# Behavioural Theory at Work: Program Transformations in a Service-Centred Calculus*

Luís Cruz-Filipe[2], Ivan Lanese[1], Francisco Martins[2], António Ravara[3], and Vasco T. Vasconcelos[2]

[1] Computer Science Department, University of Bologna, Italy
[2] Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal
[3] Security and Quantum Information Group, Instituto de Telecomunicações, and Department of Mathematics, IST, Technical University of Lisbon, Portugal

**Abstract.** We analyse the relationship between object-oriented modelling and session-based, service-oriented modelling, starting from a typical UML Sequence Diagram and providing a program transformation into a service-oriented model. We also provide a similar transformation from session-based specifications into request-response specifications. All transformations are specified in SSCC—a process calculus for modelling and analysing service-oriented systems—and proved correct with respect to a suitable form of behavioural equivalence (full weak bisimilarity). Since the equivalence is proved to be compositional, results remain valid in arbitrary contexts.

## 1 Introduction

Web Services include a set of technologies to deploy applications over the Internet, making them dynamically searchable and composable, allowing great adaptability and reusability. While nowadays Web Services are one of the main technologies for implementing coordinated behaviours over the web, most of the modelling techniques commonly used are still tailored to the object-oriented development style, and only recently did dedicated development methodologies start emerging. For instance, extensions of UML [14] for Web Services are under development [21], and dedicated languages such as WS-BPEL [2] are increasingly popular.

As a contribution to bridge the gap between traditional object-oriented and emerging service-oriented models, and to allow the reuse of existing tools and techniques, we detail a model transformation procedure from a common object-oriented communication pattern into a session-based, service-oriented one. *Sessions* are a paradigm of service interaction that isolates the communications between two remote partners, making them safe from interferences. More complex orchestration scenarios are obtained by combining remote session communications and local communications. The expressive power of sessions has been shown, *e.g.,* in [3, 4, 8, 11, 12, 19, 20].

* This work has been partially sponsored by the project SENSORIA, IST–2005–016004, and by FCT, through the Multiannual Funding Programme and project SpaceTimeTypes, POSC/EIA/55582/2004.

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 59–77, 2008.

The work presented in this paper builds on top of SSCC [10, 12], a process calculus based on session communication and featuring service definition and service invocation as first class constructs. SSCC has been developed within the EU project Sensoria [18], by adapting previous proposals of calculi for services (such as [3, 6, 13]), while trying to find the best trade-off between expressiveness and suitability for formal analysis. SSCC provides dedicated constructs for service definition/invocation, session communication, and local communication based on streams, the later abstracting operating system local communication facilities.

We illustrate herein how UML Sequence Diagrams [1] (hereafter referred to as SDs) can be straightforwardly implemented in SSCC by exploiting suitable macros. While these macros hide auxiliary local services, making use of services for local communications is usually not tremendously efficient. We thus show how to transform these models into models where local communications are realised by streams, resulting in systems with a more session-oriented flavour. However, while sessions have proved quite useful for system modelling, current Web Service technologies (*e.g.,* WSDL [9] and WS-BPEL [2]) do not fully support them. In fact, complex interactions are usually based on the simpler request and request-response patterns. To bridge this mismatch we show how to break sessions into request-response sequences, easily implementable using current technologies.

An important feature of our work is that all transformations are proved correct with respect to the observational semantics of services. In fact, SSCC is equipped with a rigorous operational semantics, and with a behavioural theory based on an equivalence capturing the main aspects of service interaction. We show that our transformations preserve behavioural equivalence. Since we also prove that this equivalence—weak full bisimilarity over an early labelled transition system—is a congruence, the behaviour of every composition built exploiting these services remains unchanged when moving from the original programs into their transformed versions.

To the best of our knowledge, there are very few studies on weak bisimulations for service calculi other than SSCC. Bruni et al. present weak bisimulation for a calculus for multiparty sessions in SOC [5], and they apply it to show that an implementation of a service is compliant to a more abstract specification. However they do not provide a general theory or methodology for the transformation as we do. Vieira et al. discuss strong bisimulation in the realm of the Conversation Calculus [20], a variant of SCC [3]. They present a congruence result and axioms that express the spatial nature of their calculus and allow them to prove a normal form lemma. The behavioural theory, however, is not used in an application, as we do in this paper. Busi et al. study a bisimulation like relation to check conformance between a calculus for orchestration and a calculus for choreography [7], while we use it to check the correctness of transformations inside the same calculus. Most of the papers on program transformations (see, *e.g.,* [15] for a functional language) use program transformations to improve efficiency, whereas we are more interested in changing the style of the program, while making it implementable using current technologies. In the context of typed $\pi$-calculus, Davide Sangiorgi uses typed behavioural equivalences to prove a program transformation to increase concurrency in a system of objects [16].

The outline of the paper is as follows. The next section introduces SSCC. Then Section 3 discusses SDs and the corresponding SSCC code. Sections 4 and 5 study the bisimilarity relation, the former for general processes and the latter concentrating on sequential sessions. They pave the way for proving, in Section 6, the correctness of the transformations set forward in Section 3. Finally, Section 7 concludes the paper, pointing directions for further work.

## 2 SSCC

The Stream-based Session-Centred Calculus [10, 12] builds on its predecessor SCC [3], by introducing the ability to model complex orchestration patterns via a dedicated stream construct. Here we recall its syntax and its operational semantics.

