Provably correct implementations of services⋆

Roberto Bruni[1], Rocco De Nicola[2], Michele Loreti[2], and Leonardo G. Mezzina[3]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Dipartimento di Sistemi e Informatica, Università di Firenze, Italy
[3] IMT Alti Studi Lucca, Italy

Abstract. A number of formalisms have been defined to support the specification and analysis of service oriented applications. These formalisms have been equipped with tools (types or logics) to guarantee the correct behavior of the specified services. Due to the semantic gap between the specification formalism and the programming languages of service oriented overlay computers a critical issue is guaranteeing that correctness is preserved when running the specified systems over available implementations. We have defined a service oriented abstract machine, equipped with a formal structural semantics, that can be used to implement service specification formalisms. We use our abstract machine to implement different service oriented formalisms that have been recently proposed, each posing specific challenges that we can address successfully. By exploiting the SOS semantics of the abstract machine and those of the considered service oriented formalisms we do prove that our implementations are correct (sound and complete). We also discuss possible implementations of other formalisms.

1 Introduction

The explosive growth of the Web has led to the widespread use of communication centered applications (often referred to as *web services*) and to the growth of a new computational paradigm known as *Service Oriented Computing* (SOC). The intrinsic complexity of such an open ended paradigm can better be governed by taking advantage of well-structured and tightly disciplined approaches to the modeling of interaction. With these motivations in mind, a number of formalisms have been defined to support the specification and analysis of service oriented applications at the right level of abstraction.

O [18] is an elegant programming model for structured orchestration. Building on three primitives for parallel composition, pipeline, and some sort of cancellation, arbitrarily complex interaction patterns can be programmed in O by relating sequences of independent service invocations via ad-hoc mechanism (e.g. state variables).

SCC [4] is a formalism based on π−calculus and influenced by O that introduces a specific notion of a *session* as the basic tool for programming services and monitoring the communication graph between client and services. Services are seen as passive objects that can be invoked by clients. Service definitions are specific instances of input prefixed processes. The two endpoints of a session (service and client) can communicate

⋆ Research supported by the Project FET-GC II IST-2005-16004. The second author has been partially supported also by IMT Alti Studi Lucca.

by exchanging messages. A fresh shared name is used to guarantee that messages are exchanged only between partners of the same session, so that two instances of the same persistent service (that was invoked from two different sessions) run separately and cannot interfere. From the original proposal in [4], other variations and refinements of SCC have stemmed. SSCC [15] exploits a *stream* oriented unidirectional communication mechanism; and values produced in a session can be pushed into dedicated queues, accessible via their names. The Conversation Calculus [10] allows bidirectional communications between a child session and its parent session. C SP S [5] is a *dataflow* oriented refinement of SCC that uses pipelines to model the local exchange of information. The improvements concern a pattern matching mechanism for communication and a discipline for the graceful closure of nested sessions.

The central role assigned to sessions by the above calculi and the direct use of operators for modeling interaction renders the logical structure of programs clearer and leads to well disciplined specification languages. Other proposals take multiparty sessions as basic work-units, instead of dyadic sessions. This is e.g. the case of [13], where the number of roles covered by participants is statically fixed by means of some sort of *global contract*, which allows for the enforcement of a rigid type discipline for their trustworthy interaction. Differently, μse (read "muse", after MUltiparty SEssions) [7] allows dynamic join to a session and unbounded number of participants. Type systems for guaranteeing type safety and progress properties are under study [6].

A different thread of research, deeply influenced by XML technologies for Web Services like WS-BPEL, exploits the idea of *correlation sets* to establish session scopes. Roughly, the idea is to route each message to a specific instance of a service depending on the values taken from certain parameters [16, 9]. Though correlation sets guarantee a good expressiveness and match a technology trend, we argue that calculi based on such concepts are less suited for analysis and verification, because most of their interactions are driven by actual values of data. Also, unrelated sessions can interfere with each other if ends up using the "right" values.

A key point for the usefulness of the above formalisms is the availability of tools (types or logics) to specify, check and guarantee the correct behavior of the considered services (see e.g. [14, 17, 23]). It is then important to make sure that the proved properties are preserved when running the specified systems over available implementations. This might not be obvious due to the semantic gap between the specification formalism and the programming languages used for their implementation.

Abstract machines that describe step-by-step execution of programs but omit the many details of real (hardware) machines are a good tool for approaching this problem. They provide an intermediate language for compilation and can be used to bridge the gap between the high level of a programming language and the low level of a real machine. The instructions of an abstract machine are tailored to the particular operations required to implement operations of a specific class of source languages. Indeed, the introduction of layers of abstractions increases maintainability and portability. Moreover, apart for these software engineering considerations, abstract machines are theoretically appealing as they facilitate correctness proofs of the generated code and simplify program analysis and transformations. With this in mind, we have defined a Service Oriented Abstract Machine (SOAM), equipped with a formal semantics, that can be used to implement the service specification formalisms. The machine that we have designed is specifically conceived to permit inter-task communication and thus to guarantee by means of dedicated queues created on service invocation a persistent, protected, communication line that naturally corresponds to the concept of session that is proper of service centered calculi. This enables programs to connect and communicate with other programs and to discover new resources in the network. Besides we use the standard imperative primitives i.e.: assignments, loops and conditionals, to write local programs.

The operational semantics of our abstract machine can be used as the basis for guaranteeing that the properties that have been proved by reasoning on the calculi-based specification are preserved by the actual implementations. Many abstract machine already exists [21, 19, 20, 22], with SOAM we focus on communications in presence of sessions, and we simplify the theory where the problems are already well-studied to keep proofs simpler.

We test our proposal by providing the implementation in SOAM of three very different session-based calculi, namely the session language (SL) of Honda, Vasconcelos and Kubo [12], C SP S and O , each posing specific challenges, like session delegation (SL), pipelining, session nesting and pattern matching (C SP S), and cancelation of activities (O). For all of them we provide a structural translation into the code of our abstract machine and prove the operational correspondence between a process and its encoding: all computations of a process are mimicked by the abstract machine obtained from the translation, and vice versa, the translation of a process only performs meaningful computations, i.e. only those computations that correspond to possible evolutions of the encoded term. The encoding scheme is similar in all three cases: a process P is translated into a network $\mathcal{N} = \mathsf{net}(P)$ which is essentially a soup of state/program pairs $\langle \sigma \vdash \mathcal{C} \rangle$ and named queues $r : h$, where the use of some name could be restricted. Initially each program can be seen as the encoding of a sequential agent, then, as the program evolves, it updates its local state, forks new processes and creates new queues.

To keep actual proofs simple, we assume availability of high level operators such as name restrictions, and concentrate on the problems raised by the actual implementation of the interactions mechanisms via shared queues. We do provide the details about proofs only for SL. The proofs for the encodings of C SP S and O being similar, we highlight only the differences and the key ideas.

The rest of the paper is organized as follows. In the next section we introduce the operational semantics of our abstract machine and discuss its main features. Then, in the three successive sections we consider in turn SL, C SP S and O . We conclude with a few final considerations.

2 A Session Oriented Abstract Machine

In this section we present the formal specification of our virtual machine for services, called SOAM (Service Oriented Abstract Machine). The syntax of SOAM programs $(\mathcal{C}, \mathcal{D})$ is reported in Fig. 1, where expressions are denoted by e, values by $v, w, \ldots$, variables by $x, y, \ldots$, patterns by $\mathcal{F}$, and n is used to denote names which may indicate elements of two different kinds, namely *sessions* ranged over by $r, s \ldots$ and *services* ranged over by a. As usual, we use $\tilde{\cdot}$ to indicate finite sequences of values.

$$
\begin{array}{lll}
\mathcal{C}, \mathcal{D} & ::= & \texttt{skip} \mid x := e \mid \texttt{while}\ e\ \texttt{do}\ \mathcal{C} \mid \texttt{if}\ e\ \texttt{then}\ \mathcal{C}\ \texttt{else}\ \mathcal{D} \mid \mathcal{C}; \mathcal{D} \\
 & \mid & \texttt{new}\ n \mid \texttt{invoke}(v, \langle x, y \rangle, \mathcal{C}) \mid \texttt{offer}(v, \langle x, y \rangle, \mathcal{C}) \\
 & \mid & \texttt{out}(w, \tilde{v}) \mid \texttt{in}(w, \Sigma_{j \in J}(\tilde{\mathcal{F}}_j.\mathcal{C}_j)) \mid \texttt{fork}(\langle x_1, y_1 \rangle, \mathcal{C}_1, \langle x_2, y_2 \rangle, \mathcal{C}_2) \\
e & ::= & \mathsf{f}(\tilde{e}) \mid v \\
\mathcal{F} & ::= & e \mid ?x \\
v, w & ::= & \underline{n} \mid true, false \mid x, y, \ldots \mid n \\
\sigma & ::= & \emptyset \mid x : v, \sigma
\end{array}
$$

Fig. 1. Syntax of SOAM programs

Standard imperative commands are the *assignment*, the *while-do*, the *if-then-else* and the sequencing. `skip` is the empty program and is often omitted from programs when it is in tail position. The remaining commands are used to interact with other programs in the network. Command `new` $n; \mathcal{C}$ allows the creation of a new name. Commands $\texttt{invoke}(v, \langle x_1, y_1 \rangle, \mathcal{C}_1)$ and $\texttt{offer}(v, \langle x_2, y_2 \rangle, \mathcal{C}_2)$ permit the synchronization between service offers and service invocations on v, the additional parameters $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$ are local variables bound in the bodies $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, to a new pair of session queues which allow bidirectional communications: x_1 and x_2 will be bound to the same queue, and similarly for y_1 and y_2. Output $\texttt{out}(w, \tilde{v})$ and input $\texttt{in}(w, \Sigma_{j \in J}(\tilde{\mathcal{F}}_j.\mathcal{C}_j))$ primitives allow, respectively, the output of a tuple $\tilde{v}$ inside session w and the input of a tuple offered within session w that matches one of the patterns $\mathcal{F}_j$: the continuation $\mathcal{C}_j$ will depend on the selected pattern. The primitive `fork` allows the creation of two threads that can communicate by means of dedicated session queues bound to the pairs $\langle x_1, y_1 \rangle$ (for $\mathcal{C}_1$) and $\langle x_2, y_2 \rangle$ (for $\mathcal{C}_2$). When the threads need no communication we write $\texttt{fork}(\mathcal{C}_1, \mathcal{C}_2)$.

