Disciplining Orchestration and Conversation in Service-Oriented Computing

Ivan Lanese
Computer Science Department
University of Bologna
Mura Anteo Zamboni, 7, 40127 Bologna, Italy
lanese@cs.unibo.it

Vasco T. Vasconcelos
Department of Informatics, Faculty of Sciences
University of Lisbon,
Campo Grande, 1749–016 Lisbon, Portugal
vv@di.fc.ul.pt

Francisco Martins
Department of Informatics, Faculty of Sciences
University of Lisbon,
Campo Grande, 1749–016 Lisbon, Portugal
fmartins@di.fc.ul.pt

António Ravara
SQIG at IT, and Dep. of Mathematics, IST,
Technical University of Lisbon,
Av. Rovisco Pais 1, 1049–001 Lisbon, Portugal
aravara@ist.utl.pt

Abstract

We give a formal account of a calculus for modeling service-based systems, suitable to describe both service composition (orchestration) and the protocol that services run when invoked (conversation). The calculus includes primitives for defining and for invoking services, for isolating conversations between requesters and providers of services, and primitives for orchestrating services, that is, to make use of existent services as building blocks to accomplish complex tasks.

The calculus is equipped with a reduction and a labeled transition semantics; an equivalence result relates the two. To hint how the structuring mechanisms of the language can be exploited for static analysis we present a simple type system guaranteeing the compatibility between client and server protocols, an application of bisimilarity to prove equivalence among services, and we discuss deadlock-avoidance.

1 Introduction

Enterprise application integration, either to reuse legacy code, or to combine third-party software modules, has long been tackled by several middleware proposals, namely using message brokers or workflow management systems. As the popularity of using the Web to disseminate client-server applications increased, traditional middleware was forced to provide integration across companies over the Web. The technologies developed lay in the concept of *Web service*: a way of exposing (to the Web) the functionalities performed by internal systems and making it discoverable and accessible through the Web in a controlled manner [1]. Web services emerged as the main paradigm to program applications on the Web. An important reason is that currently available standards [2–4, 8, 12] allow to easily orchestrate different Web services (distributed and belonging to different organizations) to achieve required business goals. This paradigm allows to maximize interoperability, a crucial feature in current software systems development.

While standards and programming tools are continuously improving, the formal bases of Service Oriented Computing (SOC) are still uncertain: there is an urgent need for models and techniques allowing the development of applications in a safe manner, while checking that systems provide the required functionalities. These techniques should be able to deal with the different aspects of services (seen in the abstract context of global computing [10]), including their dynamic behavior.

Process calculi are recognized as an important tool to give precise semantics to system specifications, and they come equipped with a rich toolbox of analysis techniques (type systems, bisimulations, control flow analysis, ...). Nevertheless, how to use process calculi to model service oriented systems is not yet clear, since not all the involved concerns have been properly addressed. When defining a calculus for SOC, different aspects influence the choice of primitives and the definition of their behavior, and a careful trade-off between expressiveness and suitability to analysis should be found. Our main concerns have been the following ones.

Expressiveness of the language: the calculus should be able to express in a direct way the different kinds of interactions that characterize SOC: invocations of services, client-server conversations and interactions among different client-server pairs. We use three different classes of operators to express these interactions: service definition/invocation, sessions, and streams. We show via examples that these are enough to model various kinds of SOC scenarios. We stress in particular the importance of the third kind of interaction, which is the heart of orchestration, the most challenging aspect in SOC. Other constructs such as, e.g., tuple spaces or shared memory would be as expressive as streams, but would be difficult to analyze.

Expressiveness of the analysis: the elements to be analyzed should correspond to explicit elements in the calculus. Concerning the classes above, service definition is fundamental to speak about service availability. It also allows easy extensions for service discovery based on quality of service. Sessions instead allow to analyze client-server compatibility and to study behavioral-based service discovery. Other mechanisms, such as BPEL correlation sets [2], would make these analyses more complex, since they rely on runtime values for determining the communication patterns, spreading the protocol code throughout the whole program.

Computability of the analysis: static analysis should be decidable, possibly also efficient to compute. For this reason the allowed communication patterns should be constrained whenever this does not destroy expressiveness. In our calculus streams and sessions are static, and the dynamism is concentrated in service invocation. To stress the effect of these considerations on the design decisions, we give some "proof of concept" analyses to illustrate how to exploit the features of the calculus, analyses to be further developed in future works.

This paper proposes SSCC (Stream-based Service Centered Calculus), a calculus for modeling service-based systems, inspired by SCC [5] and Orc [14, 17], and developed with the above considerations in mind.

We introduce SSCC after having tried to use SCC and failed. While proposing interesting concepts, like sessions, and featuring services as first class entities, SCC looks not fully adequate (at least in the form presented in [5]) for service composition. In fact the only way for a session to interact with other client-server pairs is the **return** primitive, and the functional style of invocation is not adequate for modeling complex patterns of interactions such as van der Aalst workflow patterns [21]. To overcome these problems we introduced streams and we allowed non persistent service invocations. This enhances the expressiveness of the calculus and makes it easier to program.

Another source of inspiration was Orc [17], a basic programming model for structured orchestration of Web services. Here a few coordination constructs are used to model the most common patterns, and a satisfying expressiveness is claimed by presenting a formalization of all van der Aalst workflow patterns [9]. However, in order to model the more challenging patterns, special sites (the basic computation entity in Orc) are required, acting e.g. as semaphores. This is a coordination concern, and in our opinion should be addressed within the coordination language (notice that Orc does not allow to program such a kind of site). Thus we introduced more basic mechanisms to tackle all the coordination concerns inside the calculus (notice that most of Orc operators can be expressed as macros in our model). Also, we introduced conversations, which are not present in Orc, to model service behavior (Orc leaves this unspecified). It is thus not possible to develop for Orc analysis techniques to ensure, e.g., deadlock freedom, as this would require analyzing the behavior of the sites involved.

Among the calculi, π-calculus (and its variants) has been frequently used in SOC. However we claim that general purpose concurrent calculi are not suitable for our aims, since the different communication patterns are mixed, and most of the interesting properties not reflected in the term. Thus these calculi do not satisfy the requirements above. Different proposals used types, e.g. session types [11, 13, 20, 23], to solve this problem, but since they allow free π-calculus communications the analysis is difficult. We consider our proposal as some kind of tamed π-calculus, with a good trade-off between expressiveness for SOC systems and suitability to analyze SOC-related properties.