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

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

**Fig. 1.** Syntax of SSCC

The first five cases introduce standard process calculi operators: parallel composition, restriction (for service names only), the terminated process, and recursion (we assume recursion guarded by prefixes). We then have two constructs to *build services*: definition or provider ($a \Rightarrow P$) and invocation or client ($a \Leftarrow P$). Both are defined by their name $a$ and protocol $P$. Service definition and service invocation are symmetric. *Service protocols* are built using value sending ($v.P$) and receiving ($(x)P$), allowing bidirectional communication between clients and servers. Finally, there are three constructs for *service orchestration*. The stream construct declares a stream $f$ for communication from $P$ to $Q$. Process $P$ can insert a value $v$ into the stream using $\mathsf{feed}\, v.P'$, and process $Q$ can read from stream $f$ using $f(x).Q'$. Notice that stream names cannot be communicated, thus they model static channels. The rule of thumb concerning communication is the following: service interaction is intended for remote communication; stream interaction is intended for local communication.

Processes at run-time exploit an extended syntax: the interaction of a service definition and a service invocation produces an active session with two endpoints, one at the provider side and the other at the client side. Also, values in the stream are stored

| | | | | | |
|---|---|---|---|---|---|
| $P, Q$ ::= | $\dots$ | as in Figure 1 | | *Runtime processes* | |
| $\mid$ | $r \rhd P$ | Server session | $\mid$ | $r \lhd P$ | Client session |
| $\mid$ | $(\nu\, r)P$ | Session restriction | $\mid$ | $\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q$ | Stream with values |

**Fig. 2.** Run-time syntax of SSCC

together with the stream definition, *i.e.,* the stream has buffering capabilities. Let $\vec{v}$ (and sometimes $\vec{w}$) denote a (possibly empty) sequence of values. We introduce a fourth kind of identifier: *session names*, ranged over by $r, s, \dots$, and use $n, m, \dots$ to range over both session and service names. The grammar in Figure 2 defines the *syntax of run-time processes*. The constructor $\mathsf{stream}\, P\, \mathsf{as}\, f\, \mathsf{in}\, Q$ in Figure 1 is an abbreviation for its runtime version $\mathsf{stream}\, P\, \mathsf{as}\, f = \langle\rangle\, \mathsf{in}\, Q$.

In order to define the operational semantics we introduce a few auxiliary notations. We use $r \bowtie P$ to denote one of $r \lhd P$ and $r \rhd P$, and we assume that when multiple $\bowtie$ appear in the same rule they are instantiated in the same direction ($\lhd$ or $\rhd$), and that if $\overline{\bowtie}$ also appears, then it denotes the opposite direction.

Streams are considered ordered, acting as queues. We write $w :: \vec{v}$ for the stream obtained by enqueuing $w$ to $\vec{v}$, and $\vec{v} :: w$ for a stream from which $w$ can be dequeued. In the latter case $\vec{v}$ is what remains after dequeuing $w$.

*Operational Semantics.* Let $\mathrm{Set}(\vec{w}) = \{w_1, \dots, w_n\}$, when $\vec{w} = w_1 \cdots w_n$ $(n \geq 0)$. 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 $\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q$ with scope $Q$; and process variable $X$ is bound in $\mathsf{rec}\, X.P$. Notation $\mathrm{fn}(P)$ (respectively $\mathrm{bn}(P)$) denotes the set of free (respectively bound) names in $P$, and $\mathrm{n}(P) = \mathrm{fn}(P) \cup \mathrm{bn}(P)$. We require processes to have no free process variables. The semantics of SSCC is defined using a labelled transition system (LTS, for short) in the early style.

**Definition 1 (Labelled Transition System).** *The rules in Figure 3, together with the symmetric version of rules* L-PAR*,* L-PAR-CLOSE*, and* L-SESS-COM-CLOSE*, inductively define the LTS on processes.*

In the LTS we use $\mu$ as a metavariable for labels, and extend $\mathrm{fn}(-)$, $\mathrm{bn}(-)$, and $\mathrm{n}(-)$ to labels. The only bound names in labels are $r$ in service definition activation and service invocation and $a$ in extrusion labels (conventionally, they are all in parenthesis). Label $\uparrow v$ denotes sending (the output) value $v$. Dually, label $\downarrow v$ is receiving (the input) value $v$. We use $\updownarrow v$ to denote one of $\uparrow v$ or $\downarrow v$, and we assume that when multiple $\updownarrow v$ appear in the same rule they are instantiated in the same direction, and that $\overline{\updownarrow} v$ denotes the opposite direction.