Each command is evaluated by exploiting a memory σ which is just a list of pairs binding variables to values. The memory is updated by the substitution function $\sigma[^x/_v]$, which allows to extend the memory with a new binder or to replace an existing association, as inductively defined by letting:

$$
\begin{array}{rcl}
\emptyset\ [^x/_v] & = & x : v, \emptyset \\
(x : v', \sigma')\ [^x/_v] & = & x : v, \sigma' \\
(y : v', \sigma')\ [^x/_v] & = & y : v', (\sigma'[^x/_v]) \quad \text{if } x \neq y
\end{array}
$$

We start by defining the evaluation function for commands and expressions which have only local impact and do not require any interaction over the network (see Fig. 2). The operational semantics of SOAM is based on an evaluation function, $\rightarrow$, which is defined both for expressions and commands: $\langle \sigma \vdash e \rangle \rightarrow v$ means that the result of evaluating expression e with memory σ is v; $\langle \sigma \vdash \mathcal{C} \rangle \rightarrow \langle \sigma' \vdash \mathcal{C}' \rangle$ means that the evaluation of command $\mathcal{C}$ with memory σ leads with a small step to a new memory σ' where the command $\mathcal{C}'$ must be evaluated. Rule (E F) evaluates each parameter first and returns the result of the evaluation of function $\mathsf{f}(v_1, \ldots, v_n)$. Rule (E V) returns the value associated to the variable x while the rule for the evaluation of values v other than variables returns v regardless of σ. Rules for local commands (conditional, while and assignment)

(E F)
$$\frac{\forall i.\langle \sigma \vdash e_i \rangle \rightarrow v_i \quad \mathsf{f}(v_1, \dots v_n) = v}{\langle \sigma \vdash \mathsf{f}(e_1, \dots e_n) \rangle \rightarrow v}$$

(E V)
$$\langle \sigma_1, x : v, \sigma_2 \vdash x \rangle \rightarrow v$$

(E V)
$$\frac{v \neq x}{\langle \sigma \vdash v \rangle \rightarrow v}$$

(M)
$$\langle \sigma \vdash \mathtt{skip}; \mathcal{D} \rangle \rightarrow \langle \sigma \vdash \mathcal{D} \rangle$$

(M)
$$\frac{\langle \sigma \vdash e \rangle \rightarrow v}{\langle \sigma \vdash x := e; \mathcal{D} \rangle \rightarrow \langle \sigma[^x/_v] \vdash \mathcal{D} \rangle}$$

(M T)
$$\frac{\langle \sigma \vdash e \rangle \rightarrow true}{\langle \sigma \vdash \mathtt{while}\ e\ \mathtt{do}\ \mathcal{C}; \mathcal{D} \rangle \rightarrow \langle \sigma \vdash \mathcal{C}; \mathtt{while}\ e\ \mathtt{do}\ \mathcal{C}; \mathcal{D} \rangle}$$

(M F)
$$\frac{\langle \sigma \vdash e \rangle \rightarrow false}{\langle \sigma \vdash \mathtt{while}\ e\ \mathtt{do}\ \mathcal{C}; \mathcal{D} \rangle \rightarrow \langle \sigma \vdash \mathcal{D} \rangle}$$

(MI T)
$$\frac{\langle \sigma \vdash e \rangle \rightarrow true}{\langle \sigma \vdash \mathtt{if}\ e\ \mathtt{then}\ \mathcal{C}_1\ \mathtt{else}\ \mathcal{C}_2; \mathcal{D} \rangle \rightarrow \langle \sigma \vdash \mathcal{C}_1; \mathcal{D} \rangle}$$

(M F)
$$\frac{\langle \sigma \vdash e \rangle \rightarrow false}{\langle \sigma \vdash \mathtt{if}\ e\ \mathtt{then}\ \mathcal{C}_1\ \mathtt{else}\ \mathcal{C}_2; \mathcal{D} \rangle \rightarrow \langle \sigma \vdash \mathcal{C}_2; \mathcal{D} \rangle}$$

Fig. 2. Local evaluation

$\mathcal{N} ::= \mathcal{O}$	(empty net)	$h ::= \emptyset$	(empty queue)
$\mid\ \langle \sigma \vdash \mathcal{C} \rangle$	(running program)	$\mid\ \tilde{v} \cdot h$	(sequence of values)
$\mid\ \mathcal{N} \vert \mathcal{M}$	(network composition)		
$\mid\ (\nu n)\mathcal{N}$	(name restriction)		
$\mid\ r : h$	(session queue)		

Fig. 3. Syntax of SOAM Networks

are standard. For conciseness we give the rules assuming some continuation $\mathcal{D}$ is also present, the simpler cases being recovered when $\mathcal{D} = \mathtt{skip}$.

Defining the operational semantics for the other constructs requires introducing the notion of networks which models the behavior of multiple programs running concurrently. The syntax of networks $\mathcal{N}, \mathcal{M}, \dots$ is reported in Fig. 3. A network can be empty or a program $\mathcal{C}$ running with local store σ, the parallel composition of two networks, a network with specific private names, or a session r with associated a sequence of values h (*a queue*). The set of *free names* fn of a network (defined in Fig. 4) considers the references to service and session names in the memory, in a command and in a sequence of values h. Remind that the only binders for service and session names are $\mathtt{new}\ n; \mathcal{C}$ and $(\nu n)\mathcal{N}$, the set of *bound names* $\mathsf{bn}(\mathcal{N})$ is defined as expected. As usual, processes are considered equal up-to alpha-renaming of bound names (and bound variables).

The operational semantics for the commands operating over networks is reported in Fig. 5(b). It relies on two key auxiliary notions, namely a *structural congruence* relation $\equiv$ (see Fig. 5(a)) and a *pattern matching* function match (see Fig. 5(c)). The structural congruence is similar to the one for π-calculus. Here, we just remark that rule $\langle \sigma \vdash \mathtt{skip} \rangle \equiv \mathcal{O}$ considers a vacuous program equivalent to an empty network and rule $(\nu r)(r : h) \equiv \mathcal{O}$ permits garbage collection of sessions. The pattern matching function match returns a substitution that replaces with values the variables (denoted

$\mathsf{fn}(\mathcal{O}) = \emptyset \qquad \mathsf{fn}(\langle \sigma \vdash \mathcal{C} \rangle) = \mathsf{fn}(\sigma) \cup \mathsf{fn}(\mathcal{C}) \qquad \mathsf{fn}(r : h) = \{r\} \cup \mathsf{fn}(h)$

$\mathsf{fn}(\mathcal{N}|\mathcal{M}) = \mathsf{fn}(\mathcal{N}) \cup \mathsf{fn}(\mathcal{M}) \qquad \mathsf{fn}((\nu n)\mathcal{N}) = \mathsf{fn}(\mathcal{N}) \setminus \{n\}$

Fig. 4. Free names $\mathsf{fn}(\mathcal{N})$ of a network

by $?x$) within the pattern. In Fig. 5(b) and Fig. 5(c), we do rely also on the following notation: ρ denotes a generic substitution; ϵ is the empty substitution; the composition of two substitutions is denoted $\cdot$ and we use $\sigma\rho$ and $\mathcal{C}\rho$ to denote the application of the substitution ρ to a state σ or to a command $\mathcal{C}$, respectively. Moreover, we use $\rho_{|R}$ to denote a substitution ρ whose domain is restricted to the set of *R-Value*; i.e. to a function defined only for variables that never appear on the left hand side of an assignment. Correspondingly $\rho_{|L}$ is defined only for those variables that have been previously bound to actual values. In rule (M), the R-value part of the substitution is applied directly to the continuation while the other component is applied to memory σ.

Output operation, rule (M), outputs the tuple $\tilde{v}$, in the session bound within σ to variable w. Input rule (M) non deterministically takes one of the value in the session queue bound to w within σ which pattern-matches one the patterns $\mathcal{F}_k$. Rule (M R) is used to create a new (empty) session queue r that is private of the continuation ($\mathcal{C}$) and can be possibly extruded to the network and used to exchange values. Rule (M S) is similar but does not create any queue, it only guarantees privateness of name a.

We provide two mechanisms for process communication, namely we use command `fork` for interprocess communication and the pair of commands `offer-invoke` for inter-site communication. Rule (M) says that when a `fork` operation is invoked two new threads are created duplicating the current memory σ, and two empty session queues are created to be used later for bidirectional communication between the two new threads. To guarantee sharing of information between the threads, each of the copies of the memory σ_i is enriched by bindings the two fresh session queue names r and r' to the variables of the pair $\langle x_i, y_i \rangle$. The rule for inter-site communication (M) is similar to (M) but there is no memory duplication and the bindings of the two fresh session queue names r and r' to the variables of the pair $\langle x_i, y_i \rangle$ is performed within memories σ and σ'.