Other calculi tailored for SOC exist, and we briefly compare with them.

Carbone et al. [7] aim at capturing the principles behind Web service based business processes. A global description of communication behavior needs to be complemented by an "endpoint-based" description of each participant to the protocol, a projection of the global scenario. We are at the same abstraction level of the endpoint calculus, but this one relies on shared memory and general communication mechanisms, making it more difficult to analyze.

Lapadula et al. introduce CŎWS [16], a process calculus for Web service orchestration. For isolating interactions between partners, CŎWS uses message correlation, the approach of WS-BPEL [2]. Our approach based on sessions is dual to this, and ensures more structured communication.

Busi et al. [6] propose SOCK, a process calculus inspired in Web services specifications. SOCK is composed by different layers, taking care of particular aspects such as service behavior, state, and interactions between different sessions of the same service. SOCK also uses message correlation to define client-server interactions. SOCK is quite complex since it closely follows current standards in SOC technologies, but we want to explore more in depth the semantic issues of SOC without the additional complexity needed to model industrial standards.

Proofs for the results presented herein, and an encoding of van der Aalst workflow patterns [21], can be found in a technical report [15].

2 A motivating example

We start with a simple process to deliver the price for a given date at a given hotel.

```
(date) <query-the-hotel-db>.price
```

Here, the parentheses in (date) indicate the reception of a value, and an identifier alone, as in price, means publishing a value. Hotel bologna may then turn this conversation into a service definition, by writing:

```
bologna ⇒ (date) <query-the-hotel-db>.price
```

A client is supposed to meet the expectations of the service by providing a date and requesting a price:

```
bologna ⇐ 31Jul2007.(price) <use-price>
```

When the service provider (⇒) and the service client (⇐) get together, by means, e.g., of parallel composition, a *conversation* takes place, and values are exchanged in both directions.

Now suppose that a broker comes to the market trying to provide better deals for its clients. The behavior of the broker is as follows: it asks prices to three hotels that it knows of, waits for two results, and publishes the best offer of the two. Calling the services for a given date is as above:

```
bologna ⇐ date.(price1) ... |
azores ⇐ date.(price2) ... |
lisbon ⇐ date.(price3) ...
```

In order to collect the prices for further processing, we introduce a *stream* constructor, playing the role of a *service orchestrator*. The various prices are fed into the stream; a different process reads the stream. We write it as follows.

```
stream
  bologna ⇐ date.(price1).feed price1 |
  azores ⇐ date.(price2).feed price2 |
  lisbon ⇐ date.(price3).feed price3
as f in
  f(x).f(y).<publish-the-min-of-x-and-y>
```

To write price1 into a stream we use the syntax **feed** price1. To read a value from stream f we use f(x).<use-x>. Writing is an anonymous operation (feeds to the nearest enclosing stream), whereas reading is named. The above pattern is so common that we provide a special syntax for it, inspired by Orc (the various abbreviations used in this paper are summarized in Figure 7).

```
(call bologna(date) |
 call azores(date) |
 call lisbon(date)) >2
   x y > <publish-the-min-of-x-and-y>
```

To complete the example we rely on a min service, chaining the first two answers, and publishing the result.

```
broker ⇒ (date).(
  (call bologna(date) |
   call azores(date) |
   call lisbon(date)) >2
     x y > call min(x,y) >1 m > m)
```

Notice that a client interacts with the broker as if it was interacting with a particular hotel. The downside is that the client does not know which hotel offers the best price; we leave it to the reader to adapt the example as required.

P, Q	$::=$	*Processes*
	$P \vert Q$	Parallel composition
	$\vert\ (\nu a)P$	Name restriction
	$\vert\ \mathbf{0}$	Terminated process
	$\vert\ X$	Process variable
	$\vert\ \mathsf{rec}\, X.P$	Recursive process definition
	$\vert\ a \Rightarrow P$	Service definition
	$\vert\ a \Leftarrow P$	Service invocation
	$\vert\ v.P$	Value sending
	$\vert\ (x)P$	Value reception
	$\vert\ \mathsf{stream}\, P\, \mathsf{as}\, f\, \mathsf{in}\, Q$	Stream
	$\vert\ \mathsf{feed}\, v.P$	Feed the process' stream
	$\vert\ f(x).P$	Read from a stream
u, v	$::=$	*Values*
	a	Service name
	$\vert\ \mathsf{unit}$	Unit value

Figure 1. The syntax of SSCC

Using **call** and P>$^{\mathsf{n}}$ x1...xn >Q we have avoided explicitly mentioning streams altogether. Direct stream manipulation can however be quite handy. The following example shows a broker that logs all three answers, *after* publishing the best price of the first two (cf. the Discriminator Pattern [21]).

```
stream ... as f in
  f(x).f(y).call min(x,y) >1
    m > (m | f(z).log ⇐ x.y.z)
```

Further examples can be found in Section 4, after presenting the syntax and the semantics of the calculus.

3 The SSCC calculus

This section presents the syntax and the operational semantics of SSCC.

Processes are built using three kinds of identifiers: *service names* ranged over by $a, b, x, y, \ldots$, *stream names* ranged over by $f, g, \ldots$, and *process variables* ranged over by $X, Y, \ldots$. The grammar in Figure 1 defines the *syntax of processes*.

The first five cases of the grammar introduce standard process calculi operators: parallel composition, restriction (only service names can be restricted), the terminated process, and recursion. We then have two constructs to *build services*: definition (or provider) and invocation (or client). Both are defined by their name a and protocol P. Notice that service definition and service invocation are symmet-

$$\begin{array}{lll}
P, Q & ::= & \textit{Processes} \\
 & \quad \ldots & \text{as in Figure 1} \\
 & \mid \; r \rhd P & \text{Server session} \\
 & \mid \; r \lhd P & \text{Client session} \\
 & \mid \; (\nu r)P & \text{Session restriction} \\
 & \mid \; \mathsf{stream}\, P \,\mathsf{as}\, f = \vec{v} \,\mathsf{in}\, Q & \text{Stream with values}
\end{array}$$

Figure 2. The run-time syntax of SSCC

ric (differently from [5]). *Service protocols* are built using value sending and receiving, allowing bidirectional communication between clients and servers. Finally there are the three constructs for *service orchestration*, which constitute the main novelty of SSCC. The stream construct declares a stream f for communication from P to Q. P can insert a value v into the stream using $\mathsf{feed}\, v.P'$, and Q can read from there using $f(x).Q'$. Notice that stream names cannot be communicated, thus they model static channels.

