SCC: a Service Centered Calculus*

M. Boreale, R. Bruni, L. Caires, R. De Nicola,
I. Lanese, M. Loreti, F. Martins, U. Montanari,
A. Ravara, D. Sangiorgi, V. Vasconcelos, G. Zavattaro

EU Integrated Project Sensoria
sensoria-core@di.unipi.it

Abstract. We seek for a small set of primitives that might serve as a basis for formalising and programming service oriented applications over global computers. As an outcome of this study we introduce here SCC, a process calculus that features explicit notions of service definition, service invocation and session handling. Our proposal has been influenced by Orc, a programming model for structured orchestration of services, but the SCC's session handling mechanism allows for the definition of *structured interaction protocols*, more complex than the basic *request-response* provided by Orc. We present syntax and operational semantics of SCC and a number of simple but nontrivial programming examples that demonstrate flexibility of the chosen set of primitives. A few encodings are also provided to relate our proposal with existing ones.

1 Introduction

The Sensoria project [17], funded by the European Union, aims at developing a novel, comprehensive approach to the engineering of software systems for service-oriented overlay computers. Specifically, Sensoria focuses on methods and tools for the development of global services that are context adaptive, personalisable, possibly with different constraints on resources and performance, and to be deployed on significatively different global computers. Sensoria seeks for full integration of foundational theories, techniques and methods in a pragmatic software engineering approach.

A crucial role in the project will be played by formalisms for service description that lay the mathematical basis for analysing and experimenting with components interactions, for combining services and formalising crucial aspects of service level agreements.

Industrial consortia are developing orchestration languages, targeting the standardization of Web services and XML-centric technologies. However, existing standards lack clear semantic foundations. We aim at developing a general theory of services that should lead to calculi based on process algebras but enriched with primitives for manipulating semi-structured data (such as pattern matching or unification). The theory should encompass techniques for deriving

* Research supported by the Project FET-GC II IST-2005-16004 Sensoria.

contracts, tools for querying and discovery of service specifications, transactional mechanisms to aggregate unreliable services. The calculi will be equipped with rigorous semantic foundations and analytical tools to prove correctness of system specifications and enable formal verification of properties.

Herewith we present a name passing process calculus with explicit notions of *service definition*, *service invocation* and *bi-directional sessioning*. During the first year of the project a few other work-in-progress proposals emerged [8, 10, 14, 4], and their comparison, refinement and integration will constitute a prominent research activity for the prosecution of the project. Our proposal has been influenced by Cook and Misra's Orc [16], a basic programming model for structured orchestration of services, for which we show a rather natural encoding. In particular, Orc is paticularly appealing because of its simplicity and yet great generality: its three basic composition operators can be used to model the most common workflow patterns, identified by van der Aalst et al. in [18].

Our calculus, called SCC (for Service Centered Calculus), has novel features for programming and composing services, while taking into account their dynamic behaviour. In particular, SCC supports explicit modeling of sessions both on client- and on service-side, including protocols executed by each side during an interaction and mechanisms for session naming and scoping, the latter inspired by the π-calculus. Sessions allow us to describe and reason about interaction modalities more structured than the simple *one-way* and *request-response* modalities provided by Orc and typical of a producer / consumer pattern. Moreover, in SCC sessions can be closed thus providing a mechanism for process interruption and service cancellation and update which has no counterpart in most process calculi.

Summarising, SCC combines the service oriented flavour of Orc with the name passing communication mechanism of the π-calculus. One may argue that we could, in our analysis of service oriented computing, exploit directly π-calculus instead of introducing yet another process calculus. It can be easily seen, from the encoding of a fragment of our calculus in the π-calculus reported in Figure 5, that all the information pertaining to sessioning and client-service protocols get mixed up (if not lost) with the other communication primitives, making it difficult to reason on the resulting process. The motivation behind the introduction of a new calculus is that a small set of well-disciplined primitives will favor and make more scalable the development of typing systems and proof techniques centered around the notions of service and session, for ensuring, e.g., compatibility of client and service behaviour, or the absence of deadlock in service composition.

Within SCC, services are seen as sort of interacting functions (and even stream processing functions) that can be invoked by clients. Service definitions take the form $s \Rightarrow (x)P$, where s is the service name, x is a formal parameter, and P is the actual implementation of the service. For instance, $\mathtt{succ} \Rightarrow (x)x+1$ models a service that, received an integer returns its successor. Service invocations are written as $s\{(x)P\} \Leftarrow Q$: each new value v produced by the client Q will trigger a new invocation of service s; for each invocation, an instance of the process P, with x bound to the actual invocation value v, implements the client-side protocol for interacting with the new instance of s. As an example, a client for the simple service described above will be written in SCC as $\texttt{succ}\{(x)(y)\mathsf{return}\, y\} \Leftarrow 5$: after the invocation x is bound to the argument 5, the client waits for a value from the server and the received value is substituted for y and hence returned as the result of the service invocation.

A service invocation causes activation of a new session. A pair of dual fresh names, r and $\overline{r}$, identifies the two sides of the session. Client and service protocols are instantiated each at the proper side of the session. For instance, interaction of the client and of the service described above triggers the session

$$(\nu r)\big(r \rhd 5+1 \,|\, \overline{r} \rhd (y)\mathsf{return}\, y\big)$$

(in this case, the client side makes no use of the formal parameter). The value 6 is computed on the service-side and then received at the client side, that reduces first to $\overline{r} \rhd \mathsf{return}\, 6$ and then to $6 \,|\, \overline{r} \rhd \mathbf{0}$ (where $\mathbf{0}$ denotes the nil process).

More generally, within sessions communication is bi-directional, in the sense that the interacting protocols can exchange data in both directions. Values returned outside the session to the enclosing environment can be used for invoking other services. For instance, what follows is a client that invokes the service `succ` and then prints the obtained result:

$$\texttt{print}\{(z)\mathbf{0}\} \Leftarrow (\,\texttt{succ}\{(x)(y)\mathsf{return}\, y\} \Leftarrow 5\,)\,.$$

(in this case, the service `print` is invoked with vacuous protocol $(z)\mathbf{0}$).

A protocol, both on client-side and on service-side, can be interrupted (e.g. due to the occurrence of an unexpected event), and interruption can be notified to the environment. More generally, the keyword close can be used to terminate a protocol on one side and to notify the termination status to a suitable handler at the partner site. For example, the above client is extended below for exploiting a suitable service `fault` that can handle printer failures:

$$\texttt{print}\{(z)\mathbf{0}\} \Leftarrow_{\texttt{fault}} (\,\texttt{succ}\{(x)(y)\mathsf{return}\, y\} \Leftarrow 5\,)\,.$$

The formal presentation of SCC involves some key notational and technical solutions that must be well motivated and explained. For this reason, our choice is to give a gentle, step-by-step presentation of the various ingredients.

Synopsis. The paper is organized as follows. Syntax and reduction semantics of the close-free fragment of SCC are presented in Section 2, together with a few motivating examples and encodings. The full calculus is discussed in Section 3. In Section 4 we show how to encode Orc programs into SCC. Some concluding remarks are in Section 5.