Next we show a sort of non interference property valid for SOAM programs that we use in correctness proof of each encoding, namely each transition different from an input (rule (M)) or an intra site synchronization (rule (M)) can be freely interleaved. Roughly, this helps since no matter how such actions are interleaved in the network, the result is the same. As we shall see, from the point of view of correctness proofs this means that each transition in a given calculus can be simulated by multiple machine instructions, as long as we maintain a one to one correspondence among input and synchronization transitions of the machine to input and synchronization transitions of the calculus (see Theorem 2).

In the following, we introduce some syntax. Sometimes, it is important to establish the origin of a transition therefore we denote with $\xrightarrow{\square}$ a transition originated from rules (M) and (M) and we use $\xrightarrow{\blacksquare}$ otherwise.

$$\langle \sigma \vdash \texttt{skip} \rangle \equiv \mathcal{O} \qquad \mathcal{O}|\mathcal{N} \equiv \mathcal{N} \qquad \mathcal{N}|\mathcal{M} \equiv \mathcal{M}|\mathcal{N} \qquad \mathcal{N}|(\mathcal{M}|\mathcal{M}') \equiv (\mathcal{N}|\mathcal{M})|\mathcal{M}'$$

$$(\nu n)(\nu m)\mathcal{N} \equiv (\nu m)(\nu n)\mathcal{N} \qquad n \notin \mathsf{fn}(\mathcal{N}) \Rightarrow \mathcal{N}|(\nu n)\mathcal{M} \equiv (\nu n)(\mathcal{N}|\mathcal{M}) \qquad (\nu r)(r : h) \equiv \mathcal{O}$$

(a) Structural congruence over networks

(M)

$$\frac{\langle \sigma \vdash w \rangle \to r}{\langle \sigma \vdash \texttt{out}(w, \tilde{v}); \mathcal{C} \rangle | r : h \to \langle \sigma \vdash \mathcal{C} \rangle | r : \tilde{v} \cdot h}$$

(M)

$$\frac{\langle \sigma \vdash w \rangle \to r \quad \mathsf{match}(\sigma, \tilde{\mathcal{F}}_k, \tilde{v}) = \rho}{\langle \sigma \vdash \texttt{in}(w, \Sigma_{j \in J}(\tilde{\mathcal{F}}_j.\mathcal{C}_j)); \mathcal{D} \rangle | r : h \cdot \tilde{v} \cdot h' \to \langle \sigma\rho_{|L} \vdash \mathcal{C}_k\rho_{|R}; \mathcal{D} \rangle | r : h \cdot h'}$$

(M R)

$$\frac{r \notin \mathsf{fn}(\sigma)}{\langle \sigma \vdash \texttt{new}\ r; \mathcal{C} \rangle \to (\nu r)(\langle \sigma \vdash \mathcal{C} \rangle | r : \emptyset)}$$

(M S)

$$\frac{a \notin \mathsf{fn}(\sigma)}{\langle \sigma \vdash \texttt{new}\ a; \mathcal{C} \rangle \to (\nu a)\langle \sigma \vdash \mathcal{C} \rangle}$$

(M)

$$\frac{r, r' \notin \mathsf{fn}(\langle \sigma \vdash \mathcal{C}_1 \rangle) \cup \mathsf{fn}(\langle \sigma \vdash \mathcal{C}_2; \mathcal{D} \rangle) \quad \rho_1 = [^r/_{x_1}][^{r'}/_{y_1}] \quad \rho_2 = [^r/_{x_2}][^{r'}/_{y_2}]}{\langle \sigma \vdash \texttt{fork}(\langle x_1, y_1 \rangle, \mathcal{C}_1, \langle x_2, y_2 \rangle, \mathcal{C}_2); \mathcal{D} \rangle \to \begin{array}{l} (\nu r)(\nu r')(\langle \sigma_1\rho_{1|L} \vdash \mathcal{C}_1\rho_{1|R} \rangle | \\ \langle \sigma_2\rho_{2|L} \vdash (\mathcal{C}_2; \mathcal{D})\rho_{2|R} \rangle | r : \emptyset | r' : \emptyset) \end{array}}$$

(M)

$$\frac{\forall\, i \in \{1, 2\} \quad \langle \sigma_i \vdash v_i \rangle \to a \quad \rho_i = [^{x_i}/_r][^{y_i}/_{r'}] \quad r, r' \notin \mathsf{fn}(\langle \sigma_i \vdash \mathcal{C}_i \rangle)}{\begin{array}{l} \langle \sigma_1 \vdash \texttt{offer}(v_1, \langle x_1, y_1 \rangle, \mathcal{C}_1); \mathcal{D} \rangle | \\ \langle \sigma_2 \vdash \texttt{invoke}(v_2, \langle x_2, y_2 \rangle, \mathcal{C}_2); \mathcal{D}' \rangle \end{array} \to \begin{array}{l} \langle \sigma_1 \vdash \mathcal{D} \rangle | \langle \sigma_2 \vdash \mathcal{D}' \rangle | (\nu r)(\nu r')(r : \emptyset | r' : \emptyset | \\ \langle \sigma_1\rho_{1|L} \vdash \mathcal{C}_1\rho_{1|R} \rangle | \langle \sigma_2\rho_{2|L} \vdash \mathcal{C}_2\rho_{2|R} \rangle) \end{array}}$$

(M)

$$\frac{\mathcal{N} \equiv \mathcal{N}' \quad \mathcal{N}' \to \mathcal{M}' \quad \mathcal{M}' \equiv \mathcal{M}}{\mathcal{N} \to \mathcal{M}}$$

(M)

$$\frac{\mathcal{N} \to \mathcal{N}'}{(\nu n)\mathcal{N} \to (\nu n)\mathcal{N}'}$$

(M)

$$\frac{\mathcal{N} \to \mathcal{N}'}{\mathcal{N}|\mathcal{M} \to \mathcal{N}'|\mathcal{M}}$$

(b) Network evaluation

$$\frac{\langle \sigma \vdash e \rangle \to v}{\mathsf{match}(\sigma, e, v) = \epsilon} \qquad \mathsf{match}(\sigma, ?x, v) = [^v/_x]$$

$$\frac{\mathsf{match}(\sigma, \mathcal{F}_1, v_1) = \rho_1 \ldots \mathsf{match}(\sigma, \mathcal{F}_n, v_n) = \rho_n}{\mathsf{match}(\sigma, \mathcal{F}_1, \ldots, \mathcal{F}_n, v_1, \ldots, v_n) = \rho_1 \cdot \ldots \cdot \rho_n}$$

(c) Pattern matching function

Fig. 5. Network interactions

$P, Q ::= 0$	$\mid$ `if` e `then` P `else` Q		(null / conditional)
$\mid\ (\nu n)P$	$\mid\ P \vert Q$		(restriction / parallel)
$\mid$ `accept` $a(r)$ `in` P	$\mid$ `request` $a(r)$ `in` P		(session acceptance / request)
$\mid\ r?(\tilde{x})$ `in` P	$\mid\ r!(\tilde{e}); P$		(input / output)
$\mid\ r \triangleright \{l_1; P_1 \| \dots \| l_n; P_n\}$	$\mid\ r \triangleleft l; P$		(label branching / selection)
$\mid$ `throw` $r(r'); P$	$\mid$ `catch` $r(r');$ `in` P		(session sending / receiving)

(a) Syntax of Honda et al. session calculus

(L) $(\texttt{accept}\ a(x)\ \texttt{in}\ P)|(\texttt{request}\ a(x)\ \texttt{in}\ Q) \rightarrow \texttt{accept}\ a(x)\ \texttt{in}\ P|(\nu r)(P[{}^{r^+}/_x]|Q[{}^{r^-}/_x])$

(C) $(r^p!(\tilde{e}); P)|(r^{\overline{p}}?(\tilde{x})\ \texttt{in}\ Q) \rightarrow (P|Q[{}^{\tilde{v}}/_{\tilde{x}}])$ if $\tilde{e} \rightarrow \tilde{v}$

(L) $(r^p \triangleleft l; P)|(r^{\overline{p}} \triangleright \{l_1; P_1 | \dots | l_n; P_n\}) \rightarrow (P|P_i)$ $(1 \leq i \leq n)$

(P) $(\texttt{throw}\ r^p(r'^q); P)|(\texttt{catch}\ r^{\overline{p}}(x);\ \texttt{in}\ Q) \rightarrow P|Q[{}^{r'^q}/_x]$

(I 1) $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \rightarrow P$ if $e \rightarrow true$

(I 2) $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \rightarrow Q$ if $e \rightarrow false$

(S) $P \rightarrow Q \quad \Rightarrow \quad (\nu n)P \rightarrow (\nu n)Q$

(P) $P \rightarrow P' \quad \Rightarrow \quad P|Q \rightarrow P'|Q$

(S) $P \equiv P'$ and $P' \rightarrow Q'$ and $Q' \equiv Q \quad \Rightarrow \quad P \rightarrow Q$

(b) Operational semantics of Honda et al. session calculus

Fig. 6. Honda et al. session calculus

As usual $\rightarrow^*$ (resp. $\xrightarrow{\square}{}^*$, resp $\xrightarrow{\blacksquare}{}^*$) is the reflexive and transitive closure of $\rightarrow$ (resp. $\xrightarrow{\square}$, resp $\xrightarrow{\blacksquare}$). Finally $\mathcal{M}' \leftarrow \mathcal{N} \rightarrow \mathcal{M}''$ indicates the possibility for $\mathcal{N}$ to evolve in either $\mathcal{M}'$ or $\mathcal{M}''$.