Processes at runtime exploit an extended syntax: the interaction of a service definition and a service invocation produces an active session. Also, values in the stream are stored together with the stream definition. We introduce a fourth kind of identifier: *session names*, use $r, s, \ldots$ to range over them, and use $n, m, \ldots$ to range over both session and service names. The grammar in Figure 2 defines the *syntax of runtime processes*.

We use $r \bowtie P$ to denote both $r \lhd P$ and $r \rhd P$, and we assume that when multiple $\bowtie$ appear in the same rule they are instantiated in the same way, and that if $\overline{\bowtie}$ appears too then it denotes the opposite instantiation. The constructor $\mathsf{stream}\, P \,\mathsf{as}\, f \;\mathsf{in}\, Q$ in Figure 1 is an abbreviation of $\mathsf{stream}\, P \,\mathsf{as}\, f = \langle\rangle \,\mathsf{in}\, Q$ in Figure 2.

Streams can be considered either ordered or unordered. An unordered stream is a multiset, while an ordered one is a queue. In most cases the difference is not important. We write $w :: \vec{v}$ for the stream obtained by adding w to $\vec{v}$, and $\vec{v} :: w$ for a stream from which w can be removed. In the latter case $\vec{v}$ is what we get after removing w. The semantics that we present can deal with both ordered and unordered streams, by just changing the definition of $' ::'$.

As for bindings, name x is bound in $(x)P$ and in $f(x).P$; name n is bound in $(\nu n)P$; stream f is bound in $\mathsf{stream}\, P \,\mathsf{as}\, f \;\mathsf{in}\, Q$ with scope Q; and process variable X is bound in $\mathsf{rec}\, X.P$. All bound identifiers are alpha-convertible. Notation $\mathrm{fn}(P)$ denotes the set of free (service or session) names in P. Similarly, $\mathrm{bn}(P)$ is the set of bound names. We require processes to have no free process variables.

As usual, to help the definition of the operational semantics, we use a structural congruence relation. The relation is standard, simply adding to that of the π-calculus axioms that deal with scope extrusion for sessions and streams.

$$\begin{array}{rcl}
\mathcal{C}[\![]\!] & ::= & \bullet \;\mid\; \mathcal{C}[\![]\!]|Q \;\mid\; P|\mathcal{C}[\![]\!] \\
 & & \mid\; (\nu n)\mathcal{C}[\![]\!] \;\mid\; \mathsf{stream}\, \mathcal{C}[\![]\!] \,\mathsf{as}\, f = \vec{v} \,\mathsf{in}\, Q \\
 & & \mid\; \mathsf{stream}\, P \,\mathsf{as}\, f = \vec{v} \,\mathsf{in}\, \mathcal{C}[\![]\!] \;\mid\; r \bowtie \mathcal{C}[\![]\!] \\
\mathcal{D}[\![,]\!] & ::= & \mathcal{C}'[\![]\!]|\mathcal{C}''[\![]\!] \;\mid\; \mathsf{stream}\, \mathcal{C}'[\![]\!] \,\mathsf{as}\, f = \vec{v} \,\mathsf{in}\, \mathcal{C}''[\![]\!]
\end{array}$$

Figure 4. Contexts

Definition 3.1 (Structural congruence) *The rules in Figure 3, together with the commutative monoid rules for* $(P, |, \mathbf{0})$*, inductively define the* structural congruence relation *on processes.*

Interactions can happen in different active contexts. Since all our interactions are binary, we introduce also two-holes contexts, which we call double contexts. The grammar in Figure 4 generates *active and double contexts*. Applying a double context to two processes P_1 and P_2 produces the process obtained by replacing the first (in the prefix visit of the syntax tree) hole $\bullet$ with P_1 and the second hole $\bullet$ with P_2.

Definition 3.2 (Reduction semantics) *The rules in Figure 5, together with symmetric rules of* R-COMM *and of* R-SYNC *(swapping the processes in the two holes of double context), inductively define the* reduction relation *on processes.*

Rule R-SYNC allows a service invocation to interact with a corresponding service definition. This interaction produces a pair of complementary sessions, distinguished by a fresh restricted name r. Notice that both the service invocation and the service definition disappear (in particular, service definition is not persistent as in SCC [5]). Rule R-COMM allows communication between corresponding sessions, created by the previous rule. Then there are the two rules dealing with streams: rule R-FEED puts a value in the stream while rule R-READ takes a value from the stream. Finally we have rule R-CONG that allows reduction to happen inside arbitrary active contexts, and rule R-STR for exploiting structural congruence.

The reduction semantics is intuitive, but one based on a labeled transition system (LTS, for short) is more convenient for some proofs and allows to exploit bisimulation-based techniques.

The rules are defined in the usual way, relating processes via *labels* that express interaction with the environment. We follow an *early* style.