Continuing with the labels, $a \Leftarrow (r)$ and $a \Rightarrow (r)$ denote respectively the request and the activation of a service, where $a$ is the name of the service, and $r$ is the name of the new session to be created. We use $a \Leftrightarrow (r)$ to denote one of $a \Leftarrow (r)$ or $a \Rightarrow (r)$, and we assume that when multiple $a \Leftrightarrow (r)$ appear in the same rule they are instantiated in the same direction, and that $a \overline{\Leftrightarrow} (r)$ denotes the opposite direction. Furthermore, label $\Uparrow v$ denotes the feeding of value $v$ into a stream, while label $f \Downarrow v$ reads value $v$ from stream $f$. When an input or an output label crosses a session construct (rule L-SESS-VAL), we

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

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

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

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

$$\frac{P \xrightarrow{\Uparrow v} P'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\tau} \mathsf{stream}\, P'\, \mathsf{as}\, f = v :: \vec{w}\, \mathsf{in}\, Q} \quad \text{(L-STREAM-FEED)}$$

$$\frac{Q \xrightarrow{f \Downarrow v} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w} :: v\, \mathsf{in}\, Q \xrightarrow{\tau} \mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \quad \text{(L-STREAM-CONS)}$$

$$\frac{r \notin \mathrm{fn}(P)}{a \Leftarrow P \xrightarrow{a \Leftarrow (r)} r \lhd P} \qquad \frac{r \notin \mathrm{fn}(P)}{a \Rightarrow P \xrightarrow{a \Rightarrow (r)} r \rhd P} \quad \text{(L-CALL, L-DEF)}$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P|Q \xrightarrow{\mu} P'|Q} \qquad \frac{P \xrightarrow{\updownarrow v} P'}{r \bowtie P \xrightarrow{r \bowtie \updownarrow v} r \bowtie P'} \quad \text{(L-PAR, L-SESS-VAL)}$$

$$\frac{P[^{\mathsf{rec}\, X.P}/_{X}] \xrightarrow{\mu} P'}{\mathsf{rec}\, X.P \xrightarrow{\mu} P'} \qquad \frac{P \xrightarrow{r \bowtie \uparrow v} P' \quad Q \xrightarrow{r \overline{\bowtie} \downarrow v} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{r\tau} \mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'}$$

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

$$\frac{P \xrightarrow{a \Leftrightarrow (r)} P' \quad Q \xrightarrow{a \overline{\Leftrightarrow} (r)} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\tau} (\nu\, r)\mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \quad \text{(L-SERV-COM-STREAM)}$$

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

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

$$\frac{P \xrightarrow{\mu} P' \quad n \notin \mathrm{n}(\mu)}{(\nu\, n)P \xrightarrow{\mu} (\nu\, n)P'} \qquad \frac{P \xrightarrow{r\tau} P'}{(\nu\, r)P \xrightarrow{\tau} (\nu\, r)P'} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \neq \updownarrow v \quad r \notin \mathrm{bn}(\mu)}{r \bowtie P \xrightarrow{\mu} r \bowtie P'}$$

(L-RES,L-SESS-RES, L-SESS-PASS)

$$\frac{P \xrightarrow{r \bowtie (v) \uparrow v} P' \quad Q \xrightarrow{r \overline{\bowtie} \downarrow v} Q' \quad v \notin \mathrm{fn}(Q)}{P|Q \xrightarrow{r\tau} (\nu\, v)(P'|Q')} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \in \{\uparrow a, r \bowtie \uparrow a, \Uparrow a\}}{(\nu\, a)P \xrightarrow{(a)\mu} P'}$$

(L-PAR-CLOSE, L-EXTR)

$$\frac{P \xrightarrow{r \bowtie (v) \uparrow v} P' \quad Q \xrightarrow{r \overline{\bowtie} \downarrow v} Q' \quad v \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{r\tau} (\nu\, v)\mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \quad \text{(L-SESS-COM-CLOSE)}$$

$$\frac{P \xrightarrow{(v) \Uparrow v} P' \quad v \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\tau} (\nu\, v)\mathsf{stream}\, P'\, \mathsf{as}\, f = v :: \vec{w}\, \mathsf{in}\, Q} \quad \text{(L-FEED-CLOSE)}$$

**Fig. 3.** Labelled Transition System

$$P|\mathbf{0} \;\equiv\; P \qquad P|Q \;\equiv\; Q|P \qquad (P|Q)|R \;\equiv\; P|(Q|R)$$
(S-NIL, S-COMM, S-ASSOC)

$$(\nu\, n)P|Q \;\equiv\; (\nu\, n)(P|Q) \quad \text{if } n \notin \mathrm{fn}(Q) \qquad r \bowtie (\nu\, a)P \;\equiv\; (\nu\, a)(r \bowtie P)$$
(S-EXTR-PAR, S-EXTR-SESS)

$$\mathsf{stream}\,(\nu\, a)P\,\mathsf{as}\, f = \vec{v}\,\mathsf{in}\, Q \;\equiv\; (\nu\, a)(\mathsf{stream}\, P\,\mathsf{as}\, f = \vec{v}\,\mathsf{in}\, Q) \quad \text{if } a \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{v})$$
(S-EXTR-STREAML)

$$\mathsf{stream}\, P\,\mathsf{as}\, f = \vec{v}\,\mathsf{in}\,(\nu\, a)Q \;\equiv\; (\nu\, a)(\mathsf{stream}\, P\,\mathsf{as}\, f = \vec{v}\,\mathsf{in}\, Q) \quad \text{if } a \notin \mathrm{fn}(P) \cup \mathrm{Set}(\vec{v})$$
(S-EXTR-STREAMR)

$$(\nu\, n)(\nu\, m)P \equiv (\nu\, m)(\nu\, n)P \qquad (\nu\, a)\mathbf{0} \;\equiv\; \mathbf{0} \qquad \mathsf{rec}\, X.P \;\equiv\; P[{}^{\mathsf{rec}\, X.P}/_{X}]$$
(S-SWAP, S-COLLECT, S-REC)

**Fig. 4.** Structural congruence

add to the label the name of the session and whether it is a server or a client session (for example, $\downarrow v$ may become $r \triangleleft \downarrow v$).