2 Persistent sessions: The close-free fragment of SCC

We start by presenting the close-free fragment of SCC, based on three main concepts: (i) service definition, (ii) service invocation, and (iii) bi-directional

$$
\begin{array}{rll}
P, Q \ ::= & \mathbf{0} & \text{Nil} \\
& \mid a.P & \text{Concretion} \\
& \mid (x)P & \text{Abstraction} \\
& \mid \mathsf{return}\, a.P & \text{Return Value} \\
& \mid a \Rightarrow (x)P & \text{Service Definition} \\
& \mid a\{(x)P\} \Leftarrow Q & \text{Service Invocation} \\
& \mid a \rhd P & \text{Session} \\
& \mid P|Q & \text{Parallel Composition} \\
& \mid (\nu a)P & \text{New Name}
\end{array}
$$

Fig. 1. Syntax of processes.

$$
\begin{array}{ccc}
(P|Q)|R \equiv P|(Q|R) & P|\mathbf{0} \equiv P & P|Q \equiv Q|P \\
(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P & (\nu x)\mathbf{0} \equiv \mathbf{0} & a \rhd \mathbf{0} \equiv \mathbf{0} \\
P|(\nu x)Q \equiv (\nu x)(P|Q) \text{ if } x \notin \text{fn}(P) & \multicolumn{2}{c}{a \rhd (\nu x)P \equiv (\nu x)(a \rhd P) \text{ if } x \neq a, \overline{a}} \\
\multicolumn{3}{c}{a\{(x)P\} \Leftarrow (\nu y)Q \equiv (\nu y)(a\{(x)P\} \Leftarrow Q) \text{ if } y \notin \text{fn}((x)P) \cup \{a, \overline{a}\}}
\end{array}
$$

Fig. 2. Structural congruence.

sessioning. We call it PSC for *persistent session calculus*: sessions can be established and garbage collected when the protocol has run entirely, but can neither be aborted nor closed by one of the parties.

Syntax. We presuppose a countable set $\mathcal{N}$ of names $a, b, c, ..., r, s, ..., x, y, ...$ A bijection $\bar{\cdot}$ on $\mathcal{N}$ is presupposed s.t. $\overline{\overline{a}} = a$ for each name a. Note that contrary to common use of the notation, a for input and $\overline{a}$ denoting output, in SCC a and $\overline{a}$ denote dual session names, that can be used for communicating in both directions. The syntax of PSC is in Figure 1, with the operators listed in decreasing order of precedence. Free occurrences of x in P (including $\overline{x}$) are bound in $(\nu x)P$ and $(x)P$. Capture-avoiding substitution of the free occurrences of x with v (and of $\overline{x}$ with $\overline{v}$) in P is denoted by $P[^v/_x]$. Moreover, we identify processes up to alpha-equivalence and we omit trailing $\mathbf{0}$.

Structural congruence. Structural congruence $\equiv$ is defined as the least congruence relation induced by the rules in Figure 2. We include the expected structural laws for parallel composition and restriction, and one rule for garbage collection of completed sessions ($a \rhd \mathbf{0} \equiv \mathbf{0}$). Note that scope extrusion for restriction comes in three different forms.

Well-formedness. Assuming by alpha conversion that all bound names in a process P are different from each other and from the free names, the process P is *well-formed* if each session name a occurs only once (occurrences that can be deleted using structural congruence do not count), but it is allowed to have both sessions $a \rhd Q$ and $\overline{a} \rhd Q'$. In the remainder of this paper, all processes are well-formed. It is straightforward to check that the semantic rules preserve well-formedness. Note that, for the economy of this paper, it is not stricly necessary to introduce dual names, but we prefer to keep this distinction to make evident that once the protocol is started there might still be some reasons for

$$\begin{array}{l}\mathbb{C}[\![\, s \Rightarrow (x)P \,]\!] \mid \\ \mathbb{D}[\![\, s\{(y)P'\} \Leftarrow (Q|u.R) \,]\!]\end{array} \;\rightarrow\; (\nu r)\left(\begin{array}{l}\mathbb{C}[\![\, r \rhd P[{}^{u}\!/\!_{x}] \mid s \Rightarrow (x)P \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \rhd P'[{}^{u}\!/\!_{y}] \mid s\{(y)P'\} \Leftarrow (Q|R) \,]\!]\end{array}\right)$$

if r is fresh and u, s not bound by $\mathbb{C}$,$\mathbb{D}$

$$\mathbb{C}[\![\, r \rhd (P|u.Q) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd (R|(z)S) \,]\!] \;\rightarrow\; \mathbb{C}[\![\, r \rhd (P|Q) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd (R|S[{}^{u}\!/\!_{z}]) \,]\!]$$

if u, r not bound by $\mathbb{C}$,$\mathbb{D}$

$$r \rhd (P|\mathsf{return}\, u.Q) \;\rightarrow\; u \mid r \rhd (P|Q)$$

$$\mathbb{C}[\![P]\!] \;\rightarrow\; \mathbb{C}[\![P']\!] \quad \text{if } P \equiv Q,\ Q \rightarrow Q',\ Q' \equiv P'$$

where $\mathbb{C}, \mathbb{D} \;::=\; [\![\cdot]\!] \mid \mathbb{C}|P \mid a\{(x)P\} \Leftarrow \mathbb{C} \mid a \rhd \mathbb{C} \mid (\nu a)\mathbb{C}$

Fig. 3. Reduction semantics.

distinguishing the two side ends (e.g., when typing is considered).

Operational semantics. Restriction and parallel composition have the standard meaning. Informally, the meaning of the other primitives listed in Figure 1 can be explained as follows. Service definitions take the form $s \Rightarrow (x)P$, where s is the name of the service and $(x)P$ is the body of the service: a (service-side) interaction protocol with formal parameter x. Service definitions are persistent, i.e., each new invocation is served by a fresh instance of the protocol (process calculists may think of an implicit replication prefixing each service definition). A service invocation $s\{(x)P'\} \Leftarrow Q$ invokes s for any concretion (value) u produced by the execution of Q. The process $(x)P'$ is the client-side protocol for interacting with (the instance of) s. For example if after some steps Q reduces to $Q'|u$, then a fresh session can be established, that takes the form

$$(\nu r)(r \rhd P[{}^{u}\!/\!_{x}] \mid \overline{r} \rhd P'[{}^{u}\!/\!_{x}])$$

and that runs in parallel with $s\{(x)P'\} \Leftarrow Q'$ and $s \Rightarrow (x)P$. The session r has two sides: one for the client and one for the service. Note that value u serves as the actual parameter of both P and P'. Within the session r, the protocols $P[{}^{u}\!/\!_{x}]$ and $P'[{}^{u}\!/\!_{x}]$ can communicate whenever a concretion a is available on one side and an abstraction $(z)R'$ is ready on the other side, i.e., abstractions and concretions model input and output, respectively. For example, $r \rhd (a|R) \mid \overline{r} \rhd (z)R'$ would reduce to $r \rhd R \mid \overline{r} \rhd R'[{}^{a}\!/\!_{z}]$. The primitive $\mathsf{return}\, a$ can then be used to return a value outside the current session (just one level up, not to the top level). Sessions, service definitions and service invocations can be nested at arbitrary depth, but in any interaction just the innermost service or session name counts.