Definition 3.3 (LTS semantics) *The rules in Figure 6, together with symmetric versions of rules* L-SESS-COM-STREAM *and* L-SERV-COM-STREAM*, inductively define the* labeled transition system *on processes.*

We highlight some aspects that may be less clear, explaining at the same time the labels used. We use μ as

$$
\begin{array}{rcll}
(\nu n)P|Q & \equiv & (\nu n)(P|Q) \quad \text{if } n \notin \mathrm{fn}(Q) \qquad r \bowtie (\nu a)P \equiv (\nu a)(r \bowtie P) & \text{(S-EXTR-PAR, S-EXTR-SESS)} \\
\mathsf{stream}\ (\nu a)P\ \mathsf{as}\ f = \vec{v}\ \mathsf{in}\ Q & \equiv & (\nu a)(\mathsf{stream}\ P\ \mathsf{as}\ f = \vec{v}\ \mathsf{in}\ Q) \quad \text{if } a \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{v}) & \text{(S-EXTR-STREAML)} \\
\mathsf{stream}\ P\ \mathsf{as}\ f = \vec{v}\ \mathsf{in}\ (\nu a)Q & \equiv & (\nu a)(\mathsf{stream}\ P\ \mathsf{as}\ f = \vec{v}\ \mathsf{in}\ Q) \quad \text{if } a \notin \mathrm{fn}(P) \cup \mathrm{Set}(\vec{v}) & \text{(S-EXTR-STREAMR)} \\
(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P & & (\nu a)\mathbf{0} \equiv \mathbf{0} \qquad \mathsf{rec}\ X.P \equiv P[{}^{\mathsf{rec}\ X.P}/_{X}] & \text{(S-SWAP, S-COLLECT, S-REC)}
\end{array}
$$

Figure 3. Structural congruence

$$
\frac{\mathcal{D}[\![,]\!] \text{ does not bind } r \text{ or } a \qquad r \notin \mathrm{fn}(P) \cup \mathrm{fn}(Q) \cup \mathrm{fn}(\mathcal{D}[\![,]\!])}{\mathcal{D}[\![a \Rightarrow P, a \Leftarrow Q]\!] \rightarrow (\nu r)\mathcal{D}[\![r \rhd P, r \lhd Q]\!]} \quad \text{(R-SYNC)}
$$

$$
\frac{\begin{array}{c}\mathcal{D}[\![,]\!], \mathcal{C}[\![]\!], \text{ and } \mathcal{C}'[\![]\!] \text{ do not bind } r \text{ or } v \\ \mathcal{C}[\![]\!] \text{ and } \mathcal{C}'[\![]\!] \text{ do not contain sessions around the } \bullet\end{array}}{(\nu r)\mathcal{D}[\![r \bowtie \mathcal{C}[\![v.P]\!], r\overline{\bowtie}\mathcal{C}'[\![(x)Q]\!]]\!] \rightarrow (\nu r)\mathcal{D}[\![r \bowtie \mathcal{C}[\![P]\!], r\overline{\bowtie}\mathcal{C}'[\![Q[{}^{v}/_{x}]]\!]]\!]} \quad \text{(R-COMM)}
$$

$$
\frac{\mathcal{C}[\![]\!] \text{ does not bind } w \qquad \bullet \text{ does not occur in the left part of a stream context}}{\mathsf{stream}\ \mathcal{C}[\![\mathsf{feed}\ w.P]\!]\ \mathsf{as}\ f = \vec{v}\ \mathsf{in}\ Q \rightarrow \mathsf{stream}\ \mathcal{C}[\![P]\!]\ \mathsf{as}\ f = w :: \vec{v}\ \mathsf{in}\ Q} \quad \text{(R-FEED)}
$$

$$
\frac{\mathcal{C}[\![]\!] \text{ does not bind } w \text{ or } f}{\mathsf{stream}\ P\ \mathsf{as}\ f = \vec{v} :: w\ \mathsf{in}\ \mathcal{C}[\![f(x).Q]\!] \rightarrow \mathsf{stream}\ P\ \mathsf{as}\ f = \vec{v}\ \mathsf{in}\ \mathcal{C}[\![Q[{}^{w}/_{x}]]\!]} \quad \text{(R-READ)}
$$

$$
\frac{P \rightarrow P'}{\mathcal{C}[\![P]\!] \rightarrow \mathcal{C}[\![P']\!]} \qquad \frac{Q \equiv P \rightarrow P' \equiv Q'}{Q \rightarrow Q'} \quad \text{(R-CONG, R-STR)}
$$

Figure 5. Reduction relation

metavariable for labels. Label $\uparrow v$ denotes the output of value v. Dually, $\downarrow v$ is the input of value v. We use $\updownarrow v$ to denote either $\uparrow v$ or $\downarrow v$. Also, $a \Rightarrow (r)$ and $a \Leftarrow (r)$ denote respectively the invocation and the reception of an invocation of a service a. Here r is the name of the new session to be created and it is bound. Also, $\Uparrow v$ denotes the feeding of v to a stream, while $f \Downarrow v$ is the read of value v from stream f. Notice that the value taken in input in rules L-RECEIVE and L-READ is guessed: this is an early semantics. When an input or an output label crosses a session construct (rule L-SESS-VAL), we have to add to the label its name and whether it is a server or client session (for example $\downarrow v$ may become $r \lhd \downarrow v$). This is useful in the development of the type system. Notice that we can have two contexts causing interaction: parallel composition and stream. The label denoting a conversation step in a free session r is $r\tau$, and a label τ is obtained only when r is restricted (rule L-SESS-RES). Thus a τ action can be obtained in four cases: a communication inside a restricted session, a service invocation, a feed or a read from a stream. Finally, bound actions $(a)\mu$ are like the respective free counterparts μ, but here a is extruded. There is no need to deal explicitly with these actions since, if the interaction is internal to the system, structural congruence can be used to broaden the scope of a.

We conclude this section with a theorem relating the reduction and the LTS semantics.

Theorem 3.1 (Correspondence theorem) *For each P and Q, $P \rightarrow Q$ if and only if $P \xrightarrow{\tau} Q$.*

$$
\begin{array}{rcl}
\textbf{call}\ \mathsf{a(x1\ ,...,\ xn)} & \triangleq & \mathsf{a} \Leftarrow \mathsf{x1...xn.(y)}\ \textbf{feed}\ \mathsf{y} \\
\mathsf{P} >^{\mathbf{n}} \mathsf{x1\ ...\ xn} > \mathsf{Q} & \triangleq & \textbf{stream}\ \mathsf{P}\ \textbf{as}\ \mathsf{f}\ \textbf{in}\ \mathsf{f(x1\ )...\ f(xn)Q} \\
\mathsf{P} > \mathsf{x} > \mathsf{Q} & \triangleq & \textbf{stream}\ \mathsf{P}\ \textbf{as}\ \mathsf{f}\ \textbf{in}\ \textbf{rec}\ \mathsf{X.f(x)(P\ |\ X)} \\
\mathsf{a} * \Rightarrow \mathsf{P} & \triangleq & \textbf{rec}\ \mathsf{X.\ a} \Rightarrow \mathsf{(P\ |\ X)}
\end{array}
$$

Figure 7. Derived constructs

4 Further examples

This section explores examples that highlight the versatility of SSCC. We start by discussing a few macros that speed up modeling and suggest how Orc can be mapped in SSCC. They are listed in Figure 7. The first one invokes an activity (a service which gives back one result) a with a tuple of parameters and makes the result available via a feed. The second macro allows to model sequential composition, with parameter passing. The third one more closely models Orc sequential composition, since an instance of Q is executed for each value received from P. The last macro allows to define permanent services.