Proposition 1. *If* $\mathcal{M}' \xleftarrow{\square} \mathcal{N} \xrightarrow{\blacksquare} \mathcal{M}''$ *then* $\mathcal{M}' \xrightarrow{\blacksquare} \equiv \mathcal{N}' \equiv\xleftarrow{\square} \mathcal{M}''$

Proof. By induction on the structure of $\mathcal{N}$ and on the derivation of the proofs for $\xrightarrow{\blacksquare}$ and $\xrightarrow{\square}$.

Corollary 1. *If* $\mathcal{M}' \xleftarrow{\square} \mathcal{N} \xrightarrow{\blacksquare}{}^* \mathcal{M}''$ *then* $\mathcal{M}' \xrightarrow{\blacksquare}{}^* \equiv \mathcal{N}' \equiv\xleftarrow{\square} \mathcal{M}''$

3 Honda-Vasconcelos-Kubo Session Language

In this section we show how to encode in SOAM the popular session calculus in [12]. The set of variables $x, y, \dots$ and services a are the same to previous sections. In addition we use a denumerable set of labels l and a denumerable set of polarized session names r^p. Syntactic categories, expressions e, patterns $\mathcal{F}$, and values $v, w \dots$ are the same as in Fig. 1. The syntax of the calculus is reported in Fig. 6(a), differently from the original proposal we miss the recursive definitions which can be easily accommodated inserting functions call in SOAM. We however maintain a recursive flavor by replicating services. Without recursive definition the structural congruence is the π-calculus one.

Labeled branching $r \triangleright \{l_1; P_1 | \dots | l_n; P_n\}$ is sometimes abbreviated $r \triangleright \{l_i; P_i\}_i$.Binders for session r are `accept` $a(r)$ `in` P, `request` $a(r)$ `in` P, `catch` $r'(r);$ `in` P and $(\nu r)P$, binders for service a are only (νa). The derived notions of free and bound names, session names and service names are standard. In the following we assume each process *is close*d at least w.r.t. session names. The operational semantics is the smallest relation on processes $P \rightarrow Q$ generated by the rules in Fig. 6(b), where $e \rightarrow v$ says that expression e evaluates to the value v. With respect to the original proposal we use an operational semantics more similar to [24] where polarities appear, because the original rule for session communication would otherwise need a runtime check for session linearity (see [12]), which we do not want to impose in a general machine like SOAM.

Rule (L) establishes a new session between the service `accept` $a(r)$ `in` P and the client `request` $a(r)$ `in` P via shared name a. Rule (C) transmits tuples between the client and the service at the private channel so that determinacy or value delivery is ensured among the two parties. Rule (S) allows a process to choose from a series of offered options in the dual side. Rule (P) is the key rule to allow higher-order session communication, i.e. session channel send and receive with which various protocols are expressed, allowing complex nested structured communications.

Since the calculus is synchronous we assume a primitive for synchronous output $\texttt{Sout}(w, \tilde{v}); \mathcal{C}$ in SOAM which is achieved with standard technique. We do not detail further on the implementation but the rule needs to behave as follows.

(MS)

$$\frac{\forall i \in \{1,2\} \quad \langle \sigma_i \vdash w_i \rangle \rightarrow r \quad \mathsf{match}(\sigma_1, \mathcal{F}_k, \tilde{v}) = \rho}{\begin{array}{l}\langle \sigma_1 \vdash \texttt{in}(w_1, \Sigma_{j\in J}(\tilde{\mathcal{F}}_j.\mathcal{C}_j)); \mathcal{D}\rangle| \\ \langle \sigma_2 \vdash \texttt{Sout}(w_2, \tilde{v}); \mathcal{C}\rangle|r : h\end{array} \rightarrow \langle \sigma_1\rho_{|L)} \vdash \mathcal{C}_k\rho_{|R}; \mathcal{D}\rangle|\langle \sigma_2 \vdash \mathcal{C}\rangle|r : h}$$

Notice that (M) originates a $\xrightarrow{\square}$ transition. As outlined in the introduction, the translation of the calculus is given by means of two functions net and prg: the former returns the SOAM network associated to a process P while the latter returns the static program associated to a process. Intuitively prg is used to block the simulation of a process under a prefix until the prefix is consumed. For example in $r^+!(4); (r^+!(5)|r^+!(6))$ we want the delivery of 5 and 6 to be performed *after* the delivery of 4. In fact, later we prove that whenever $\mathsf{prg}(P)$ is active, then it can become $\mathsf{net}(P)$ after some internal evaluation steps. The translation in Fig. 7 has some redundancy, because two different translations must be considered depending of when the session subject is a variable x or when the session subject is a session side r^p. A variable x appearing as a session subject is translated in two variables x^+ and x^- the former used to output tuples and the latter to input tuples. Label choice and label selection are viewed as input/ouput actions too.

In case that the session subject is a polarized session name r^p, in the same way we have two variables r^p and $r^{\overline{p}}$ but the translation is parametric on p. The polarity information p says that r^p is used for output operations and $r^{\overline{p}}$ is used for input operations. Notice that for ease of clarity, with a (slight) abuse of notation we use polarized sessions and variables in the machine too. In case of sessions communication `throw` and `catch` both session sides are sent/received. Function prg is similar to net, but without memory.

Let start proving some facts. The next proposition says that none of the variables generated during the translation is an R-Value and according to the operational semantics of SOAM this implies that all the substitutions are applied directly to the commands.

Proposition 2. *Let x a variable appearing in the translation* $\mathsf{net}(P)$ *then x is not an R-Value.*

$$
\begin{array}{ll}
\mathtt{net}((\nu a)P) = (\nu a)(\mathtt{net}(P)) & \\
\mathtt{net}((\nu r)P) = (\nu r^{+})(\nu r^{-})(\mathtt{net}(P)|r^{+} : \emptyset|r^{-} : \emptyset) & \\
\mathtt{net}(P|Q) = \mathtt{net}(P)|\mathtt{net}(Q) & \\
\mathtt{net}(P) = \langle\emptyset \vdash \mathtt{prg}(P)\rangle \qquad \text{otherwise} & \\
\mathtt{prg}(\mathtt{accept}\ a(x)\ \mathtt{in}\ P) = \mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{offer}(a, \langle x^{+}; x^{-}\rangle, \mathtt{fork}(\mathtt{prg}(P), \mathtt{skip})) & \\
\mathtt{prg}(\mathtt{request}\ a(x)\ \mathtt{in}\ P) = \mathtt{invoke}(a, \langle x^{-}; x^{+}\rangle, \mathtt{prg}(P)) & \\
\mathtt{prg}(x!(\tilde{e}); P) = \mathtt{Sout}(x^{+}, \tilde{e}); \mathtt{prg}(P) & \mathtt{prg}(r^{p}!(\tilde{e}); P) = \mathtt{Sout}(r^{p}, \tilde{e}); \mathtt{prg}(P) \\
\mathtt{prg}(x?(\tilde{y})\ \mathtt{in}\ P) = \mathtt{in}(x^{-}, (\tilde{?y}.\mathtt{prg}(P))) & \mathtt{prg}(r^{p}?(\tilde{y})\ \mathtt{in}\ P) = \mathtt{in}(r^{\overline{p}}, (\tilde{?y}.\mathtt{prg}(P))) \\
\mathtt{prg}(x \lhd l; P) = \mathtt{Sout}(x^{+}, l); \mathtt{prg}(P) & \mathtt{prg}(r^{p} \lhd l; P) = \mathtt{Sout}(r^{p}, l); \mathtt{prg}(P) \\
\mathtt{prg}(x \rhd \{l_i; P_i\}_i) = \mathtt{in}(x^{-}, \Sigma_i(l_i.\mathtt{prg}(P_i))) & \mathtt{prg}(r^{p} \rhd \{l_i; P_i\}_i) = \mathtt{in}(r^{\overline{p}}, \Sigma_i(l_i.\mathtt{prg}(P_i))) \\
\mathtt{prg}(\mathtt{throw}\ x(r^{p}); P) = \mathtt{Sout}(x^{+}, (r^{\overline{p}}, r^{p})); \mathtt{prg}(P) & \\
\mathtt{prg}(\mathtt{throw}\ r'^{q}(r^{p}); P) = \mathtt{Sout}(r'^{q}, (r^{\overline{p}}, r^{p})); \mathtt{prg}(P) & \\
\mathtt{prg}(\mathtt{throw}\ x(y); P) = \mathtt{Sout}(x^{+}, (y^{-}, y^{+})); \mathtt{prg}(P) & \\
\mathtt{prg}(\mathtt{throw}\ r^{p}(y); P) = \mathtt{Sout}(r^{p}, (y^{-}, y^{+})); \mathtt{prg}(P) & \\
\mathtt{prg}(\mathtt{catch}\ x(y); \mathtt{in}\ P) = \mathtt{in}(x^{-}, ((y^{-}, y^{+}).\mathtt{prg}(P))) & \\
\mathtt{prg}(\mathtt{catch}\ r^{p}(y); \mathtt{in}\ P) = \mathtt{in}(r^{\overline{p}}, ((y^{-}, y^{+}).\mathtt{prg}(P))) & \\
\mathtt{prg}(\mathtt{if}\ v = w\ \mathtt{then}\ P\ \mathtt{else}\ Q) = \mathtt{if}\ v = w\ \mathtt{then}\ \mathtt{prg}(P)\ \mathtt{else}\ \mathtt{prg}(Q) & \\
\mathtt{prg}((\nu a)P) = \mathtt{new}\ a; \mathtt{prg}(P) & \\
\mathtt{prg}((\nu r)P) = \mathtt{new}\ r^{+}; \mathtt{new}\ r^{-}; \mathtt{prg}(P) & \\
\mathtt{prg}(P|Q) = \mathtt{fork}(\mathtt{prg}(P), \mathtt{prg}(Q)) &
\end{array}
$$

Fig. 7. Translation of the session calculus in SOAM

Next proposition relates the substitutions of the calculus with the encoding.