Formally, the reduction semantics is defined in Figure 3. The reduction rules are defined using the active contexts $\mathbb{C}, \mathbb{D}$ that specify where the processes that interact can be located. An active context is simply a process with a hole in an active position, i.e., a place where a process can execute. With $\mathbb{C}[\![P]\!]$ we denote the process obtained by filling the hole in $\mathbb{C}$ with P. In PSC only two kinds of interactions are permitted: service invocation and session communication modeled by the first two rules, respectively. A third rule models returned values, that are made available outside the session. Finally, the last rule simply closes the reduction semantics with respect to structural congruence and active contexts.

2.1 Toy examples

We present in this section a few simple examples. Some of them are also used to introduce some shorthand notation for syntax of frequent use.

Precisely, the notations that we introduce are:

$$\begin{aligned} s \Leftarrow P &\text{ Example 1} \\ (-)P &\text{ Example 2} \\ a\{\} \Leftarrow P &\text{ Example 3} \end{aligned}$$

Also, we presuppose a distinct name $\bullet$ to be used as a unit value.

The examples are chosen so to evaluate the expressiveness and usability of PSC as a language for service orchestration, challenging its ability of encoding some frequently used service composition patterns. A library of basic patterns, called the *workflow patterns*, has been identified by van der Aalst et al. in [18]. It will be shown in Section 4 that full SCC can encode Orc [16], a script language for service orchestration able to model the workflow patterns [9].

Example 1 (Functional flavour). A simple example of service invocation is

$$s\{(x)(y)\mathsf{return}\, y\} \Leftarrow v$$

where the service s is invoked with just one value v. The client-side protocol $(x)(y)\mathsf{return}\, y$ has the following meaning: the name x is bound to the invocation value v at invocation time, thus the actual protocol run after service invocation is $((y)\mathsf{return}\, y)[{}^{v}\!/\!_{x}] = (y)\mathsf{return}\, y$ that simply waits for a value as the result of the service invocation and then publishes it locally (outside the private session started upon invocation).

This example reports a typical pattern of service invocation for which we introduce the specific notation $s \Leftarrow P$ which stands for $s\{(x)(y)\mathsf{return}\, y\} \Leftarrow P$. In order to show the advantages of this abbreviation, consider e.g. the functional composition of services: a service f is invoked first (with argument v) and the returned value (if any) is then given as an argument to another service g. With the shorthand notation, then the process can be written as

$$g \Leftarrow (f \Leftarrow v)$$

or simply $g \Leftarrow f \Leftarrow v$, stipulating that $\Leftarrow$ is right-associative. For example, $\mathtt{succ} \Leftarrow \mathtt{succ} \Leftarrow 5$ will return 7. Without abbreviations, one should write something like $\mathtt{succ}\{(z)(w)\mathsf{return}\, w\} \Leftarrow (\mathtt{succ}\{(x)(y)\mathsf{return}\, y\} \Leftarrow v)$.

Example 2 (Pairing service). Starting from this example, to shorten the notation, we use tuples of values $\langle v_1, \ldots, v_n \rangle$ and polyadic abstractions $(a_1, \ldots, a_n)P$. As an example of service definition consider the following *pairing* service

$$pair \Rightarrow (z)(x)(y)\langle x, y\rangle$$

Note that the invocation value z is not used, and the two values to be paired are passed to the protocol executed on service-side from the protocol run on client-side (after service invocation) and bound to x and y respectively. Binding occurrences of names that are not subsequently used (like z above) are abbreviated with $-$. Hence the pairing service can be written as $pair \Rightarrow (-)(x)(y)\langle x, y\rangle$.

A sample usage of the *pairing* service is

$$pair\big\{(-)\big(\,P \mid Q \mid (p)\mathsf{return}\, p\,\big)\big\} \Leftarrow \bullet$$

where P and Q give results to be paired. The pair produced by the service is bound to p and returned as the result. This example also shows that client-side and service-side protocols can exchange values bi-directionally.

Though for the sake of simplicity, this example (and other examples discussed later) might suggest that an instantiation value for starting the session is not always necessary, we have wired its presence in the syntax as a guidance to a uniform style of service programming: in practice it is often the case that sessions can be established only upon authentication checks or that different kinds of sessions are selected based on the kind of the request (e.g. for balancing the load of different servers).

Another point to notice is that inside a session it is possible not only to exchange data with the partner and return values to the environment, but also to input data from outside source (in the example above, this can be achieved by using service invocations within P and Q).

Example 3 (Blind invocation). Sometimes no reply is expected from a service, thus the client employs a vacuous protocol, in which case we just write

$$a\{\} \Leftarrow P$$

for $a\{(-)\mathbf{0}\} \Leftarrow P$. As an example combining the notational conventions seen so far, assume that there are the following available services: service *emailMe* that expects a value *msg* and then sends the message *msg* to your email address; services *ANSA*, *BBC* and *CNN* that return the latest news. Then the process

$$emailMe\{\} \Leftarrow pair\big\{(-)\big(ANSA \Leftarrow \bullet \mid BBC \Leftarrow \bullet \mid CNN \Leftarrow \bullet \mid (p)\mathsf{return}\, p\big)\big\} \Leftarrow \bullet$$

will send you only the first two news items collected from *ANSA*, *BBC* and *CNN*.

Example 4 (Recursion). Service invocations can be nested recursively inside a service definition. For example

$$clock \Rightarrow (-)\big(\mathsf{return}\ \mathtt{tick} \mid clock\{\} \Leftarrow \bullet\big)$$

defines a service that, when invoked with $clock\{\} \Leftarrow \bullet$, produces an infinite number of `tick` values on the service-side. To produce the `tick` values on a specific location different from the service-side, the service to be invoked can be written as

$$remoteClock \Rightarrow (s)\big(s\{\} \Leftarrow \mathtt{tick} \mid remoteClock\{\} \Leftarrow s\big)$$

and a local publishing service

$$pub \Rightarrow (s)\mathsf{return}\, s$$

should be located where the `tick` is to be produced. The name *pub* should be passed to *remoteClock* as argument: $remoteClock\{\} \Leftarrow pub$. This is also an example of service-name passing. The service *pub* (or alike) can be useful in many applications, because it allows to publish values in the location where it is placed. In fact, in PSC *sessions cannot be closed* and therefore recursive invocations on the client-side are nested at increasing depth (while the return instruction can move values only one level up).

Similarly to the last example, a recursive process that receives the name of a service s and a value x and then repeatedly invokes s (the first time on x, then on the last value computed by previous invocations) is shown below:

$$rec \Rightarrow (s,x)\big(s\big\{(-)(y)(\mathsf{return}\, y \mid rec\{\} \Leftarrow \langle s,y\rangle)\big\} \Leftarrow x\big).$$

Again, if the computed values have to be published on the client-side, then the service can carry the name of the publishing service p located on the client-side as an additional parameter:

$$remoteRec \Rightarrow (s,x,p)\big(s\big\{(-)(y)(p\{\} \Leftarrow y \mid remoteRec\{\} \Leftarrow \langle s,y,p\rangle)\big\} \Leftarrow x\big).$$

As an example of invocation of the service *remoteRec*, consider the client

$$remoteRec\{\} \Leftarrow \langle \mathsf{succ}, 0, pub\rangle \mid pub \Rightarrow (x)\mathsf{return}\, x$$

that returns (at the client-side) the stream of positive integers.

