravetti, M. N., and G. Zavattaro, editors, *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC 2006)*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 2006.
14. 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.
15. M. B. Juric. A Hands-on Introduction to BPEL, 2006. Oracle (white paper).
16. D. Kitchin, W. R. Cook, and J. Misra. A Language for Task Orchestration and Its Semantic Properties. In C. Baier and H. Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer-Verlag, 2006.
17. 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.
18. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In R. De Nicola, editor, *Proc. of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science. Springer, 2007.
19. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.
20. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
21. P. Sewell and J. Vitek. Secure Composition of Untrusted Code: Box π, Wrappers, and Causality. *Journal of Computer Security*, 11(2):135–188, 2003.