The label denoting a conversation step in a free session $r$ is $r\tau$, and if the value passed in the session channel is private, it remains private in the resulting process; rules L-PAR-CLOSE and L-SESS-COM-CLOSE accomplish this using the usual $\pi$-calculus approach. A label $\tau$ is obtained only when $r$ is restricted (rule L-SESS-RES). Thus a $\tau$ action can be obtained in four cases: a communication inside a restricted session, a service invocation, a feed or a read from a stream. Notice also that we have two contexts causing interaction: parallel composition and stream. Finally, bound actions, $(a)\mu$, represent the extrusion of $a$ in their respective free counterparts $\mu$.

This LTS is slightly different from its original version [12]. In particular it does not exploit structural congruence, in order to simplify some proofs. The two LTSs are, however, equivalent up to the structural congruence relation (inductively defined by the rules in Figure 4). Since we show in Lemma 2 that structural congruence is included in strong full bisimilarity, all our results are valid also for the original LTS. A detailed equivalence proof can be found in a technical report [10].

Some processes, such as $r \triangleleft r \triangleleft P$, can be written using the run-time syntax, but they are not reachable from processes in the basic syntax of Figure 1. We consider these processes ill-formed, and therefore make the following assumption, which is necessary for most results.

**Assumption 1.** *Henceforth, all processes under consideration are either in the syntax of Figure 1, or can be obtained from these via LTS transitions.*

## 3 Common Sequence Diagram Patterns

UML Sequence Diagrams [1] describe the exchange of messages among the components in a complex system. We present a typical SD, and then describe how the same behaviour can be modelled first in a more session-centred style, and then in a request-response style suitable for implementation. Diagrams for the initial SD and those describing the result of each transformation are in Figures 5 to 7.

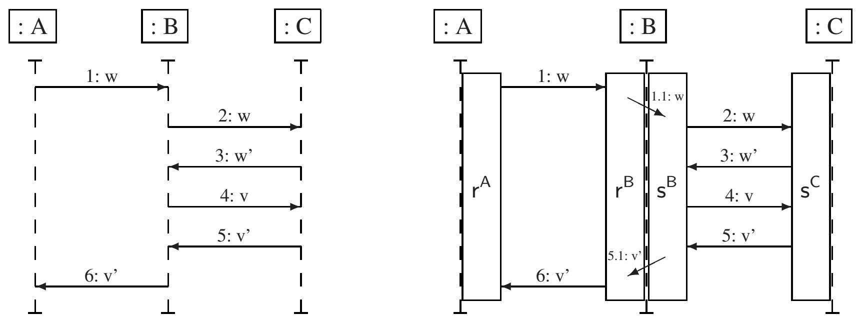


**Fig. 5.** Sequence diagram communication pattern: object-centred and session-centred view

*Object-Oriented View.* The SD on the left of Figure 5 describes a very common pattern appearing in scenarios involving (at least) three partners. The description of the communication pattern is as follows.

> Object B receives from object A the value w and forwards it to object C. After receiving the value, object C answers with a value w'. Object B replies with v and finally object C replies with v'. Now object B forwards it to object A.

Notice that by "Object B receives from object A the value w" we mean that object A invokes a method in object B passing the value w.

*Session-Centred View.* Assume that the components of this abstract communication scenario are clients and servers of a service-oriented architecture, and further assume that communication happens via sessions. We refine the diagram by incorporating information about the running sessions, in the diagram on the right of Figure 5, where the slanted arrows mean message passing between sessions[1]. An instance of service B (let us call these instances *participants*) has a session r running with an instance of service A and another session s running with an instance of service C. Since sessions involve two partners, a session r between instances of services A and B has two sides—called endpoints, $r^A$ at the instance of service A and $r^B$ at the instance of service B. The communication pattern is now like this:

> Participant B receives in session $r^B$ the value w, passes it to its part of the session with participant C ($s^B$), and then forwards the value through this session to C. Inside the same session C sends w' to B, B sends v to C and C sends v' to B. Participant B now forwards the value v' back to A, passing it from session s to session r.

In addition to the normal constructs in the calculus, to model object-oriented systems (that do not follow the laws of session communication), it is useful to have two

[1] Since to the best of our knowledge no extension of SDs with session information exists, we introduce a notation for it.

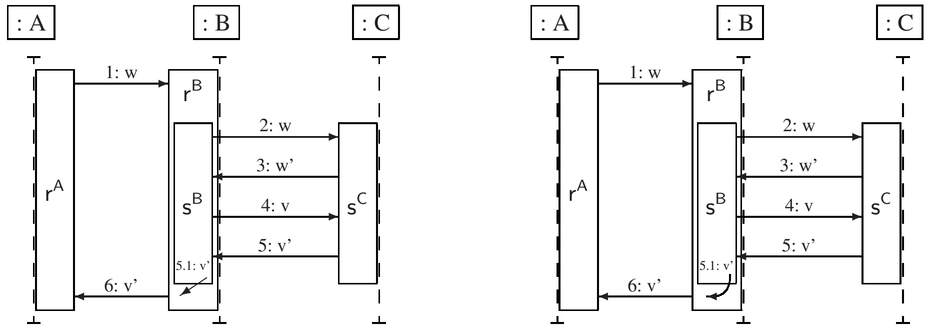


**Fig. 6.** Sequence diagram: subsession communicating via message passing and via a stream

constructs enabling arbitrary message passing. These can be expressed in SSCC by exploiting fresh auxiliary services.