Example 5 (Pipeline and forwarder). The process seen at the end of the previous example produces an unbound stream of values. More generally, it should be possible to deploy some sort of pipeline between two services p and q in such a way that q is invoked for each value produced by p. If P is a process that produces a stream of values then the composition $q \Leftarrow P$ already achieves the aim. Thus to compose p and q in a pipeline it suffices to design a client-side protocol for collecting all the values returned by p. If the calculus included a π-calculus like replicator $!P$ or even just abstraction guarded like $!(x)P$, then the protocol could be written just as

$$pipe = (-)!(x)\mathsf{return}\, x.$$

Another possibility is to extend the return prefix so to return an arbitrary process, with syntax $\mathsf{return}\, P.Q$ and semantics:

$$r \rhd (R|\mathsf{return}\, P.Q) \;\rightarrow\; P \mid r \rhd \;(R|Q).$$

Replication can then be coded as follows:

$$!P = (\nu\, rec)\big(\; rec \Rightarrow (-)(\,\mathsf{return}\, P \mid rec\{\} \Leftarrow \bullet\,) \mid rec\{\} \Leftarrow \bullet\;\big).$$

In absence of replicator, one might think to exploit recursion to deploy local receivers of the form $(x)\mathsf{return}\, x$, but unfortunately the implicit nesting of sessions would cause all such receivers to collect values only from different sessions than the original one.

Without extending the syntax of the calculus, a solution is to exploit a publishing service like *pub* above, which must be passed to p (and properly used therein). For instance, if *EATCS* and *EAPLS* return streams of conference announcements on the received service name, then the process

$$emailMe\{\} \Leftarrow \big(pub \Rightarrow (s)\mathsf{return}\, s \mid EATCS\{\} \Leftarrow pub \mid EAPLS\{\} \Leftarrow pub\big)$$

will send you all the announcements collected from *EATCS* and *EAPLS*, one by one. More concisely, this can be equivalently written as

$$EATCS\{\} \Leftarrow emailMe \mid EAPLS\{\} \Leftarrow emailMe.$$

Example 6 (Structured protocols). As an example that requires a more elaborated client-side protocol than those examined so far, let us consider the room reservation service

$$bookRoom \Rightarrow (d)\left(\begin{array}{l} avail \Leftarrow d \mid \\ (cs)(\nu\, code)code.(cc)epay\{(-)cc.(i)\mathsf{return}\, i\} \Leftarrow price \Leftarrow cs \end{array}\right)$$

that must be invoked with the dates d for the reservation, then proposes to the client the set of available rooms for that dates (obtained by invoking the local service *avail* with d), then waits for the client selection *cs* and sends a fresh reservation code to the client, then waits for the credit card number *cc* and debits the price of the selection to the credit card by exploiting a suitable electronic payment service *epay*, and finally, if everything is ok, communicates to the client the confirmation id i obtained from *epay*. Note that we suppose a service *price* that computes the price of the chosen room.

The corresponding client can can be written as:

$$bookRoom\big\{(-)(r)(select \Leftarrow r \mid (c)myCCnum.(cid)\mathsf{return}\, \langle c, cid\rangle)\big\} \Leftarrow \mathtt{dates}$$

It invokes the room reservation service, then waits for the available rooms r, then selects a suitable room (assume the local service *select* is exploited e.g. for interacting with the user) and communicates the choice to the service-side protocol, then waits for the reservation code c before sending the credit card number, and finally waits for the payment confirmation id *cid*, which is returned outside the session together with the reservation code c.

Example 7 (Encoding of the lazy λ-calculus). As a last example, we analyse the expressive power of the PSC in a more traditional manner by discussing the encoding of a typical computational model such as the lazy λ-calculus [3, 1]. We recall that the λ expressions $M, N, ...$ can be either a variable x, the abstraction $\lambda x.M$ or the application $M\, N$, and that the β-reduction rules for the lazy semantics are:

$$(\lambda x.M)N \rightarrow M[^N\!/\!_x] \qquad \frac{M \rightarrow M'}{M\, N \rightarrow M'\, N}$$

$$
\begin{aligned}
[\![x]\!]_p &= x\{\} \Leftarrow p \\
[\![\lambda x.M]\!]_p &= p \Rightarrow (x)(q)[\![M]\!]_q \\
[\![M\,N]\!]_p &= (\nu m)(\nu n)\,(\,[\![M]\!]_m \mid n \Rightarrow (s)[\![N]\!]_s \mid m\{(-)p\} \Leftarrow n\,)
\end{aligned}
$$

Fig. 4. Encoding of the lazy λ-calculus.

The translation is much in the spirit of Milner's encoding of λ-calculus in π-calculus [15]: agents can represent both "functions" and "arguments" which are composed in parallel and interact to β-reduce. Likewise [15], during communication we just transmit *access points* to terms instead of terms themselves.

The encoding is in Figure 4. We use a translation $[\![M]\!]_p$, with p representing the port to be used for interaction between M and the environment. From the point of view of syntax, the main differences w.r.t. the π-calculus encoding are: (i) service definitions replace input and replicated input prefixes; (ii) service invocations (with empty protocol) replace output particles. From the point of view of semantics, the more important differences are: (i) each service invocation opens a new session where the computation can progress (remind that sessions cannot be closed in PSC); (ii) all service definitions will remain available even when no further invocation will be possible!

If on one hand, the encoding witnesses the expressive power of PSC, on the other hand, it also motivates the introduction of some mechanism for closing sessions, like the one available in the full calculus.

2.2 Encoding of PSC into π-calculus

In this subsection we aim to show that PSC can be seen as a disciplined fragment of the π-calculus, where processes can communicate only according to the interaction mechanisms provided by the service oriented metaphor. This strong relationship between PSC and the π-calculus does not hold any longer for the full SCC due to the session interruption mechanism (discussed in the next section) that has no direct couterpart in the π-calculus.

In Figure 5 we define the translation $[\![-]\!]_{in,out,ret}$ from PSC to π-calculus (all the operators not treated in Figure 5 are mapped homomorphically). The encoding is parametric on three names used to receive values from (in), send values to (out), and return values to the enclosing session (ret). These channels mimic the structure of sessions, which is lost in the π-calculus, and must be different w.r.t. service names. To avoid confusion with the use of overline in the π-calculus, in this mapping we use $\tilde{a}$ instead of $\overline{a}$ in the syntax of PSC. Moreover, we assume that the two operators are unrelated, namely, for any name a we have that $\tilde{a}$ and $\overline{a}$ are two distinct names.