Proposition 3. *Let v a value different from a session then* $\mathsf{prg}(P)[^{v}/_{x}] = \mathsf{prg}(P[^{v}/_{x}])$. *Moreover* $\mathsf{prg}(P)[^{r^{+}}/_{x^{+}}][^{r^{-}}/_{x^{-}}] = \mathsf{prg}(P[^{r^{+}}/_{x}])$ *and* $\mathsf{prg}(P)[^{r^{-}}/_{x^{+}}][^{r^{+}}/_{x^{-}}] = \mathsf{prg}(P[^{r^{-}}/_{x}])$

As discussed before each programs takes some internal "adjustment steps" to come from $\mathsf{prg}(P)$ to $\mathsf{net}(P)$.

Proposition 4. *Let* $\mathsf{prg}(P)$ *defined then* $\langle\emptyset \vdash \mathsf{prg}(P)\rangle \xrightarrow{\blacksquare}{}^{*} \mathsf{net}(P)$.

The two main theorems state the completeness and the correctness of the translation, in particular the machine needs multiple steps to simulate a single transition of the calculus, and in the correctness proof we must show that the interleaving of these actions, with other actions due to other network components does not cause troubles (e.g., deadlock).

Theorem 1 (Completeness). *If $P \rightarrow Q$ then* $\mathsf{net}(P) \rightarrow^{*} \equiv \mathsf{net}(Q)$.

Proof. The proof is by induction on the derivation of $P \rightarrow Q$. We sketch the base case when the rule (C) is applied. Since we are considering closed process we have $P = (\nu r)(r^{p}!(\tilde{e}); P_1)|(r^{\overline{p}}?(\tilde{x})\ \mathtt{in}\ P_1))$ and $\mathsf{net}(P) = (\nu r^{+})(\nu r^{-})(\langle\emptyset \vdash \mathtt{Sout}(r^{+}, \tilde{e}); \mathsf{prg}(P_1)\rangle|\langle\emptyset \vdash \mathtt{in}(r^{-}, (\tilde{?x}.\mathsf{prg}(P_2)))\rangle) \rightarrow (\nu r^{+})(\nu r^{-})(\langle\emptyset \vdash \mathsf{prg}(P_1)\rangle|\langle\emptyset \vdash \mathsf{prg}(P_2)[^{\tilde{v}}/_{\tilde{x}}]\rangle) \underbrace{\rightarrow \ldots \rightarrow}_{\text{Prop. 3 and 4}} (\nu r^{+})(\nu r^{-})(\mathsf{net}(P_1)|\mathsf{net}(P_2[^{\tilde{v}}/_{\tilde{x}}]))$. Notice that due to Prop. 2 R-Values are not bound by susbtitutions. In particular, the last equality is exactly $\mathsf{net}(Q)$ which concludes the proof.

The proof of correctness is split in two parts, first we prove a proposition of local correctness in which we consider the simulation of each transition of the process calculus without worry about the interleaving with other networks components, then we prove the final theorem.

Proposition 5 (Local-Correctness). *If* $\mathsf{net}(P) \rightarrow \mathcal{N}_1 \rightarrow \ldots \rightarrow \mathcal{N}_k \rightarrow\equiv \mathsf{net}(Q')$ *for some* $k > 0$ *and* $\nexists P_{\mathcal{N}}$ *s.t.* $\mathcal{N}_i \equiv \mathsf{net}(P_{\mathcal{N}})$*, then* $P \rightarrow^* Q$ *and* $Q \equiv Q'$.

Proof. By structural induction on the depth of the derivation of $\rightarrow$ and case analysis on the structure of $\mathsf{net}(P)$. We sketch the base case when $P = (\mathtt{accept}\ a(x)\ \mathtt{in}\ P_1)|(\mathtt{request}\ a(x)\ \mathtt{in}\ P_2)$. We have that

$$\mathsf{net}(P) = \langle \emptyset \vdash \mathcal{C}\rangle | \langle \emptyset \vdash \mathtt{invoke}(a, \langle x^-; x^+\rangle, \mathsf{prg}(Q))\rangle$$

where

$$\mathcal{C} = \mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{offer}(a, \langle x^+; x^-\rangle, \mathtt{fork}(\mathsf{prg}(P), \mathtt{skip}))$$

we have that:

$$\begin{aligned}
&\langle \emptyset \vdash \mathcal{C}\rangle | \langle \emptyset \vdash \mathtt{invoke}(a, \langle x^-; x^+\rangle, \mathsf{prg}(P_2))\rangle \\
&\quad \rightarrow^* \langle \emptyset \vdash \mathtt{offer}(a, \langle x^+; x^-\rangle, \mathsf{prg}(P_1))\rangle | \langle \emptyset \vdash \mathcal{C}\rangle | \langle \emptyset \vdash \mathtt{invoke}(a, \langle x^-; x^+\rangle, \mathsf{prg}(P_2))\rangle \\
&\quad \rightarrow (\nu r^+)(\nu r^-)(\langle \emptyset \vdash \mathsf{prg}(P_1)[^{r^+}/_{x^+}][^{r^-}/_{x^-}]\rangle | \langle \emptyset \vdash \mathsf{prg}(P_2)[^{r^+}/_{x^-}][^{r^-}/_{x^+}]\rangle) | \langle \emptyset \vdash \mathcal{C}\rangle \\
&\quad \rightarrow^* (\nu r^+)(\nu r^-)(\langle \emptyset \vdash \mathsf{prg}(P_1[^{r^+}/_x])\rangle | \langle \emptyset \vdash \mathsf{prg}(P_2[^{r^-}/_x])\rangle) | \langle \emptyset \vdash \mathcal{C}\rangle \\
&\underbrace{\rightarrow \ldots \rightarrow}_{\text{Prop. 3}} (\nu r^+)(\nu r^-)(\langle \emptyset \vdash \mathsf{net}(P_1[^{r^+}/_x])\rangle | \langle \emptyset \vdash \mathsf{net}(P_2[^{r^-}/_x])\rangle) | \langle \emptyset \vdash \mathcal{C}\rangle = \mathcal{M}
\end{aligned}$$

It is easy to prove that each net occurring in the above derivation, but for $\mathcal{M}$ and $\mathsf{net}(P)$, does not correspond to any process Q. At the same time we have that

$$P \rightarrow \mathtt{accept}\ a(x)\ \mathtt{in}\ P_1 | (\nu r)(P_1[^{r^+}/_x] | P_2[^{r^-}/_x])$$

and, as expected, $\mathcal{M} = \mathsf{net}(\mathtt{accept}\ a(x)\ \mathtt{in}\ P_1 | (\nu r)(P_1[^{r^+}/_x] | P_2[^{r^-}/_x]))$.

Next is a technical lemma which establishes that one $\xrightarrow{\square}$ transition can be freely interleaved with arbitrary $\xrightarrow{\blacksquare}$ transitions without compromising the final result.

Lemma 1. *If* $\mathsf{net}(P) \xrightarrow{\blacksquare}{}^* \mathcal{M} \xrightarrow{\square} \mathcal{N}$ *and* $\mathcal{N} \xrightarrow{\blacksquare}{}^* \equiv \mathsf{net}(Q)$ *then* $P \rightarrow^* Q$

Theorem 2 (Correctness). *If* $\mathsf{net}(P) \rightarrow^* \mathcal{M}$ *then either* $\mathcal{M} \equiv \mathsf{net}(Q')$ *or there exists* $k > 0$ *s.t.* $\mathcal{M} \underbrace{\rightarrow \ldots \rightarrow}_{k} \equiv \mathsf{net}(Q')$ *and* $P \rightarrow^* Q$ *with* $Q \equiv Q'$.

Proof. The proof proceeds by first proving a slight weak statement, that is all the transitions in $\mathcal{M} \underbrace{\rightarrow \ldots \rightarrow}_{k} \equiv \mathsf{net}(Q')$ are $\xrightarrow{\blacksquare}$ transitions. The statement is proved by induction on the number of $\xrightarrow{\square}$ transition in $\mathsf{net}(P) \rightarrow^* \mathcal{M}$. In the base case where there are neither inputs nor synchronizations we use Proposition 1 and Corollary 5, to conclude. In the inductive cases we use Lemma 1 on the last $\xrightarrow{\square}$ transition together with Proposition 1 and Proposition 5 to conclude. Then, the theorem is proved by induction on the number of $\xrightarrow{\square}$ transitions in $\mathcal{M} \underbrace{\rightarrow \ldots \rightarrow}_{k} \equiv \mathsf{net}(Q')$.

$$
\begin{array}{lllr}
P, Q ::= 0 & \mid \ \mathtt{if}\ v = w\ \mathtt{then}\ P\ \mathtt{else}\ Q & & \text{(null / conditional)} \\
\mid \ (\nu n)P & \mid \ P|Q & & \text{(restriction / parallel)} \\
\mid \ a.P & \mid \ \overline{v}.Q & & \text{(service definition / invocation)} \\
\mid \ r^p \triangleright P & \mid \ P > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i & & \text{(session / pipeline)} \\
\mid \ \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i & \mid \ \langle \tilde{e} \rangle & & \text{(abstraction / concretion)} \\
\mid \ !a.P & \mid \ !\Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i & & \text{(replications)} \\
\mid \ \langle \tilde{e} \rangle^{\uparrow}.P & & & \text{(return)}
\end{array}
$$

Fig. 8. Syntax of C SP S

4 CaSPiS

C SP S *(Calculus of Sessions and Pipelines)* [5] is a core calculus where *sessions* and *pipelines* are viewed as natural tools for structuring client-service interaction and service orchestration. The syntax of the calculus is reported in Fig. 8. We have the same standard sets of names, variables and expressions, but we redefine the set of values to avoid session names, since in C SP S sessions are not first order values. The only binder for names (sessions and services) is (νn), and in the following we consider translation of only processes without free session names. Processes are considered up to alpha-conversion of bound names and modulo the structural congruence relation $\equiv$ which is standard but in addition allows floating of restrictions in $P > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i$ and in $r \triangleright P$ and garbage collection of empty sessions (see [5]).