$$b \Uparrow \langle v_1, ..., v_n\rangle.P \triangleq \textsf{stream}\, b \Leftarrow v_1...v_n.\textsf{feed unit as}\ f\ \textsf{in}\ f(v).P$$

$$b \Downarrow (x_1, ..., x_n)P \triangleq \textsf{stream}\, b \Rightarrow (z_1)...(z_n).\textsf{feed}\, z_1 ...\textsf{feed}\, z_n\ \textsf{as}\ f$$
$$\textsf{in}\ f(x_1)...f(x_n).P$$

where name $v$ and stream $f$ are not used in $P$.

The diagram on the right of Figure 5 is directly implemented in SSCC as

$$\mathsf{SC} \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{B} \mid \mathsf{C}\ ),$$

where

$$\mathsf{A} \triangleq b \Leftarrow w.(y)P,\ \mathsf{B} \triangleq (\nu\, b_1, b_2)\ (\ \mathsf{B}_1 \mid \mathsf{B}_2\ ),\ \text{and } \mathsf{C} \triangleq c \Rightarrow (x)w'.(y)v'.S,$$

with

$$\mathsf{B}_1 \triangleq b \Rightarrow (x)b_1 \Uparrow x.b_2 \Downarrow (y)y.Q,\ \text{and } \mathsf{B}_2 \triangleq c \Leftarrow b_1 \Downarrow (x)x.(z)v.(y)b_2 \Uparrow y.R.$$

The process above, although not deterministic (its first step may either be the invocation of service b or of service c), is confluent, and it is easy to check that its behaviour reflects the one described on the right of Figure 5.

*A First Optimisation.* When the participant B has the value sent by A, it may immediately send it to participant C, calling it (and thus opening a subsession). One simply has to perform a "local" transformation on B. The resulting diagram is on the left of Figure 6, and it is implemented in SSCC as process SC′, where we denote by E the new instance of B.

$$\mathsf{SC}' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{E} \mid \mathsf{C}\ ) \qquad \text{where}$$
$$\mathsf{E} \triangleq b \Rightarrow (x)(\nu\, b_1)(c \Leftarrow x.(z)v.(y)b_1 \Uparrow y.R \mid b_1 \Downarrow (y)y.Q)$$

Still, E (which replaces B from the previous version) needs to pass the value sent by C in their session, to its session with A, and a communication based on an auxiliary service is used. Notice that the process SC′ is deterministic.

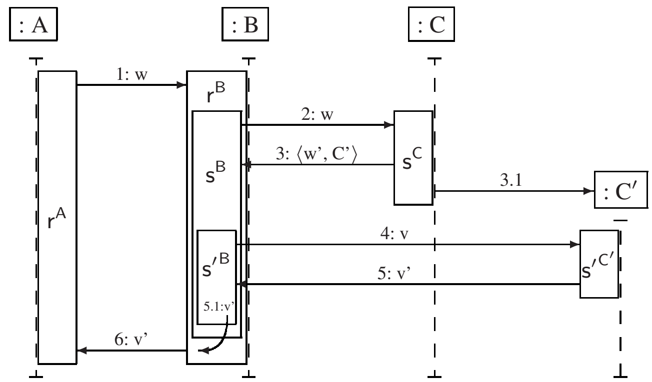


**Fig. 7.** Sequence diagram communication pattern: using subsessions and continuations

*A Second Optimisation.* The transfer of a value from a subsession to its parent session is, in the previous implementation, not straightforward, since it requires the use of local services. In fact, as discussed in [12], it is more convenient to use a trans-session construct like stream. Now participant F (the initial B), passes the value received from C from the subsession to the main session using another communication construct—a stream—instead of using a local service. This is implemented in process SC′′, also deterministic.

$$\mathsf{SC}'' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{F} \mid \mathsf{C}\ )$$

where the new code for B (now F) is as follows.

$$\mathsf{F} \triangleq b \Rightarrow (x)(\mathsf{stream}\ c \Leftarrow x.(z)v.(y)\mathsf{feed}\, y.R\ \mathsf{as}\ f\ \mathsf{in}\ f(y).y.Q)$$

The corresponding diagram is on the right of Figure 6. The two diagrams in Figure 6 are quite similar: the one on the left uses message passing (denoted by the straight arrow from the inner to the outer session), whereas the one on the right uses stream communication (described by the curved arrow).

*Implementing the Diagram.* The previous diagram, and the corresponding SSCC process, model the pattern at hand in a service-oriented session-based style. However, current Web Service technologies do not provide support for a complex mechanism such as sessions, considering instead request (one-way communication) and request-response (two one-way communications in opposite directions) only—see, *e.g.,* the definition of WSDL [9]. It is easy to see that these are particular cases of sessions, where protocols are composed respectively by one output or by one output followed by one input. The new communication pattern is described in Figure 7, and reads as follows.