The most interesting part is the translation of service invocation. Outputs on channel z where process Q produces the parameters for service invocation are intercepted, and each value v triggers an output on the service name a. The

$$
\begin{aligned}
[\![a\{(x)P\} \Leftarrow Q]\!]_{in,out,ret} &= (\nu z)\big(\ [\![Q]\!]_{in,z,ret} \,|\, !z(x).(\nu r,\tilde{r})\overline{a}\langle r,\tilde{r},x\rangle.[\![P]\!]_{r,\tilde{r},out}\ \big) \\
[\![a \Rightarrow (x)P]\!]_{in,out,ret} &= !a(r,\tilde{r},x).([\![P]\!]_{\tilde{r},r,out}) \\
[\![a \rhd P]\!]_{in,out,ret} &= [\![P]\!]_{a,\tilde{a},out} \\
[\![a.P]\!]_{in,out,ret} &= \overline{out}\ a.[\![P]\!]_{in,out,ret} \\
[\![(x)P]\!]_{in,out,ret} &= in(x).[\![P]\!]_{in,out,ret} \\
[\![\mathsf{return}\, a.P]\!]_{in,out,ret} &= \overline{ret}\ a \,|\, [\![P]\!]_{in,out,ret}
\end{aligned}
$$

Fig. 5. Encoding PSC into π-calculus.

output extrudes two new names r and $\tilde{r}$ to be used for communication between the two sessions to be created, it also carries the parameter v (substituted for x) of the invocation. Note that the client protocol uses the channels r and $\tilde{r}$ above for communication and *out* for returning values. On the service-side, an instance of service protocol is started, using the same channels but swapped (so that input of client is connected to output of service and vice versa).

3 The full Service Centered Calculus

Even though PSC is expressive enough to model service definitions and invocations, it does not provide operators for explicit closing of sessions. Namely, once the two protocols $\overline{r} \rhd P_1$ at client-side and $r \rhd P_2$ at service-side are instantiated (as the effect of a service invocation), the session $\overline{r}$ (resp. r) is garbage collected by the structural congruence only if the protocol P_1 (resp. P_2) reduces to $\mathbf{0}$. However, many sessions can never reduce to $\mathbf{0}$, e.g., those containing service definitions. Also, one may want to explicit program session termination, for instance in order to implement cancellation workflow patterns [18], or to manage abnormal events, or to use timeouts.

The full SCC comprises a mechanism for closing sessions that can be roughly described as follows. Let us consider the session r running the protocol P; we associate to this session a service name k which identifies a *termination handler* service, the first time the protocol P invokes such a service, the session r is closed. The notation that we consider is

$$r \rhd_k P$$

where the name of the termination handler service appears in subscript position. In case P contains an invocation to k, like $k\{(x)P'\} \Leftarrow (Q|v.R)$, the overall session r may be closed, formally

$$r \rhd_k \big(k\{(x)P'\} \Leftarrow (Q|v.R) \mid S\big) \qquad \text{may evolve to} \qquad k\{\} \Leftarrow v$$

where only the invocation to the termination handler service is kept and the session r (thus also the processes Q, R and S) is removed.

The termination handler service is associated to sessions on their instantiation. The intuition that we follow is that the termination of the session on one side, should be communicated to the opposite side. To achieve this, the clients indicate the name of the termination handler service for the session on the service-side, while services manage the termination handler service for the session on the client-side. Nevertheless, an asymmetric approach among the client- and the service-side is adopted, that reflects the asymmetry in the modeling of clients and services that we have already discussed in the previous section.

The syntax of clients becomes $a\{(x)P\} \Leftarrow_k Q$ where, besides the explicit indication of the service a to be invoked, we add the name k of the termination handler service to be associated to the session instantiated on the service-side. We usually omit the subscript k when it is not relevant. On the other hand, services are now specified with the process $a \Rightarrow (x)P : (y)T$ where, besides the service protocol $(x)P$, an additional protocol $(y)T$ is specified which represents the body of a termination handler service that will be instantiated on service invocation; this fresh service will be included in the corresponding session on the client-side.

Finally, the full calculus has a special name close that can be used in the specification of session protocols; this name is replaced by the name of the corresponding termination handler on session instantiation.

Remark 1. A first alternative would be to use close as a primitive for terminating instantaneously *both* the client-side and service-side sessions. This strategy conflicts with parties being in charge for the closing of their own sessions. A second alternative would be to use close as a synchronization primitive, so that the client-side and service-side sessions are terminated when close is encountered on one side and $\overline{\mathsf{close}}$ on the other side. This strategy conflicts with parties being able to decide autonomously when to end their own sessions. The use of termination handler services looks a reasonable compromise: each party can exit a session autonomously but it is obliged to inform the other party.

Example 8. In order to become more familiar with the new service invocation mechanism that includes also the termination handler services, let us consider the following process composed (from left to right) by a termination handler service k, a client willing to invoke service a with value v, and the definition of the service a:

$$\big(k \Rightarrow (x)S : (-)\mathbf{0}\big) \mid \big(a\{(x)P\} \Leftarrow_k (v|Q)\big) \mid \big(a \Rightarrow (x)P' : (y)T\big)$$

This process can start a new session. This happens as soon as the value v is passed to the corresponding service a. The new session is assigned a fresh session name r, identifying the service- and the client-sides with r and $\overline{r}$, respectively. As discussed in the previous section, the protocols $(x)P$ and $(x)P'$ specified by the client and the service, will be installed on the respective sides upon session creation. Moreover, a fresh service name k' is associated to the newly installed termination handler service specified on the service-side. Notice that the new service k' has an associated empty second protocol $(-)\mathbf{0}$. Thus, the freshly activated processes will look like:

$$(\nu r, k')\big(\overline{r} \rhd_{k'} P[{}^{v}\!/\!_{x}][{}^{k'}\!/\!_{\mathsf{close}}] \mid r \rhd_{k} (P'[{}^{v}\!/\!_{x}][{}^{k}\!/\!_{\mathsf{close}}] \mid k' \Rightarrow (y)T[{}^{k}\!/\!_{\mathsf{close}}] : (-)\mathbf{0})\big)$$

The process $P[{}^{v}\!/\!_{x}][{}^{k'}\!/\!_{\mathsf{close}}]$ is the instance of the client session protocol that will exchange values with the instance $P'[{}^{v}\!/\!_{x}][{}^{k}\!/\!_{\mathsf{close}}]$ of the service protocol. Note that the session on the client-side has associated the name k', while the session on the service-side has associated the name k. Moreover, note that the termination handler service on service side is included inside the instantiated session and the name close occurring in its protocol is replaced by k, the name that permits to close the overall session on the service-side.

Example 9 (Closure protocol). A typical usage of termination handler services is to program them to close the current session. This can be achieved on the service-side with the service definition

$$s \Rightarrow (x)P' \,:\, (y)\mathsf{close}\,\{\} \Leftarrow y$$

and on the client-side with the process

$$(\nu\, end)s\{(y)\big(P \mid \mathsf{close}\,\{\} \Leftarrow (end \Rightarrow (x)\mathsf{return}\; x : (-)\mathbf{0})\big)\} \Leftarrow_{end} v$$

Indeed, after invocation of service s with value v, the instantiated sessions will be of the form

$$\begin{array}{l} r \rhd_{end} \big(P'[{}^{v}\!/\!_{x}][{}^{end}\!/\!_{\mathsf{close}}] \mid k' \Rightarrow (y)\,(end\{\} \Leftarrow y) : (-)\mathbf{0}\big) \\ \overline{r} \rhd_{k'} \big(P[{}^{v}\!/\!_{y}][{}^{k'}\!/\!_{\mathsf{close}}] \mid k'\{\} \Leftarrow (end \Rightarrow (x)\mathsf{return}\; x : (-)\mathbf{0})\big) \end{array}$$

Note that in case one of the two session closes, the corresponding notification will cause the closure of the session on the opposite side.