In C SP S, *service definition* and *service invocation* are respectively rendered as $a.P$ and $\overline{a}.Q$, where a is the name of the service (rule (I)). However, differently from π-calculus, P and Q are not ordinary continuations but rather the protocols governing the interaction between client (service invoker) and server (service provider). Synchronization of $a.P$ and $\overline{a}.Q$ leads to the creation of a *new* session, identified by a fresh name r that can be viewed as a private, synchronous channel binding caller and callee. Since client and service may be far apart, a session naturally comes with two sides, written $r^+ \triangleright P$ and $r^- \triangleright Q$. Processes at the two sides of a session can interact with each other by means of *concretions* ($\langle \tilde{e} \rangle.P$) and *abstractions* ($\Sigma_{i=0}^n(\tilde{\mathfrak{F}}).P$): the former *produce* a value resulting from the evaluation of the expressions tuple $\tilde{e}$ while the latter *read* a value matching one of pattern $\tilde{\mathfrak{F}}_i$ (rule (C)). Values produced by P via concretions can be consumed by abstractions in Q, and vice-versa. Values can be returned outside a session to the enclosing environment using the return operator, $\langle \cdot \rangle^{\uparrow}$ rule (R). These values can be sent over sessions, or used to invoke other services, to start new activities. This is achieved using the *pipeline* operator, written $P > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.Q_i$. Here, a *new* instance of process Q_j is activated each time P emits a value matching $\tilde{\mathfrak{F}}_j$. Such new instance runs in parallel with $P' > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.Q_i$, where P' is the continuation of P.

We present the operational semantics in Fig. 9(a) by exploiting reduction contexts in Fig. 9(b). We have four different types of contexts generated from the respective grammar. The one-hole context $\mathbb{C}[\![\cdot]\!]$ is useful to insert a process P into an arbitrary nesting of sessions with arbitrary processes in parallel, the result being denoted $\mathbb{C}[\![P]\!]$. The one-hole context $\mathbb{C}_{r^p}$ allows inserting a process into the session r^p together with an

(I) $\mathbb{D}[\![\overline{a}.P, a.Q]\!] \;\rightarrow\; (\nu r)\mathbb{D}[\![r^- \triangleright P, r^+ \triangleright Q]\!]$ if $r \notin \mathsf{fn}(\mathbb{D}[\![\overline{a}.P, a.Q]\!])$

(C) $\mathbb{D}_r[\![\Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i, \langle\tilde{e}\rangle]\!] \;\rightarrow\; \mathbb{D}_r[\![P_j[^{\tilde{v}'}/_{\tilde{x}}], 0]\!]$ if $\tilde{e} \rightarrow \tilde{v}$ and $\mathsf{match}(\tilde{\mathfrak{F}}_j, \tilde{v}) = [^{\tilde{v}'}/_{\tilde{x}}]$

(R) $\mathbb{C}_{r_1^q}[\![\mathbb{C}_{r^p}[\![\langle\tilde{e}\rangle^{\uparrow}.Q]\!]]\!] \;\rightarrow\; \mathbb{C}_{r_1^q}[\![\langle\tilde{v}\rangle|\mathbb{C}_{r^p}[\![Q]\!]]\!]$ if $\tilde{e} \rightarrow \tilde{v}$

(B) $\mathbb{D}_r[\![!P, Q]\!] \;\rightarrow\; \mathbb{D}_r[\![!P|P, Q]\!]$

(I T) $\mathbb{C}[\![\texttt{if}\ v = w\ \texttt{then}\ P\ \texttt{else}\ Q]\!] \rightarrow \mathbb{C}[\![P]\!]$ if $(v = w) \rightarrow true$

(I F) $\mathbb{C}[\![\texttt{if}\ v = w\ \texttt{then}\ P\ \texttt{else}\ Q]\!] \rightarrow \mathbb{C}[\![Q]\!]$ if $(v = w) \rightarrow false$

(P) $\mathbb{C}[\![(\langle\tilde{v}\rangle|P') > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i]\!] \;\rightarrow\; \mathbb{C}[\![(P') > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i \mid P_j[^{\tilde{v}}/_{\tilde{x}}]]\!]$ $\mathsf{match}(\tilde{\mathfrak{F}}_j, \tilde{v}) = [^{\tilde{v}'}/_{\tilde{x}}]$

(S) $P \;\rightarrow\; P' \;\Rightarrow\; (\nu n)P \rightarrow (\nu n)P'$

(S) $P \equiv P' \;\wedge\; P' \rightarrow Q' \;\wedge\; Q' \equiv Q \Rightarrow\; P \rightarrow Q$

(a) CaSPiS operational semantics

$\mathbb{C} \;::=\; [\![\cdot]\!] \;\mid\; \mathbb{C}|P \;\mid\; r^p \triangleright \mathbb{C} \;\mid\; \mathbb{C} > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i$ $\qquad \mathbb{C}_{r^p} \;::=\; r^p \triangleright ([\![\cdot]\!]|P)$

$\mathbb{D} \;::=\; \mathbb{C}[\![\mathbb{C}' \mid \mathbb{C}'']\!]$ $\qquad \mathbb{D}_r \;::=\; \mathbb{D}[\![\mathbb{C}'_{r^p}, \mathbb{C}''_{\overline{r^p}}]\!] \quad r \notin \mathsf{fn}(\mathbb{D})$

(b) Contexts

Fig. 9. C SP S

arbitrary process in parallel. Contexts $\mathbb{D}$ and $\mathbb{D}_r$ are the two-holes counterparts of the previous contexts. Note that in each context, binders cannot appear above the hole.

As before, the encoding in SOAM machine is given by means of the two functions net and prg. However, due to the fact that sessions can be nested and that processes can be inserted in a pipeline, the function net needs three sessions for determining the surrounding communication context: the input session used to address the receive operations the output session used to address the sending operations and the return session used to send values to parent session. We have two overloaded version of the net function. The first one without parameters is initially called to prepare the network and to create three new fresh sessions to be used for top level operations.

The other function $\mathsf{net}(P, r_i, r_o, r_r)$ creates a network for P assuming P can input from r_i, output values to r_o and return values to r_r. Consider the net-encoding of $r^p \triangleright P$, the surrounding session r^p says that r^p is used for output, $r^{\overline{p}}$ for inputs and the old session r_o used before as outputs session can be used now as a return session. A program generated from the function prg can statically access to the value of the r_i,r_o and r_r referencing the three variables $m1^-$, $m1^+$, $m2^+$ respectively. In fact, the encoding makes an additional work to prepare the memory in order to have these variables correctly set. Thanks to the operational semantics of rule (M) each program can be evaluated only in a memory containing only these three values. To this end we assume that the programmers of a process cannot access to the variables $m1^-$, $m1^+$ and $m2^+$. Note that the rule for service definition $a.P$ sets in memory the variable $m2^+$ used for return values to the session previously used for output. The replicated nature of a service definition is simulated by a while construct and the value for variables $m1^-$, $m1^+$ used respectively for input and output are set by the machine once the connection is established. Similarly for service invocation, but the variables $m1^+$, $m1^-$ are provided in inverse order to allow communication. Encoding of net for a pipe requires the creation of a new fresh queue r_t used by P to output values to a forever input which forks a new copy of Q with the received value bound in $\tilde{x}$. The notation $?\tilde{x}$ creates a pattern $?x_i$ for each variable