> Participant B receives in session $\mathsf{r}^{\mathsf{B}}$ the value w, and then forwards it through its session with participant C ($\mathsf{s}^{\mathsf{B}}$) to C itself. Inside the same session C sends to B value w' together with the name of a freshly generated service C' to continue the conversation on. Now B invokes C' creating a new subsession $\mathsf{s}'$ of session s and, inside $\mathsf{s}'$, B sends v and receives as answer v'. Participant B now forwards the value v' back to A, passing it from session $\mathsf{s}'$ to session r.

This pattern can be implemented as:

$$\mathsf{SC}''' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{G} \mid \mathsf{D}\ )$$

where the new codes for $\mathsf{B}$ (now $\mathsf{G}$) and $\mathsf{C}$ (now $\mathsf{D}$) are below. To write these new codes we need to consider polyadic inputs and outputs, denoted respectively by $(x_1, \ldots, x_n)$ and $\langle v_1, \ldots, v_n \rangle$. They can be easily accommodated in the theory.

$$\mathsf{G} \triangleq b \Rightarrow (x)(\mathsf{stream}\ c \Leftarrow x.(z, c')c' \Leftarrow v.(y)\mathsf{feed}\, y.R\ \mathsf{as}\ f\ \mathsf{in}\ f(y).y.Q)$$

$$\mathsf{D} \triangleq c \Rightarrow (x)(\nu\, c')\langle w', c' \rangle.c' \Rightarrow (y)v'.S$$

Naturally, one asks whether the transformations of SC into SC$'$, SC$''$ and finally SC$'''$ are correct, not changing the observable behaviour of processes. The next sections introduce suitable tools to give a positive answer to this question in Section 6.

## 4 Bisimilarity in SSCC

We study the usual two notions of bisimilarity, strong and weak. Both are non-input congruences in the class of SSCC processes. One can get a congruence by considering (strong or weak) full bisimilarity, *i.e.,* by closing bisimilarity with respect to service name substitutions (notice that there is no reason to close with respect to session or stream names, since no substitutions are performed on them). Although the general strategy is the same as for the $\pi$-calculus, the proof techniques themselves differ significantly. Herein we only present the main results. Detailed proofs can be found in a technical report [10].

Section 2 defines a labelled transition system (LTS) in the early style; we now study the notions of bisimilarity known in the literature as *ground* and *full*. The reason for choosing these is simple: we are interested in a compositional equivalence, *i.e.,* equivalent services should behave the same even if used as parts of complex systems. Therefore, we choose the simplest possible setting where this may happen. It is well-known already from the $\pi$-calculus that ground bisimilarity over a late LTS is not preserved by parallel composition, requiring the more demanding notions of late and early bisimilarity (which in turn are not preserved by input prefix, since they are not closed under general substitutions). Not surprisingly, this fact also occurs in SSCC: ground bisimilarity is a non-input congruence. Sangiorgi and Walker present counter-examples for the preservation of both strong and weak bisimilarities in the synchronous $\pi$-calculus without sum and match [17, pp. 224–225]. Of these, the first can be easily translated to our language, using services to mimic $\pi$'s input and output constructors; for the weak case, the counter-example is not so directly transposable, but it can still be adapted.

*Strong Bisimilarity.* The relation, hereafter referred to simply as "bisimilarity", is defined as usual over the class of all processes.

**Definition 2 (Strong Bisimilarity).** *A symmetric binary relation $\mathcal{R}$ on processes is a* (strong) bisimulation *if, for any processes $P$, $Q$ such that $P\mathcal{R}Q$, if $P \xrightarrow{\alpha} P'$ with* $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$*, then there exists $Q'$ such that $Q \xrightarrow{\alpha} Q'$ and $P'\ \mathcal{R}\ Q'$. (Strong)* *bisimilarity $\sim$ is the largest bisimulation. Two processes $P$ and $Q$ are* (strong) bisimilar *if $P \sim Q$.*

*Also, a full bisimulation is a bisimulation closed under service name substitutions, and we call full bisimilarity $\sim_{\mathrm{f}}$ the largest full bisimulation.*

Since bisimilarity is not closed under service name substitutions, $\sim_{\mathrm{f}} \subsetneq \sim$.

Notice that bisimilarity (respectively full bisimilarity) can be obtained as the union of all bisimulations (respectively full bisimulations) or as a fixed-point of a suitable monotonic operator; moreover, as expected, structurally congruent processes (Figure 4) are bisimilar.

**Lemma 1 (Harmony Lemma).** *Let $P$ and $Q$ be processes with $P \equiv Q$. If $P \xrightarrow{\alpha} P'$, then $Q \xrightarrow{\alpha} Q'$ with $P' \equiv Q'$, and vice-versa.*

**Lemma 2.** *Structurally congruent processes are full bisimilar.*

As in the $\pi$-calculus, bisimilarity is a non-input congruence, *i.e.,* it is closed under contexts different from value reception and input from stream.

**Theorem 1.** *Bisimilarity is a non-input congruence.*

**Corollary 1.** *Full bisimilarity is a congruence.*

*Weak Bisimilarity.* As usual, we introduce some abbreviations: $P \stackrel{\tau}{\Rightarrow} Q$ iff $P \xrightarrow{\tau} \cdots \xrightarrow{\tau} Q$ and $P \stackrel{\alpha}{\Rightarrow} Q$ iff $P \stackrel{\tau}{\Rightarrow} \xrightarrow{\alpha} \stackrel{\tau}{\Rightarrow} Q$ for $\alpha \neq \tau$. In particular, $P \stackrel{\tau}{\Rightarrow} P$ for every process $P$.