We are now ready to formally define the syntax and semantics of the full SCC. To be complete, we report also some auxiliary definitions already discussed for PSC.

Syntax. We presuppose a countable set $\mathcal{N}$ of names $a, b, c, ..., r, s, ..., x, y, ...$ A distinct name close belongs to this set. A bijection $\bar{\cdot}$ on $\mathcal{N}$ is presupposed s.t. $\overline{\overline{a}} = a$ for each name a.

The syntax of processes $P, Q, ...$ is given in Figure 6 with the operators listed in decreasing order of precedence. All operators have been discussed either in the previous section or in the initial part of this section.

Abstraction $(x)P$ and restriction $(\nu x)P$ act as binders for the name x (and also $\overline{x}$) with scope P. Given a process $a \Rightarrow (x)P : (y)T$, the name close is bound in P and T; given a process $a\{(x)P\} \Leftarrow_k Q$, the name close is bound in P. Notions of free names $\mathrm{fn}(\cdot)$ and alpha-equivalence arise as expected. We identify processes up to alpha-equivalence.

$$
\begin{array}{rll}
P, Q, T, \ldots ::= & \mathbf{0} & \text{Nil} \\
& \mid a.P & \text{Concretion} \\
& \mid (x)P & \text{Abstraction} \\
& \mid \mathsf{return}\, a.P & \text{Return Value} \\
& \mid a \Rightarrow (x)P : (y)T & \text{Service Definition} \\
& \mid a\{(x)P\} \Leftarrow_k Q & \text{Service Invocation} \\
& \mid a \rhd_k P & \text{Session} \\
& \mid P|Q & \text{Parallel Composition} \\
& \mid (\nu a)P & \text{New Name}
\end{array}
$$

Fig. 6. Syntax of processes.

Notational conventions. We omit trailing $\mathbf{0}$. Also, in service invocation we write $a\{\} \Leftarrow_k Q$ for $a\{(x)\mathbf{0}\} \Leftarrow_k Q$. In service definition we write $a \Rightarrow (x)P$ for $a \Rightarrow (x)P : (y)\mathbf{0}$. We also omit k in $a\{(x)P\} \Leftarrow_k Q$ and $a \Leftarrow_k Q$ when it is not relevant. Under these conventions and exploiting operator precedences, the process in the Example 8 would be written

$$k \Rightarrow (x)S \mid a\{(x)P\} \Leftarrow_k (v|Q) \mid a \Rightarrow (x)P' : (y)T$$

and the instantiated session is

$$(\nu r)(\nu k')\big(\overline{r} \rhd_{k'} P[{}^{v}\!/\!_{x}][{}^{k'}\!/\!_{\mathsf{close}}] \mid r \rhd_k (P'[{}^{v}\!/\!_{x}][{}^{k}\!/\!_{\mathsf{close}}] : k' \Rightarrow (y)T[{}^{k}\!/\!_{\mathsf{close}}])\big)$$

Operational semantics. As for PSC, the operational semantics is defined in terms of a structural congruence and a reduction relation. The rules for structural congruence are as in Figure 2 where occurrences of the symbols $\rhd$ and $\Leftarrow$ are replaced by $\rhd_k$ and $\Leftarrow_k$, respectively.

The rules for the reduction semantics of the full calculus are reported in Figure 7.

An auxiliary function *tn* is defined on active contexts that keeps track of the termination names associated to sessions in which the hole of the context is enclosed. This function is used to check whether a service invocation should be interpreted as a closing signal for some of the enclosing sessions. For instance, in the first rule (which is an adaptation of the corresponding first rule in Figure 3) the function *tn* is used to check whether the invocation of the service s must be interpreted as a termination signal or not. The second is a novel rule; in the case the name s of the service to be invoked is a termination name for an enclosing session, the closest of these sessions is closed. The remaining rules are trivial adaptations of the corresponding last three rules of Figure 3.

Example 10 (Service update). Another example where session closing is needed is service update. Consider, for instance, the service

$$\mathit{soccerWorldChampion} \Rightarrow (-)\mathtt{brasil}$$

that returns the name of the last winner of the soccer world championship. The service must be updated as soon as a new team becomes the new world champion.

$$\begin{array}{l}\mathbb{C}[\![\, s \Rightarrow (x)P : (z)T \,]\!] \mid \\ \mathbb{D}[\![\, s\{(y)P'\} \Leftarrow_k (Q|u.R) \,]\!]\end{array} \;\rightarrow\; (\nu r)(\nu k') \left(\begin{array}{l} \mathbb{C}[\![\;\; s \Rightarrow (x)P : (z)T \mid \\ \qquad r \rhd_k (k' \Rightarrow (z)T[{}^{k}\!/\mathsf{close}\,] \mid \\ \qquad\qquad P[{}^{u}\!/x][{}^{k}\!/\mathsf{close}\,])\;]\!] \\ \mid \mathbb{D}[\![\;\; \overline{r} \rhd_{k'} P'[{}^{u}\!/y][{}^{k'}\!/\mathsf{close}\,] \mid \\ \qquad s\{(y)P'\} \Leftarrow_k (Q|R)\;]\!] \end{array} \right)$$

if $s \notin tn(\mathbb{D})$, r, k' are fresh and u, s, k not bound by $\mathbb{C}$,$\mathbb{D}$

$$r \rhd_s \mathbb{D}[\![\, s\{(y)P\} \Leftarrow_k (Q|u.R) \,]\!] \;\rightarrow\; s\{\} \Leftarrow_k u$$

if $s \notin tn(\mathbb{D})$ and u, k not bound by $\mathbb{D}$

$$\mathbb{C}[\![\, r \rhd_k (P|u.Q) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd_{k'} (R|(z)S) \,]\!] \;\rightarrow\; \mathbb{C}[\![\, r \rhd_k (P|Q) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd_{k'} (S[{}^{u}\!/z] \mid R) \,]\!]$$

if u, r not bound by $\mathbb{C}$,$\mathbb{D}$

$$r \rhd_k (P|\mathsf{return}\; u.Q) \;\rightarrow\; u \mid r \rhd_k (P|Q)$$

$$\mathbb{C}[\![\, P \,]\!] \;\rightarrow\; \mathbb{C}[\![\, P' \,]\!] \text{ if } P \equiv Q,\; Q \rightarrow Q',\; Q' \equiv P'$$

where $\mathbb{C}, \mathbb{D} \;::=\; [\![\cdot]\!] \mid \mathbb{C}|P \mid a\{(x)P\} \Leftarrow_k \mathbb{C} \mid a \rhd_k \mathbb{C} \mid (\nu a)\mathbb{C}$

$$\begin{array}{lll}\text{and} & tn([\![\cdot]\!]) = \emptyset & tn(\mathbb{C}|P) = tn(a\{(x)P\} \Leftarrow_k \mathbb{C}) = tn(\mathbb{C}) \\ & tn(a \rhd_k \mathbb{C}) = tn(\mathbb{C}) \cup \{s\} & tn((\nu a)\mathbb{C}) = tn(\mathbb{C}) \setminus \{a\}\end{array}$$

Fig. 7. Reduction semantics.