Example 4.1 (Fork-join) *The first example shows that naming streams can be handy.* Fork-join *is a pattern that*

$$v.P \xrightarrow{\uparrow v} P \qquad (x)P \xrightarrow{\downarrow v} P[{}^{v}\!/\!_{x}] \qquad \mathsf{feed}\, v.P \xrightarrow{\Uparrow v} P \qquad f(x).P \xrightarrow{f \Downarrow v} P[{}^{v}\!/\!_{x}]$$

(L-SEND, L-RECEIVE, L-FEED, L-READ)

$$\frac{r \notin \mathrm{fn}(P)}{a \Leftarrow P \xrightarrow{a \Leftarrow (r)} r \lhd P} \qquad \frac{r \notin \mathrm{fn}(P)}{a \Rightarrow P \xrightarrow{a \Rightarrow (r)} r \rhd P}$$

(L-CALL, L-DEF)

$$\frac{P \xrightarrow{\mu} P' \quad \mu \neq \Uparrow v \quad \mathrm{bn}(\mu) \cap (\mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})) = \emptyset}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\mu} \mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q} \qquad \frac{Q \xrightarrow{\mu} Q' \quad \mu \neq f \Downarrow v \quad \mathrm{bn}(\mu) \cap (\mathrm{fn}(P) \cup \mathrm{Set}(\vec{w})) = \emptyset}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\mu} \mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'}$$

(L-STREAM-PASS-P, L-STREAM-PASS-Q)

$$\frac{P \xrightarrow{\Uparrow v} P'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\tau} \mathsf{stream}\, P'\, \mathsf{as}\, f = v :: \vec{w}\, \mathsf{in}\, Q} \qquad \frac{Q \xrightarrow{f \Downarrow v} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w} :: v\, \mathsf{in}\, Q \xrightarrow{\tau} \mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'}$$

(L-STREAM-FEED, L-STREAM-CONS)

$$\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P|Q \xrightarrow{\mu} P'|Q} \qquad \frac{P \xrightarrow{\updownarrow v} P'}{r \bowtie P \xrightarrow{r \bowtie \updownarrow v} r \bowtie P'} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \neq \updownarrow v \quad r \notin \mathrm{bn}(\mu)}{r \bowtie P \xrightarrow{\mu} r \bowtie P'}$$

(L-PAR, L-SESS-VAL, L-SESS-PASS)

$$\frac{P \xrightarrow{r \bowtie \uparrow v} P' \quad Q \xrightarrow{r \overline{\bowtie} \downarrow v} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{r\tau} \mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \qquad \frac{P \xrightarrow{a \Rightarrow (r)} P' \quad Q \xrightarrow{a \Leftarrow (r)} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\tau} (\nu r)\mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'}$$

(L-SESS-COM-STREAM,L-SERV-COM-STREAM)

$$\frac{P \xrightarrow{r \bowtie \uparrow v} P' \quad Q \xrightarrow{r \overline{\bowtie} \downarrow v} Q'}{P|Q \xrightarrow{r\tau} P'|Q'} \qquad \frac{P \xrightarrow{a \Rightarrow (r)} P' \quad Q \xrightarrow{a \Leftarrow (r)} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')} \qquad \frac{P \xrightarrow{\mu} P',\ P \equiv Q,\ P' \equiv Q'}{Q \xrightarrow{\mu} Q'}$$

(L-SESS-COM-PAR, L-SERV-COM-PAR, L-STRUCT)

$$\frac{P \xrightarrow{\mu} P' \quad n \notin \mathrm{n}(\mu)}{(\nu n)P \xrightarrow{\mu} (\nu n)P'} \qquad \frac{P \xrightarrow{r\tau} P'}{(\nu r)P \xrightarrow{\tau} (\nu r)P'} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \in \{\uparrow a, r \bowtie \uparrow a, \Uparrow a\}}{(\nu a)P \xrightarrow{(a)\mu} P'}$$

(L-RES,L-SESS-RES,L-EXTR)

Figure 6. Labeled transition system

spawns two threads, and resumes computation after receiving a value from each thread. In the example below, services a *and* b *are run in parallel;* **call** a *feeds the first result produced by the service into stream* f*, and similarly for* **call** b *and stream* g.

```
fork−and−join ⇛ (a)(b)(
  stream call a as f in
    stream call b as g in
      f(x).g(y).x.y)
```

The example is inspired by Orc *[14, 17]* **where***, but here we do not kill service invocations* a *and* b*, instead let them run to completion.* Orc *is not able to match our semantics: reading a single value from an expression can only be performed via the* **where** *construct, and that necessarily means terminating the evaluation of the expression. We feel that termination should be distinct from normal orchestration; we leave for further work termination (and the corresponding compensation).*

It is difficult to model the same pattern in SCC *too, since the two clients should use* **return** *to make their result available, but the two values would be mixed. Auxiliary services are required to match this semantics.*

Example 4.2 (Memory cell) *Even if a memory cell is not a common scenario in SOC, stateful services are. Examples abound in the literature, from data-structures to weblog update [5]. Contrary to* SCC *[5], our language allows writing stateful services without exploiting service termination. Inspired in the encoding of objects in the pi-calculus [19], we set up a simple, ephemeral, service to produce a value:* buffer ⇒ v*. Service* get *calls the buffer service to obtain its value (thus consuming the service provider), replies the value to the client, and replaces the buffer service.*

```
get ⇛ call buffer >¹ v > (v | buffer ⇒ v)
```

Service set *calls the buffer service (in order to consume the service provider), then gets the new value from the client and replaces the buffer with this value.*

```
set ⇛ call buffer >¹> (w)(buffer ⇒ w)
```