$$\mathtt{net}(P) = (\nu r_i)(\nu r_o)(\nu r_r)(\mathtt{net}(P, r_i, r_o, r_r) | r_i : \emptyset | r_o : \emptyset | r_r : \emptyset) \text{ where } r_i, r_o, r_r \text{ fresh}$$
$$\mathtt{net}(0, r_i, r_o, r_r) = \langle \emptyset \vdash \mathtt{skip} \rangle$$
$$\mathtt{net}(r^p \triangleright P, r_i, r_o, r_r) = \mathtt{net}(P, r_{\overline{p}}, r_p, r_o)$$
$$\mathtt{net}(a.P, r_i, r_o, r_r) = \langle \emptyset[^{r_o}/_{m2^+}] \vdash \mathtt{offer}(a, \langle m1^+, m1^- \rangle, \mathtt{prg}(P)) \rangle$$
$$\mathtt{net}(!a.P, r_i, r_o, r_r) = \langle \emptyset[^{r_o}/_{m2^+}] \vdash \mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{offer}(a, \langle m1^+, m1^- \rangle, \mathtt{fork}(\mathtt{prg}(P), \mathtt{skip})) \rangle$$
$$\mathtt{net}((\nu r)P, r_i, r_o, r_r) = (\nu r^+)(\nu r^-)(\mathtt{net}(P, r_i, r_o, r_r) | r^+ : \emptyset | r^- : \emptyset)$$
$$\mathtt{net}((\nu a)P, r_i, r_o, r_r) = (\nu a)(\mathtt{net}(P, r_i, r_o, r_r))$$
$$\mathtt{net}(P|Q, r_i, r_o, r_r) = \mathtt{net}(P, r_i, r_o, r_r) | \mathtt{net}(Q, r_i, r_o, r_r)$$
$$\mathtt{net}(\overline{a}.Q, r_i, r_o, r_r) = \langle \emptyset[^{r_o}/_{m2^+}] \vdash \mathtt{invoke}(a, \langle m1^-, m1^+ \rangle, \mathtt{prg}(Q)) \rangle$$
$$\mathtt{net}(\mathtt{if}\ v = w\ \mathtt{then}\ P\ \mathtt{else}\ Q, r_i, r_o, r_r) =$$
$$\quad \langle \emptyset[^{r_o}/_{m1^+}][^{r_i}/_{m1^-}][^{r_r}/_{m2^+}] \vdash \mathtt{if}\ v = w\ \mathtt{then}\ \mathtt{prg}(P, r_i, r_o, r_r)\ \mathtt{else}\ \mathtt{prg}(Q, r_i, r_o, r_r) \rangle$$
$$\mathtt{net}(\Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i, r_i, r_o, r_r) = \langle \emptyset[^{r_o}/_{m1^+}][^{r_i}/_{m1^-}][^{r_r}/_{m2^+}] \vdash \mathtt{in}(m1^-, \Sigma_{i \in \{1,\ldots,n\}}(\tilde{\mathfrak{F}}_i.\mathtt{prg}(P_i))) \rangle$$
$$\mathtt{net}(\langle \tilde{e} \rangle, r_i, r_o, r_r) = \langle \emptyset[^{r_o}/_{m1^+}][^{r_i}/_{m1^-}][^{r_r}/_{m2^+}] \vdash \mathtt{out}(m1^+, \tilde{e}) \rangle$$
$$\mathtt{net}(\langle \tilde{e} \rangle^{\uparrow}.P, r_i, r_o, r_r) =$$
$$\quad \langle \emptyset[^{r_o}/_{m1^+}][^{r_i}/_{m1^-}][^{r_r}/_{m2^+}] \vdash \mathtt{out}(m2^+, \tilde{e}) \rangle | \langle \emptyset[^{r_o}/_{m1^+}][^{r_i}/_{m1^-}][^{r_r}/_{m2^+}] \vdash \mathtt{prg}(P) \rangle$$
$$\mathtt{net}(!\Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i, r_i, r_o, r_r) = \langle \emptyset[^{r_o}/_{m1^+}][^{r_i}/_{m1^-}][^{r_r}/_{m2^+}] \vdash$$
$$\qquad \mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{in}(m1^-, \Sigma_{i \in \{1\ldots n\}}(\tilde{\mathfrak{F}}_i.\mathtt{fork}(\mathtt{prg}(P_i), \mathtt{skip}))) \rangle$$
$$\mathtt{net}(P > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i, r_i, r_o, r_r) = (\nu r_t)(\mathtt{net}(P, r_i, r_t, r_r) | \langle \emptyset[^{r_o}/_{m1^+}][^{r_i}/_{m1^-}][^{r_r}/_{m2^+}] \vdash$$
$$\quad \mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{in}(r_t, (\Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.\mathtt{fork}(\mathtt{prg}(P_i), \mathtt{skip}))) \rangle | r_t : \emptyset) \quad \text{where } r_t \text{ fresh}$$
$$\mathtt{prg}(0) = \mathtt{skip}$$
$$\mathtt{prg}(r^p \triangleright P) = m2^+ := m1^+; m1^{\overline{p}} := r^{\overline{p}}; m1^p := r^p; \mathtt{prg}(P)$$
$$\mathtt{prg}(a.P) = m2^+ := m1^+; \mathtt{offer}(a, \langle m1^+, m1^- \rangle, \mathtt{prg}(P))$$
$$\mathtt{prg}(!a.P) = m2^+ := m1^+; \mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{fork}(\mathtt{offer}(a, \langle m1^+, m1^- \rangle, \mathtt{prg}(P)), \mathtt{skip})$$
$$\mathtt{prg}((\nu a)P) = \mathtt{new}\ a; \mathtt{prg}(P)$$
$$\mathtt{prg}((\nu r)P) = \mathtt{new}\ r^+; \mathtt{new}\ r^-; \mathtt{prg}(P)$$
$$\mathtt{prg}(P|Q) = \mathtt{fork}(\mathtt{prg}(P), \mathtt{prg}(Q))$$
$$\mathtt{prg}(\overline{v}.Q) = m2^+ := m1^+; \mathtt{invoke}(v, \langle m1^-, m1^+ \rangle, \mathtt{prg}(Q))$$
$$\mathtt{prg}(\mathtt{if}\ v = w\ \mathtt{then}\ P\ \mathtt{else}\ Q) = \mathtt{if}\ v = w\ \mathtt{then}\ \mathtt{prg}(P)\ \mathtt{else}\ \mathtt{prg}(Q)$$
$$\mathtt{prg}(\Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i) = \mathtt{in}(m1^-, \Sigma_{i \in \{1\ldots n\}}(\tilde{\mathfrak{F}}_i.\mathtt{prg}(P_i)))$$
$$\mathtt{prg}(\langle \tilde{e} \rangle) = \mathtt{out}(m1^+, \tilde{e})$$
$$\mathtt{prg}(\langle \tilde{e} \rangle^{\uparrow}.P) = \mathtt{fork}(\mathtt{out}(m2^+, \tilde{e}), \mathtt{prg}(P))$$
$$\mathtt{prg}(!\Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i) = \mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{in}(m1^-, \Sigma_{i \in \{1\ldots n\}}(\tilde{\mathfrak{F}}_i.\mathtt{fork}(\mathtt{prg}(P_i), \mathtt{skip})))$$
$$\mathtt{prg}(P > \Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.P_i) = m2^- := m1^-; \mathtt{fork}(\langle m1^+, \bullet \rangle, \mathtt{prg}(P), \langle m1^-, \bullet \rangle,$$
$$\qquad \mathtt{while}\ \mathit{true}\ \mathtt{do}\ \mathtt{in}(m1^-, (\Sigma_{i=1}^n \tilde{\mathfrak{F}}_i.\mathtt{fork}(m1^- := m2^-; \mathtt{prg}(P_i), \mathtt{skip}))))$$

Fig. 10. Translation of C SP S in SOAM

in $\tilde{x}$. As a general observation notice that replication is simulated by the machine in such a way that a new thread is created only after the previous one is consumed. The function prg is similar to net but, since it cannot access memory directly, it uses explicit assignments. The only subtle case is the prg-encoding of a pipe which uses the `fork` feature to synchronize and communicate. The $\bullet$ indicates that the variable it replaces is not used by the program, and thus that communication is unidirectional. After the synchronization $m1^+$ is bound in P to a new queue and used by P to output and the same queue is used in Q but bound in $m1^-$. After a new copy of Q is created the previous input channel stored in $m2^-$ is replaced in place of $m1^-$.

The key idea for correspondence results is to prove a basic proposition which relates the encoding of $\mathbb{C}[\![P]\!]$ with the separate encoding of $\mathbb{C}[\![0]\!]$ and P. Moreover, the rule (B) is not directly simulated by the machine which gives raise to an overflow in a real implementation, than the final result of completeness asserts that the machine is able to simulate the session nesting structure of a process.

5 ORC

In this section we show O [18] can be implemented over SOAM. In this sections we also introduce the SOAM primitives for handling unexpected behavior and for closing allocated queues. The syntax of SOAM programs presented in Fig. 1 is extended with the following commands: $\texttt{try}\ \mathcal{C}\ \texttt{with}\ \Sigma_{j\in J}(\tilde{\mathcal{F}}_j;\mathcal{C}_j)$, $\texttt{raise}\ \tilde{e}$ and $\texttt{dispose}\ x;\mathcal{C}$ These new primitives permit defining *controlled* execution blocks where exceptions can be raised by means of command $\texttt{raise}\ \tilde{e}$. Each block is equipped with a set of guarded inputs (a sort of catch blocks) that are used for handling raised exceptions. These inputs are used for selecting the exception handler to activate. Finally, command $\texttt{dispose}\ x$; can be used for deallocating existing queues. Syntax of SOAM networks (Fig. 3) is consequently extended with *try block* $\{\mathcal{N}\}_{(\sigma,\Sigma_{i\in I}(\tilde{\mathcal{F}}_i;\mathcal{C}_i))}$ and the nets executed within a *try-block* are terminated as soon as an exception is raised.