In PSC there is no way to cancel a definition and replace it with a new one. By contrast, in the full calculus, we can exploit session closing in order to remove services and the termination handler service can be used to instantiate a new version of the same service. Consider, for instance,

$$\begin{array}{l} r \rhd_{new} \left(\begin{array}{l} \mathit{soccerWorldChampion} \Rightarrow (-)\mathtt{brasil} \mid \\ \mathit{new}\{\} \Leftarrow_{new} (\mathit{update} \Rightarrow (y)\mathsf{return}\; y) \end{array} \right) \mid \\ \mathit{new} \Rightarrow (z) \left(\begin{array}{l} \mathit{soccerWorldChampion} \Rightarrow (-)z \mid \\ \mathit{new}\{\} \Leftarrow_{new} (\mathit{update} \Rightarrow (y)\mathsf{return}\; y) \end{array} \right) \end{array}$$

The service *update*, when invoked with a new name z, permits to cancel the currently available service *soccerWorldChampion* and replace it with a new instance that returns the name z. Notice that the service *update* is located within the same session r of the service *soccerWorldChampion*; this ensures that when it invokes the termination handler service k the initial instance of the service *soccerWorldChampion* is removed.

Example 11 (A blog service). We consider a service that implements a *blog*, i.e. a web page used by a web client to log personal annotations. A blog provides two services *get* and *set*, the former to read the current contents of the blog and the latter to modify them. The close-free fragment is not expressive enough to faithfully model such a service because it does not support service update, here needed to update the blog contents.

We use the service *newBlog* as a factory of blogs; this receives three names, the initial content v, the name for the new *get* service, and the name for the new *set* service. Upon invocation, the factory forwards the three received values to the *blog* service which is the responsible for the actual instantiation of the *get* and *set* services:

$$\begin{aligned}
&newBlog \Rightarrow (v, get, set)\big(blog\{\} \Leftarrow_{newBlog} \langle v, get, set\rangle\big) \mid \\
&blog \Rightarrow (v, get, set)\big(\ get \Rightarrow (-)v \mid \\
&\qquad\qquad\qquad\qquad \mathsf{close}\,\{\} \Leftarrow (set \Rightarrow (v')\mathsf{return}\,\langle v', get, set\rangle)\ \ \big)
\end{aligned}$$

Note that the update of the blog contents is achieved by invoking the service close which is bound to *newBlog*; this invocation cancels the currently available *get* and *set* services and delegates to *newBlog* the creation of their new instances passing also the new contents v'.

As an example of a client of the blog service, we consider a process that installs a wiki page with initial contents v, then it adds some new contents v'.

$$\begin{aligned}
&newBlog\{\} \Leftarrow \langle v, get, set\rangle \mid \\
&set\{\} \Leftarrow \big(concat\{(-)v'.get \Leftarrow \bullet | (x)\mathsf{return}\,x\} \Leftarrow \bullet\big)
\end{aligned}$$

The service *concat* simply computes the new contents appending v' to v, that are received in this order after service invocation:

$$concat \Rightarrow (-)(v')(v).v \circ v'$$

Here $\circ$ denotes juxtaposition of blog contents.

4 Encoding Orc in SCC

Orc [16, 9] is one of the emerging basic programming models for service orchestration. In this section we show that SCC is expressive enough to model in a natural manner the Orc language.

We start by a brief overview of Orc. Orc is centered on the idea of service orchestration, and it assumes that basic services, able to perform computations, are available on primitive *sites*. Orc concentrates on invoking and orchestrating those services to reach some goal. Services may *publish* streams of values.

Orc uses the following syntax categories: site names, ranged by $a, b, c, ...$, variables, ranged by $x, y, ...$, values (including site names), ranged by $u, v, ...$. Actual parameters, ranged by $p, q, ...$, can be either values or variables. We use $P, Q, ...$ to range over expressions (since they correspond to processes in SCC) and $E, F, ...$ to range over expression names.

An Orc expression can be either a site call, an expression call or a composition of expressions according to one of the three basic orchestration patterns.

Site call: a site call can have either the form $a(p)$ or $x(p)$. In the first case the site name is known statically, in the other case it is computed dynamically. In both the cases p is the parameter of the call. If p is a variable, then it must be instantiated before the call is made. A site call may publish a value (but it is not obliged to do so).

Expression call: an expression call has the form $E(p)$, and it executes the expression defined by $E(x) \triangleq P$ after having replaced x by p. Here p is passed by reference. Note that expression definitions can be recursive.

Symmetric parallel composition: the composition $P|Q$ executes both P and Q concurrently, assuming that there is no interaction between them. It publishes the interleaving of the two streams of values published by P and Q, in temporal order.

Sequential composition: the composition $P > x > Q$ executes P, and, for each value v returned by P, it executes an instance of Q with v assigned to x. It publishes the interleaving (in temporal order) of the streams of values published by the different instances of Q.

Asymmetric parallel composition: the composition Q **where** $x :\in P$ starts in parallel both P and the parts of Q that do not need x. When P publishes the first value, let say v, it is killed and v is assigned to x. The composition publishes the stream obtained from Q (instantiated with v).

An Orc program is composed by an expression and a set of expression definitions. The encoding of an Orc program in SCC is the parallel composition of the expression and of the expression definitions.

We define now the different cases of the encoding $[\![\text{-}]\!]$. A value is trivially encoded as itself, i.e., $[\![u]\!] = u$. For variables (and thus for actual parameters) we need two different encodings, depending on whether they are passed by name or evaluated. We distinguish the two encodings by different subscripts:

$$[\![x]\!]_n = x \qquad [\![x]\!]_v = x \Leftarrow \bullet$$

The evaluation of a variable x is encoded as a request for the current value to the variable manager of x. Variable managers are created by both sequential composition and asymmetric parallel composition.

In general, both site calls and expression calls are encoded as service invocations returning the published results. Expressions too return their published results. Thus the encoding of an expression definition is simply:

$$[\![E(x) \triangleq P]\!] = E \Rightarrow (x)[\![P]\!]$$

The encoding of Orc expressions is detailed in Figure 8 and explained below:

$[\![a(p)]\!]$: a call to a statically-known site a with argument p is encoded as a service invocation of service a with arguments from $[\![p]\!]_v$;

$[\![x(p)]\!]$: in case the name of the site is stored in a variable x, we first ask the variable manager for x to get its current value v, and then make the site invocation through the auxiliary service *forw*; the result from the site v is received in an inner session, in order to pass the value at top level we use another auxiliary service *pub*;

$$
\begin{aligned}
[\![a(p)]\!] &= a \Leftarrow [\![p]\!]_v \\
[\![x(p)]\!] &= (\nu forw, pub)\big(\ forw\{\} \Leftarrow [\![x]\!]_v \mid \\
&\qquad\qquad\qquad\quad forw \Rightarrow (a)pub\{\} \Leftarrow [\![a(p)]\!] \mid \\
&\qquad\qquad\qquad\quad pub \Rightarrow (y)\mathsf{return}\, y\ \big) \\
[\![E(p)]\!] &= E \Leftarrow [\![p]\!]_n \\
[\![P|Q]\!] &= [\![P]\!] | [\![Q]\!] \\
[\![P > x > Q]\!] &= (\nu z, pub)\big(\ z\{\} \Leftarrow [\![P]\!] \mid \\
&\qquad\qquad\qquad z \Rightarrow (y)(\nu x)(x \Rightarrow (-)y \mid pub\{\} \Leftarrow [\![Q]\!]) \mid \\
&\qquad\qquad\qquad pub \Rightarrow (y)\mathsf{return}\, y\ \big) \\
[\![Q \ \mathbf{where}\, x :\in P]\!] &= (\nu x, z, s)\big(\ [\![Q]\!] \mid (z \Rightarrow (y)(x \Rightarrow (-)y)) \mid \\
&\qquad\qquad\qquad (s\{\} \Leftarrow_z \bullet) \mid s \Rightarrow (-)(\mathsf{close}\,\{\} \Leftarrow [\![P]\!])\ \big)
\end{aligned}
$$

Fig. 8. Encoding of Orc expressions in SCC.