Finally, the cell *service sets up three services—*get, set, *and* buffer*—sends the first two to the client, and keeps* buffer *locally with initial value* 0.

```
cell ⇛ (ν buffer, get, set).get.set.
 (buffer ⇒ 0 |
  get ⇛ call buffer >¹ v >
```

```
      (v | buffer ⇒ v) |
   set *⇒ call buffer >¹> (w)(buffer ⇒ w))
```

Example 4.3 (Interleaved parallel routing) *The workflow patterns of van der Aalst [21] provide a well-known benchmark of orchestration scenarios. Even if these are aimed at workflow languages (and thus, e.g., do not consider conversations or dynamic creation of services), it is interesting to look at them. All the patterns that do not require killing ongoing computations can be implemented. See [15] for a complete description.*

In interleaved parallel routing *workflow pattern, a set of activities is executed in arbitrary order, and no two activities are executed at the same moment.*

*We assume that each service (*a1 *to* an*) signals termination by sending a value.*

Contrary to Orc [9], SSCC is expressive enough to describe the pattern within the language. This requires a backward communication w.r.t. the direction of the stream, and shows that unidirectional streams are expressive enough. A back *service relays the values from the right to the left part of a stream construct, where they are fed into the stream.*

```
interleave ⇒ (a1)...(an)(ν back)(
  stream
    back *⇒ (x)feed x
  as lock in
    back ⇐ unit |
    lock(_).a1 ⇐ (x)(back ⇐ unit) | ... |
    lock(_).an ⇐ (x)(back ⇐ unit))
```

Example 4.4 (Complex protocols) *One of the main limitations of other proposals, e.g. Orc, is that they allow just very simple kinds of client-server interactions. We show here how sessions can be used to overcome this limitation.*

Let us consider an hotel booking: the client sends the dates and the type of room to the hotel, the hotel answers with the price (we skip many details of a real protocol for space constraints), the client provides a credit card, the hotel checks with the bank that the required amount of money is available and sends to the client a confirmation. We show for simplicity just the hotel server, and we suppose to have a server for the bank and a server pricetable *to compute the price of the staying.*

```
hotel ⇒ (date)(room)
  call pricetable(date,room) >¹ price >
  price. (cc) call bank(price,cc) >¹ avail >
  if avail then confirm else reject
```

The if-then-else construct can be defined as a macro in the language (see [15]).

5 Protocol compatibility

We present a simple type system to ensure protocol compatibility between clients and servers, inspired by works on session types [11, 13, 20, 23]. Notice that here we can deal with many interacting services at the same time.

T ::=		*Types*
	Unit	unit type
	$\mid\ [U]$	service type
U ::=		*Conversation types*
	$?T.U$	input
	$\mid\ !T.U$	output
	$\mid$ end	end of conversation
	$\mid\ X$	type variable
	$\mid$ rec $X.U$	recursive type

Figure 8. The syntax of types

$$\overline{?T.U} \triangleq !T.\overline{U} \qquad \overline{!T.U} \triangleq ?T.\overline{U} \qquad \overline{\mathsf{end}} \triangleq \mathsf{end}$$
$$\overline{X} \triangleq X \qquad \overline{\mathsf{rec}\ X.U} \triangleq \mathsf{rec}\ X.\overline{U}$$

Figure 9. Complement of a protocol

Definition 5.1 (Types) *The grammar in Figure 8 defines the* syntax of types.

Types for values, T, are either Unit, which denotes the only basic type[1], and $[U]$ is the type of a service (and of a session) with protocol U. The protocol is always seen from the server point of view. *Types for streams* are of the form $\langle T\rangle$ where T is the type of the values the stream carries. *Types for processes* are of the form (U,T) where U is the protocol followed by the process, and T is the type of the values the process feeds into its stream.

The rec operator for types is a binder, giving rise, in the standard way, to notions of bound and free variables and alpha-equivalence. Similarly to processes, we do not distinguish between alpha-convertible types. Furthermore, we take an *equi-recursive* view of types [18], not distinguishing between a type rec $X.U$ and its unfolding $T[^{\mathsf{rec}\ X.U}/_X]$. We are interested on *contractive* (not including subterms of the form rec X.rec $X_1 \ldots$ rec $X_n.X$) types only [18].

We need to find whether two protocols are complementary, thus we introduce the complement operation in Figure 9. Intuitively, if a client executes protocol U and a server protocol $\overline{U}$, the conversation between them can proceed without errors.

Typing judgments are as follows,

$$\Gamma \vdash P\colon (U,T) \qquad\qquad \textit{Processes}$$
$$\Gamma \vdash v\colon T \qquad\qquad \textit{Values}$$

where Γ is a map with entries $a\colon T$, $r\colon T$, $f\colon \langle T\rangle$, and $X\colon (U,T)$. The rules in Figure 10 inductively define the *type system*.

[1]To be possibly extended with, say, integers and strings.

$$\Gamma, n\colon T \vdash n\colon T \qquad \Gamma, f\colon \langle T\rangle \vdash f\colon \langle T\rangle \qquad \Gamma \vdash \mathsf{unit}\colon \mathsf{Unit} \qquad \text{(T-NAME, T-SNAME, T-UNIT)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash v\colon T'}{\Gamma \vdash v.P\colon (!T'.U,T)} \qquad \frac{\Gamma, x\colon T' \vdash P\colon (U,T)}{\Gamma \vdash (x)P\colon (?T'.U,T)} \qquad \text{(T-SEND, T-RECEIVE)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash a\colon [U]}{\Gamma \vdash a \Rightarrow P\colon (\mathsf{end},T)} \qquad \frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash a\colon [\overline{U}]}{\Gamma \vdash a \Leftarrow P\colon (\mathsf{end},T)} \qquad \text{(T-DEF, T-CALL)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash r\colon [U]}{\Gamma \vdash r \rhd P\colon (\mathsf{end},T)} \qquad \frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash r\colon [\overline{U}]}{\Gamma \vdash r \lhd P\colon (\mathsf{end},T)} \qquad \text{(T-SESS-S, T-SESS-C)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash v\colon T}{\Gamma \vdash \mathsf{feed}\, v.P\colon (U,T)} \qquad \frac{\Gamma, x\colon T \vdash P\colon (U,T') \quad \Gamma \vdash f\colon \langle T\rangle}{\Gamma \vdash f(x).P\colon (U,T')} \qquad \text{(T-FEED, T-READ)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash Q\colon (\mathsf{end},T)}{\Gamma \vdash P|Q\colon (U,T)} \qquad \frac{\Gamma \vdash P\colon (\mathsf{end},T) \quad \Gamma \vdash Q\colon (U,T)}{\Gamma \vdash P|Q\colon (U,T)} \qquad \text{(T-PAR-L, T-PAR-R)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma, f\colon \langle T\rangle \vdash Q\colon (\mathsf{end},T') \quad w \in \mathrm{Set}(\vec{v}) \Rightarrow \Gamma \vdash w\colon T}{\Gamma \vdash \mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q\colon (U,T')} \qquad \text{(T-STREAM-L)}$$