The operational semantics of new commands is defined as follows:

(M)
$$\langle\sigma \vdash \texttt{try}\ \mathcal{C}\ \texttt{with}\ \Sigma_{i\in J}(\tilde{\mathcal{F}}_i;\mathcal{C}_i)\rangle \rightarrow \{\langle\sigma \vdash \mathcal{C}\rangle\}_{(\sigma,\Sigma_{i\in I}(\tilde{\mathcal{F}}_i;\mathcal{C}_i))}$$

(M)
$$\frac{\langle\sigma \vdash x\rangle \rightarrow r}{\langle\sigma \vdash \texttt{dispose}\ x;\mathcal{C}\rangle|r:h \rightarrow \langle\sigma \vdash \mathcal{C}\rangle}$$

(M)
$$\frac{\mathcal{N} \not\equiv \mathcal{N}'|r:h \quad \mathsf{match}(\tilde{\mathcal{F}}_i,\tilde{v}) = \rho}{\{\langle\sigma \vdash \texttt{raise}\ \tilde{v}\rangle|\mathcal{N}\}_{(\sigma',\Sigma_{j\in J}(\tilde{\mathcal{F}}_j;\mathcal{C}_j))} \rightarrow \langle\sigma'\rho_{|L} \vdash \mathcal{C}_i\rho_{|R}\rangle}$$

Notice that (M) is easy to implement. Indeed, each network in a try-block can be thought of as a thread waiting for a given signal. Semantics of (M) guarantees that queues are not killed. Rule (M) states that, when command $\texttt{dispose}\ x$; is executed, the queue referenced by x is removed. The structural equivalence of SOAM nets is extended with the following cases:

$$r : h|\{\mathcal{N}\}_{(\sigma,\Sigma_{i\in I}(\tilde{\mathcal{F}}_i;\mathcal{C}_i))} \equiv \{\mathcal{N}|r : h\}_{(\sigma,\Sigma_{i\in I}(\tilde{\mathcal{F}}_i;\mathcal{C}_i))}$$
$$(\nu n)\{\mathcal{N}\}_{(\sigma,\Sigma_{i\in I}(\tilde{\mathcal{F}}_i;\mathcal{C}_i))} \equiv \{(\nu n)\mathcal{N}\}_{(\sigma,\Sigma_{i\in I}(\tilde{\mathcal{F}}_i;\mathcal{C}_i)))} \quad n \notin \mathsf{fn}(\sigma) \cup \mathsf{fn}(\mathcal{F}_i) \cup \mathsf{fn}(\mathcal{C}_i)$$

The check of free names in the second rule is necessary since `raise` communicate a name. We now give a brief overview of O . O is centered on the idea of service orchestration, and it assumes that basic services, able to perform computations, are available on primitive *sites*. O concentrates on invoking and orchestrating those services to reach some goal. Services may *publish* streams of values.

O 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. An O expression can be either a site call, an expression call or a composition of expressions according to one of the three basic orchestration patterns. The term 0 denotes the inactive O term. 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). The composition $P|Q$ executes both P and Q concurrently (differently from the original proposal, here we assume parallel and 0 define a commutative monoid), assuming that there is no interaction between them. It publishes the interleaving of the two streams of values published by P and Q. The composition $P > u > Q$ executes P, and, for each value v returned by P, it spawns a new instance of Q with v assigned to u. The asymmetric parallel composition Q **where** $x :\in P$ starts in parallel both P and the parts of Q that do not need x. When P publishes a value, say v, the continuation of P is canceled and v is assigned to x.

Now, we give a detailed description of the most interesting cases of O encoding:

$p_1(p_2)$**:** a call to a site is translated into a service invocation and relies on macro `call` defined as follows:

$$\mathtt{call}(a, v, o, b) = \begin{cases} \mathtt{invoke}(a, \langle x, y\rangle, \mathtt{out}(x, v); \mathtt{in}(y, ?w); \mathtt{out}(o, w)) & (b = true) \\ \mathtt{invoke}(a, \langle x, y\rangle, \mathtt{out}(x, v); \mathtt{in}(y, ?w); \mathtt{raise}\ w) & (b = false) \end{cases}$$

This macro implements a *request-response* interaction with a remote service. Notice that if b is true, the result of the invocation is stored in the queue o. Otherwise, the obtained value is raised.

$a(x) = f(x)$**:** each site definition is mapped to a service definition. The service associated to site a retrieves a value w from the first queue and then sends the result of $f(w)$ on the second queue.

$P > x > Q$**:** a private queue o' is created, where $\mathsf{net}(P, \sigma, true, o')$ will send all computed values. Values stored in queue o' are retrieved by a process that will instantiate a copy of $\mathsf{prg}(Q, b, o)$ where a new queue is created and associated to variable x. This task is performed by the following macro:
$\mathtt{wait}(o, x, \mathcal{C}) = \mathtt{while}\ true\ \mathtt{do}\ \mathtt{in}(o, ?w); \mathtt{fork}(\mathtt{new}\ x; \mathtt{out}(x, w); \mathcal{C}, \mathtt{skip})$

Q **where** $x :\in P$**:** a new queue is created and associated to x. This queue is shared between the nets associated to P and Q. Process P is executed within a *try-catch* block and returns a value via the `raise` command, the returned value will be stored in the queue associated to x.

6 Conclusions

We have defined a service oriented abstract machines, equipped with a formal structural semantics, that can be used to implement the service specification formalisms. Other than the usual constructs for imperative programming (if-then-else, while, assignment, sequencing), the machine has primitives for connecting and communicating over a network and to discover new resources in the network. Dedicated queues guarantee that, on service invocation, a persistent and protected, communication can be established to mirror the concept of session that is proper of service centered calculi.

We have used the proposed machine to implement three very different formalisms for service specification namely the session language (SL) of [12], C SP S, and O . For all of them we have proved that the proposed implementation is operationally correct (sound and complete). This fact is important if we consider that many useful properties (progress, non-interference, ...) can be established by using formal tool such as session types, see e.g. [11, 3, 8] and reused for free, because thanks to correspondence results we have that such properties are preserved by the encodings in SOAM.

The actual implementations of the three different formalisms in SOAM have helped us in better understanding their nature, and in appreciating their differences and their interrelationships. This could be instrumental for reconsidering some design choices that we have made for C SP S (the fact the we have implicit top level sessions) or the the fact that the SL with polarities is more suitable for an implementation w.r.t the original proposal with runtime checking of linearity.

As future work, we plan to investigate the extensions that are needed to deal with more advanced features of service oriented computing such as controlled service closures, compensations and multiparty synchronization.

Moreover, we plan to provide a complete prototype implementation of our machine, possibly exploiting the IMC framework described in [1] (see also [2]) so that we can conduct full fledged experiments.

References

1. L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, L. M. B. Lopes, L. Oliveira, H. Paulino, and V. T. Vasconcelos. A software framework for rapid prototyping of run-time systems for mobile calculi. In Corrado Priami and Paola Quaglia, editors, *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*, pages 179–207. Springer, 2005.
2. L. Bettini, R. De Nicola, and M. Loreti. Implementing session centered calculi. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2008.
3. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence assertions for process synchronization in concurrent communications. *J. Funct. Program.*, 15(2):219–247, 2005.
4. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V.T. Vasconcelos, and G. Zavattaro. SCC: a service centered calculus. In *Web Services and Formal Methods, Third International Workshop, WS-FM 2006*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

5. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *Proc. of FMOODS'08*, volume 5051 of *Lecture Notes in Computer Science*. Springer, 2008.
6. R. Bruni, I. Lanese, H. Melgratti, L.G. Mezzina, and E. Tuosto. Towards trustworthy multiparty sessions (extended abstract). In V.T. Vasconcelos and N. Yoshida, editors, *Pre-proceedings of PLACES 2008, 1st Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*, pages 22–27, 2008. Technical Report DI-FCUL TR-08-14, Departamento de Informatica, Faculdade de Ciencias da Universidade de Lisboa.
7. R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty sessions in SOC. In D. Lea and G. Zavattaro, editors, *Proceedings of COORDINATION 2008, 10th International Conference on Coordination Models and Languages*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 67–82. Springer, 2008.
8. R. Bruni and L.G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In G. Rosu and J. Meseguer, editors, *Proceedings of AMAST'08, 12th International Conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lect. Notes in Comput. Sci.*, pages 100–115. Springer, 2008.
9. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Sock: a calculus for service oriented computing. In *Proc. of 4th Int. Conf. on Service-Oriented Computing*, volume 4294 of *Lect. Notes in Comput. Sci.*, pages 327–338. Springer, 2006.
10. L. Caires, H.T. Viera, and J.C. Seco. The conversation calculus: a model of service oriented computation. Technical Report TR DIFCTUNL 6/07, Univ. Lisbon, 2007.
11. M. Dezani-Ciancaglini, de' Liguoro U., and U. Yoshida. On progress for structured communications. In *TGC'07*, volume 4912 of *Lect. Notes in Comput. Sci.*, pages 257–275. Springer, 2008.
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. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of POPL'08, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284. ACM, 2008.
14. N. Kobayashi. Typical: Type-based static analyzer for the pi-calculus. Tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`.
15. I. Lanese, F. Martins, A. Ravara, and V.T. Vasconcelos. Disciplining orchestration and conversation in service-oriented computing. In *SEFM '07*, pages 305–314. IEEE Computer Society Press, 2007.
16. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *ESOP '07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 33–47. Springer, 2007.
17. L. G. Mezzina. How to infer finite session types in a calculus of services and sessions. In *COORDINATION'08*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 216–231. Springer, 2008.
18. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 6(1):83–110, 2007.
19. F. Peschanski and S. Hym. A stackless runtime environment for a pi-calculus. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 57–67, New York, NY, USA, 2006. ACM.
20. B. C. Pierce. Programming in the pi-calculus: A tutorial introduction to pict. available electronically. Technical report, 1997.
21. A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, February 2006.

22. David N. Turner and Ph. D. The polymorphic pi-calculus: Theory and implementation. Technical report, 1995.
23. H. Vieira and L. Caires. The spatial logic model checker users manual. Technical report, 2004.
24. N. Yoshida and V.T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Elect. Notes in Th. Comput. Sci.*, 171(4):73–93, 2007.