- $[\![E(p)]\!]$**:** an expression call is simply a service call; just notice that variables are passed by name;
- $[\![P|Q]\!]$**:** obvious;
- $[\![P > x > Q]\!]$**:** a private service z is created, where $[\![P]\!]$ will send all computed values; at each invocation service z will activate fresh instances of $[\![Q]\!]$ in parallel with fresh variable managers for x that will serve value requests in $[\![Q]\!]$; in this case too, a service *pub* is used to pass the results at top level;
- $[\![Q \ \mathbf{where}\, x :\in P]\!]$**:** both P and Q are executed (the parts of Q requiring the value of x are stopped since there is no manager for x available yet), but P is executed inside a session: the first value v published by P is used to terminate the session. Also, the termination handler will take the value and create a variable manager for x with this value.

Our encoding allows to simulate Orc orchestration policies inside SCC as far as the asynchronous semantics [16] is concerned (the synchronous semantics is mainly used to deal with timing issues, thus it is left for future extensions of SCC with time). We give here a simple example, inspired by [16], to show how the encoding actually works.

Example 12 (Emailing news in Orc). Let us consider the Orc expression

$$CNN(d)|BBC(d) > x > emailMe(x)$$

which invokes the news services of both *CNN* and *BBC* asking for news of day d. For each reply it sends to me an email with the received news. Thus this expression can send from zero up to two emails.

The encoding is as follows:

$$
\begin{aligned}
(\nu z, pub)\big(&z\{\} \Leftarrow (CNN \Leftarrow d | BBC \Leftarrow d) \mid \\
&z \Rightarrow (y)(\nu x)(x \Rightarrow (-)y \mid pub\{\} \Leftarrow emailMe \Leftarrow x \Leftarrow \bullet) \mid \\
&pub \Rightarrow (y)\mathsf{return}\, y\ \big)
\end{aligned}
$$

We have supposed here to have *CNN*, *BBC* and *emailMe* available as services.

When the expression is executed, both *CNN* and *BBC* are invoked. For each returned value y, z is invoked with that value, a new variable manager is created for it and the email protocol is called with the value taken from the variable, i.e., y. If some acknowledgment is returned by *emailMe*, then it is returned using the auxiliary service *pub*.

5 Conclusions and future work

We have presented SCC, a core calculus for service-oriented applications. SCC draws inspiration from different sources, primarily the π-calculus and Cook and Misra's service orchestration language [16], but enhances them with a mechanism for handling sessions. Sessions permit to model interaction modalities between clients and services which are more structured than the simple *one-way* and *request-response* modalities. Moreover, sessions can be explicitly closed, thus providing for a mechanism of process interruption.

Some features that naturally fall within the scope of service oriented computing have been left out of (well-formed processes in) the present version of the calculus. While distribution of processes over sites is certainly a needed issue in our agenda, the development of a type system is a major goal for future work. Specifically, it seems natural to associate service names with types describing the expected behaviour of clients and services, possibly along the lines of the session type systems in [12] or [11]. We believe that this type system would show the benefits of the concept of session even more clearly. Moreover, typing could be used in a prescriptive way to refine and redesign certain aspects and primitives of our calculus, whenever necessary for rendering their use more natural and smooth. The impact of adding a mechanism of *delegation* deserves further investigation. In fact, delegation could be simply achieved by enabling session-name passing, forbidden in the present version; consequences of this choice at the level of semantics and (prospect) type systems are at the moment not clear, though. For example, while many-party sessions have not been considered here, they could be modeled by passing the name of the current session as an argument to a third invoked service. We also plan to investigate the use of the session-closing mechanism for programming long-running transactions and related compensation policies in the context of web applications, in the vein e.g. of [7, 13], and its relationship with the cCSP and the sagas-calculi discussed in [6]. Finally, integration of XML documents querying, in the vein of e.g. [2, 5], and related typing issues, deserve further consideration.

Acknoweldgements. We warmfully thank Hernán Melgratti, Diego Latella, Mieke Massink, Flemming Nielson for many interesting comments on preliminary versions of this paper. The names SCC and PSC have been chosen as an homage the pioneering process algebras designed by Robin Milner and Tony Hoare.

References

1. S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, 1990.
2. L. Acciai and M. Boreale. Xpi: a typed process calculus for xml messaging. In *Proc. of FMOODS'05*, volume 3535 of *Lect. Notes in Comput. Sci.*, pages 47–66. Springer, 2005.
3. H. Barendregt. *The lambda calculus, its syntax and semantics*. North-Holland, 1984.
4. M. Bartoletti, P. Degano, and G. L. Ferrari. Types and effects for secure service orchestration. In *Proc. of CSFW'06*, 2006. To appear.
5. A. L. Brown, C. Laneve, and L. G. Meredith. Piduce: A process calculus with native xml datatypes. In *Proc. of EPEW'05/WS-FM'05*, volume 3670 of *Lect. Notes in Comput. Sci.*, pages 18–34. Springer, 2005.
6. R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *Proc. of CONCUR'05*, volume 3653 of *Lect. Notes in Comput. Sci.*, pages 383–397. Springer, 2005.
7. R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: extending join. In *Proc. of IFIP TCS'04*, pages 367–379. Kluwer Academics, 2004.
8. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of COORDINATION'06*, volume 4038 of *Lect. Notes in Comput. Sci.*, pages 63–81. Springer, 2006.
9. W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in orc. In *Proc. of COORDINATION'06*, volume 4038 of *Lect. Notes in Comput. Sci.*, pages 82–96. Springer, 2006.
10. G. Ferrari, R. Guanciale, and D. Strollo. Jscl: a middleware for service coordination. In *Proc. of FORTE'06*, Lect. Notes in Comput. Sci. Springer, 2006. To appear.
11. S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In *Proc. of ESOP'99*, volume 1576 of *Lect. Notes in Comput. Sci.*, pages 74–90. Springer, 1999.
12. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP'98*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 122–138. Springer, 1998.
13. C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proc. of FOSSACS'05*, volume 3441 of *Lect. Notes in Comput. Sci.*, pages 282–298. Springer, 2005.
14. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. Technical report, University of Florence, 2006.
15. R. Milner. Functions as processes. *Math. Struct. in Comput. Sci.*, 2(2):119–141, 1992.
16. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 2006. To appear. A preliminary version of this paper appeared in the Lecture Notes for NATO summer school, held at Marktoberdorf in August 2004.
17. Sensoria Project. Public web site. http://www.sensoria-ist.eu/.
18. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.