$$\frac{\Gamma \vdash P\colon (\mathsf{end},T) \quad \Gamma, f\colon \langle T\rangle \vdash Q\colon (U,T') \quad w \in \mathrm{Set}(\vec{v}) \Rightarrow \Gamma \vdash w\colon T}{\Gamma \vdash \mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q\colon (U,T')} \qquad \text{(T-STREAM-R)}$$

$$\frac{\Gamma, X\colon (U,T) \vdash P\colon (U,T)}{\Gamma \vdash \mathsf{rec}\, X.P\colon (U,T)} \qquad \frac{\Gamma, n\colon \_ \vdash P\colon (U,T)}{\Gamma \vdash (\nu n)P\colon (U,T)} \qquad \text{(T-REC, T-RES)}$$

$$\Gamma, X\colon (U,T) \vdash X\colon (U,T) \qquad \Gamma \vdash \mathbf{0}\colon (\mathsf{end},T) \qquad \text{(T-VAR, T-NIL)}$$

Figure 10. The type system

The type of a process abstracts its behavior: the first component shows the protocol that the process wants to follow (provided that it is inserted in a suitable session) while the second component traces the type of the values fed to its stream. Notice that the properties of internal sessions and streams are guaranteed by the typing derivation and the typing assumption in Γ and they do not influence the type of the process. For instance if the process is a session $r \rhd P$, then its protocol is end, but the protocol followed by P is traced by an assumption $r\colon [U]$ in Γ. When the complementary session is found, the compatibility check is performed.

Our types force protocols to be sequential: we think that this is a good programming style. Suppose for instance that the protocol contains two parallel outputs: then there should be two inputs in the complementary protocol, and one can not know which output is matched with each input. Either this is not important (and in this case one can just sort the outputs in an arbitrary way) or it is, and in this second case errors could occur. Having parallel protocols also makes the check for protocol compatibility much more complex. Notice that this does not forbid, e.g., to have two concurrent service invocations, since sequentiality is only enforced in protocols.

As an example we show the typing judgment for the protocol of the hotel service in Example 4.4.

```
true:Bool, confirm:Flag, reject:Flag ⊢
  (date)(room) ... reject:
    ([?Date.?Room.!Int.?CC.!Flag.end],T)
```

We have supposed to have types Bool, Int, Date, Room, CC, and Flag to model domain specific data. Also, the hotel service does not feed into its stream, hence the arbitrary type T.

SSCC equipped with this type system is type safe. As usual this result requires a progress property—subject reduction—and a definition of erroneous processes.

Theorem 5.1 (Subject reduction) *Let P be a process such that $\Gamma \vdash P\colon (U,T)$ and $P \to P'$. Then $\Gamma \vdash P'\colon (U,T)$.*

Typable processes are not errors, nor can generate errors.

Theorem 5.2 (Type Safety) *Let P be a typable process. Then P has no subterm of the following forms.*

Protocol:

$\mathcal{D}\llbracket r \bowtie \mathcal{C}\llbracket v.P\rrbracket, r \overline{\bowtie} \mathcal{C}'\llbracket u.Q\rrbracket\rrbracket$	*Two outputs*
$\mathcal{D}\llbracket r \bowtie \mathcal{C}\llbracket v.P\rrbracket, r \overline{\bowtie} \mathbf{0}\rrbracket$	*Output and* $\mathbf{0}$
$\mathcal{D}\llbracket r \bowtie \mathcal{C}\llbracket (x)P\rrbracket, r \overline{\bowtie} \mathcal{C}'\llbracket (y)Q\rrbracket\rrbracket$	*Two inputs*
$\mathcal{D}\llbracket r \bowtie \mathcal{C}\llbracket (x)P\rrbracket, r \overline{\bowtie} \mathbf{0}\rrbracket$	*Input and* $\mathbf{0}$

where in all the cases $\mathcal{D}\llbracket, \rrbracket$ does not bind r, and $\mathcal{C}\llbracket\rrbracket$ and $\mathcal{C}'\llbracket\rrbracket$ do not contain sessions around the $\bullet$.

Sequentiality:

$\mathcal{D}\llbracket v.P, u.Q\rrbracket$	*Parallel outputs*
$\mathcal{D}\llbracket (x)P, u.Q\rrbracket$	*Parallel input and output*
$\mathcal{D}\llbracket v.P, (y)Q\rrbracket$	*Parallel output and input*
$\mathcal{D}\llbracket (x)P, (y)Q\rrbracket$	*Parallel inputs*

where in all cases $\mathcal{D}\llbracket, \rrbracket$ does not contain sessions around the $\bullet$.

An example of a *protocol failure* is $r \rhd v.P | r \lhd \mathbf{0}$, and this cannot be typed since the two parallel components require different assumptions for r ($r\colon$ [$!T.U'$] where T is the type of v, and $r\colon$ [end] respectively). Similarly a *non-sequential conversation* is $r \rhd (v.P|u.Q)$, and this cannot be typed since both $v.P$ and $u.Q$ have non end protocols, thus rules for parallel composition can not be applied.

Techniques used for session types can be adapted to *type check* SSCC processes [22].

6 Further analysis techniques

In this section we propose two "proof of concept" techniques to further highlight the suitability of SSCC. Their development is part of future work.