**Definition 3.** *A symmetric binary relation $\mathcal{R}$ on processes is a* weak bisimulation *if, for any processes $P$, $Q$ such that $P\mathcal{R}Q$, if $P \stackrel{\alpha}{\Rightarrow} P'$ with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$, then there exists a process $Q'$ such that $Q \stackrel{\alpha}{\Rightarrow} Q'$ and $P' \ \mathcal{R}\ Q'$. Weak bisimilarity $\approx$ is the largest weak bisimulation. Two processes $P$ and $Q$ are said to be* weak bisimilar *if $P \approx Q$.*

*Also, a full weak bisimulation is a weak bisimulation closed under service name substitutions, and we call full weak bisimilarity $\approx_{\mathrm{f}}$ the largest full weak bisimulation.*

Again, weak bisimilarity can be obtained as the union of all weak bisimulations or as a fixed-point of a suitable monotonic operator. Moreover, as for the strong case, weak bisimilarity is a non-input congruence, as in in the $\pi$-calculus.

**Theorem 2.** *Weak bisimilarity is a non-input congruence.*

**Corollary 2.** *Weak full bisimilarity is a congruence.*

*Useful Axioms.* Even if presenting a complete axiomatization for such a complex calculus is out of the scope of this paper, we present here some axioms (equational laws correct with respect to strong/weak full bisimilarity) that capture key facts about the behaviour of processes. Some of them are useful to prove the correctness of the transformations presented in the previous section. We need the following definitions of contexts and double contexts.

**Definition 4 (Contexts and double contexts).** *A context $\mathcal{C}[\![\bullet]\!]$ is any* SSCC *process where a subterm has been replaced by the symbol $\bullet$. The application $\mathcal{C}[\![P]\!]$ of context $\mathcal{C}[\![\bullet]\!]$ to process $P$ is the process obtained by replacing $\bullet$ with $P$ inside $\mathcal{C}[\![\bullet]\!]$.*

*A double context $\mathcal{D}[\![\bullet_1, \bullet_2]\!]$ is any* SSCC *process where two subterms have been replaced by symbols $\bullet_1$ and $\bullet_2$. The application $\mathcal{D}[\![P_1, P_2]\!]$ of double context $\mathcal{D}[\![\bullet_1, \bullet_2]\!]$ to processes $P_1$ and $P_2$ is the process obtained by replacing $\bullet_1$ with $P_1$ and $\bullet_2$ with $P_2$ inside $\mathcal{D}[\![\bullet_1, \bullet_2]\!]$.*

Unless otherwise specified, the correctness of the axioms below can be proved by considering as full bisimulation all the instances of the equations together with the identity.

**Proposition 1**

**Session Garbage Collection**

$$(\nu\, \mathsf{r})\mathcal{D}[\![\mathsf{r} \rhd \mathbf{0}, \mathsf{r} \lhd \mathbf{0}]\!] \sim_{\mathrm{f}} \mathcal{D}[\![\mathbf{0}, \mathbf{0}]\!] \qquad \textit{where } \mathcal{D} \textit{ does not bind } r \tag{1}$$

**Stream Garbage Collection**

$$\mathsf{stream}\ \mathbf{0}\ \mathsf{as}\ f\ \mathsf{in}\ P \sim_{\mathrm{f}} P \qquad \textit{if } f \textit{ does not occur in } P \tag{2}$$

**Session Independence**

$$r \bowtie \mathsf{Q} \mid s \bowtie \mathsf{P} \sim_{\mathrm{f}} r \bowtie (s \bowtie \mathsf{Q} \mid \mathsf{P}) \qquad \textit{if } s \neq r \tag{3}$$

*The same holds if the sessions have opposite polarities.*

**Stream Independence**

$$\begin{aligned}&\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ \mathsf{stream}\ P'\ \mathsf{as}\ g\ \mathsf{in}\ Q \sim_{\mathrm{f}} \\ &\qquad \mathsf{stream}\ P'\ \mathsf{as}\ g\ \mathsf{in}\ \mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q \qquad \textit{if } f \neq g\end{aligned} \tag{4}$$

**Streams are Orthogonal to Sessions**

$$r \bowtie (\mathsf{feed}\ v \mid P) \sim_{\mathrm{f}} \mathsf{feed}\ v \mid r \bowtie P \tag{5}$$

**Stream Locality**

$$\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ (Q \mid Q') \sim_{\mathrm{f}} (\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q) \mid Q', \quad \textit{if } f \notin \mathrm{fn}(Q') \tag{6}$$

**Unused Stream**

$$\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ 0 \approx_{\mathrm{f}} P\{\mathsf{feed}\ v.Q \to Q\} \tag{7}$$

**Parallel Composition Versus Streams**

$$\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q \sim_{\mathrm{f}} P|Q \qquad \textit{if } f \notin \mathrm{fn}(Q) \textit{ and } P \textit{ does not contain } \mathsf{feed} \tag{8}$$

The Session Independence law shows that different sessions are independent. Interestingly this property is strongly dependent on the available operators, and fails in similar calculi such as [3, 4].