Service equivalence. Bisimilarity techniques are a common tool used in process calculi to obtain process equivalences. We show here how weak bisimilarity can be used to prove the equivalence between a service and a possible refined implementation.

We write $P \Rightarrow P'$ iff $P \xrightarrow{\tau} \xrightarrow{\tau} \dots \xrightarrow{\tau} P'$ and $P \overset{\mu}{\Rightarrow} P'$ iff $P \Rightarrow \xrightarrow{\mu} \Rightarrow P'$.

Definition 6.1 (Bisimilarity) *A* weak bisimulation *is a relation* R *such that* $P \,\mathrm{R}\, Q$ *implies:*
if $P \overset{\mu}{\Rightarrow} P' \wedge \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$ *then* $Q \overset{\mu}{\Rightarrow} Q' \wedge P' \,\mathrm{R}\, Q'$ *and vice versa.*

Weak bisimilarity *is the maximal weak bisimulation.*

Let us consider the simple service:

```
add2 ⇒ (n)n+2
```

A possible implementation of this service exploits a local service add1:

```
(ν add1)(add1 *⇒ (n)n+1 |
          add2 ⇒ (n) call add1(n) >¹ m >
                     call add1(m) >¹ o > o)
```

It is easy to prove that the two services add2 are weak bisimilar, thus the second one is a correct implementation of the first.

Towards deadlock avoidance. One of the main aims of this calculus is to allow the development of static analysis techniques to ensure some liveness properties, like, e.g., deadlock freedom for a large class of services. Even if this is outside the scope of the present paper, we highlight the main ideas.

There are three possible cases of deadlock in the language: non matched service definitions/invocations, non matched value send/receive in protocols, and read from empty stream. These correspond to different issues: service availability, communication errors, and orchestration errors. Different techniques can be developed to deal with the three problems separately: Section 5 shows e.g. how to avoid communication errors. One has also to check that there are no cyclic dependencies (e.g., a service waiting for a value from a stream, which can be produced only after the service execution has been completed). It is not difficult to ensure the following result.

Let P be a process such that:

Service availability: *all services are defined, persistent, and at top level, i.e., P has the form $(\nu\vec{a})(D|Q)$ where D is a parallel composition of persistent service definitions for all the services occurring in D and Q;*

Protocol compatibility: *P is typable according to the type system in Section 5;*

Orchestration: *for each subterm* stream Q' as $f = \vec{v}$ in Q'' *we have $n + |\vec{v}| \geq m$ where n is the number of feeds in Q' feeding f (i.e., not in the left part of other stream operators) and m is the number of reads from f in Q''. We additionally require that no read operation is inside a recursive process.*

For each computation $P \xrightarrow{\mu_1} \dots \xrightarrow{\mu_n} P'$, if P' has not the form $(\nu\vec{a}')D$ where D is the same as above (in this case the computation in Q is terminated), there are μ and P'' such that $P' \xrightarrow{\mu} P''$.

The main idea here is to avoid each possibility of blocked processes: all processes at top level can execute an action unless they depend on some other process, but the condition of service availability and directionality in stream communications guarantee that cyclic dependencies are avoided. Note that this result strongly exploits the features of the language, i.e., it would be very difficult to obtain it if unstructured communications are added.

7 Conclusion and future work

SSCC is a typed language aiming at flexibly describing *services*, *conversations*, and *orchestration*, with a restricted set of constructors. The expressivity of the language is witnessed by the simple implementation of all workflow patterns in [21] (except for the ones that require some form of explicit process termination) available in [15] and by the examples in Sections 2 and 4. We have shown instead in Sections 5 and 6 how different analysis techniques can be applied to the calculus.

Future works will start from there, since all these techniques can be refined. Some ideas include analyzing the relationships between contextual equivalence and bisimilarity and up-to techniques for bisimilarity, more detailed techniques for proving service availability (e.g., linearity of service invocation and definition) and proofs of deadlock freedom for large classes of protocols.

Another thread for future development concerns the development of a *compensation* mechanism to recover from failures, and to study the behavioral theory of such a mechanism.

Acknowledgments. This work was partially supported by FEDER, the EU IST proactive initiative FET-Global Computing (project Sensoria, IST–2005–16004). We thank L. Caires, R. Bruni, D. Sangiorgi, and G. Zavattaro for valuable comments and suggestions.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services—Concepts, Architectures and Applications*. Springer, November 2003.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services*. Version 1.1, 2003.

[3] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams. *Web Services Conversation Language (WSCL) 1.0*, 2002.

[4] T. Bellwood, L. Clément, D. Ehnebuske, A. Hately, M. Hondo, Y. L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, and C. von Riegen. *UDDI Version 3.0*, 2002.

[5] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: a service centered calculus. In *Proceedings of WS-FM 2006*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[6] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Sock: a calculus for service oriented computing. In *Proceedings of ICSOC'06*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.

[7] M. Carbone, K. Honda, N. Yoshida, and R. Milner. Structured communication-centred programming for web services. In *Proceedings of ESOP'07*, LNCS. Springer, 2007.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *WSDL: Web Services Definition Language*. World Wide Web Consortium, 2004.

[9] W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in orc. In *Proceedings of Coordination'06*, volume 4038 of *LNCS*, pages 82–96. Springer, 2006.

[10] FET-GC2 Workprogramme text. `http://www.cordis.lu/ist/fet/gc.htm`.

[11] S. J. Gay and M. J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

[12] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. *Simple Object Access Protocol (SOAP) 1.2*. World Wide Web Consortium, 2003.

[13] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[14] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *Proceedings of Concur'06*, pages 477–491, 2006.

[15] I. Lanese, V. T. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. DI/FCUL TR 07–3, Department of Informatics, Faculty of Sciences, University of Lisbon, Mar. 2007.

[16] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proceedings of ESOP'07*, LNCS. Springer, 2007.

[17] 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.

[18] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[19] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Proceedings of TPPP'94)*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.

[20] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[21] W. van der Aalst, B. Hofstede, and A. Kiepuszewski. Advanced workflow patterns. In *Proceedings of CoopIS'00*, volume 1901 of *LNCS*, pages 18–29. Springer, 2000.

[22] V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.

[23] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *Proceedings of IWSRT'06*, ENTCS, 2006.