The notation $\{\mathsf{feed}\ v.Q \to Q\}$ in the Unused Stream law (Axiom 7) denotes a transformation on processes defined by induction on the syntax which is the identity but for transforming $\mathsf{feed}\ v.Q$ into $Q$. Notice also that Axiom 7 is correct only with respect to full weak bisimilarity. It becomes correct with respect to strong full bisimilarity if and only if $P$ does not contain any feed which is not inside another stream. Together with Equation 2 this allows to prove the relation between streams and parallel composition in Equation 8.

$$\frac{P : U}{v.P : !.U} \qquad \frac{P : U}{(x)P : ?.U} \qquad \frac{P : U}{a \Rightarrow P : \mathsf{end}} \qquad \frac{P : U}{a \Leftarrow P : \mathsf{end}}$$

(T-SEND, T-RECEIVE, T-DEF, T-CALL)

$$\frac{P : U}{r \rhd P : \mathsf{end}} \qquad \frac{P : U}{r \lhd P : \mathsf{end}} \qquad \frac{P : U}{\mathsf{feed}\, v.P : U} \qquad \frac{P : U}{f(x).P : U}$$

(T-SESS-S, T-SESS-C, T-FEED, T-READ)

$$\frac{P : U \quad Q : \mathsf{end}}{P|Q : U} \qquad \frac{P : \mathsf{end} \quad Q : U}{P|Q : U}$$

(T-PAR-L, T-PAR-R)

$$\frac{P : U \quad Q : \mathsf{end}}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q : U} \qquad \frac{P : \mathsf{end} \quad Q : U}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q : U}$$

(T-STREAM-L, T-STREAM-R)

$$\frac{P : \mathsf{end}}{\mathsf{rec}\, X.P : \mathsf{end}} \qquad \frac{P : U}{(\nu\, n)P : U} \qquad X : \mathsf{end} \qquad \mathbf{0} : \mathsf{end}$$

(T-REC, T-RES, T-VAR, T-NIL)

**Fig. 8.** Type system for sequentiality

## 5 Breaking Sequential Sessions

The equations shown in Section 4 hold for general processes, and will allow us to prove the correctness of the two optimizations in Section 3. However to prove the correctness of the implementation step they are not enough.

In fact, it is not easy to break a session allowing the conversation to continue in a freshly generated new session, since, in general, communication patterns inside sessions can be quite complex, *e.g.,* since sessions may include many ongoing concurrent communications. However, a small class of sequential sessions captures the most interesting within-session behaviours. Such a class can be identified by a type system.

We start by presenting the type system for sequentiality, and then we will present some properties of well-typed processes that will allow us to prove the correctness of the last transformation in Section 3. The type system is a simplification of the one in [12], which guarentees also protocol compatibility. Moreover, we consider just finite types, thus session protocols should be finite. Notice that this constraint does not forbid infinite behaviours, but just infinite sessions. In particular, if a process is typable according to the type system in [12], and all the involved types are non recursive, then the process is typable according to the type system presented herein.

We consider typed processes of the form $P : U$ where $U$ is the protocol type. We consider as types $?.U$, $!.U$, and $\mathsf{end}$, denoting respectively a protocol that performs an input and then continues as prescribed by $U$, a protocol that performs an output and then continues as prescribed by $U$, and the terminated protocol. It is clear that in this setting a request is a session with protocol $!.\mathsf{end}$ (and complementary protocol $?.\mathsf{end}$), while a request-response has protocol $!.?.\mathsf{end}$ (and complementary protocol $?.!.\mathsf{end}$). The type system is inductively defined by the rules in Figure 8.

Under the typability assumption, SSCC sessions are sequential in a very strong sense: we can statically define a correspondence between inputs and outputs such that each input is always matched by the corresponding output. We show how to break sessions, *i.e.,* how to make the conversation continue on a freshly created new session. The general law is presented under Theorem 3. The two pieces of sessions have protocols that are simpler than the original one, thus by repeatedly applying the transformation we can reduce any protocol to a composition of request and request-response patterns. We formalise the procedure described so far.

**Definition 5.** *Let $P$ be a process. An input/output prefix inside $P$ is* at top-level in $P$ *if it is neither inside a service definition/invocation nor inside a session. Given a process $P$ we can assign sequential indices to top-level input/output prefixes in $P$ according to the position of their occurrence in the term, starting from* $1$. *Thus the* $i$-th top-level prefix in $P$ *is the top-level prefix in $P$ that occurs in $i$-th position.*

For instance, let $P$ be $a.(x).\mathsf{stream}\,(y).\mathsf{feed}\,y\,\mathsf{as}\,f\ \mathsf{in}\,f(y).y \Leftarrow a.(z).\mathsf{feed}\,z$. Then $P$ annotated with indices on its top-level prefixes is:

$$a : 1.(x) : 2.\mathsf{stream}\,(y) : 3.\mathsf{feed}\,y\,\mathsf{as}\,f\ \mathsf{in}\,f(y).y \Leftarrow a.(z).\mathsf{feed}\,z$$

**Definition 6 (Active contexts).** *Active contexts are contexts defined by the following grammar.*

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

**Definition 7.** *Given a process $P$ with a subterm $Q$, we say that $Q$ is enabled in $P$ if $P = \mathcal{A}[\![Q]\!]$ for some active context $\mathcal{A}[\![\bullet]\!]$. The same definition holds also for prefixes.*

Intuitively a subterm is enabled when it can execute.

**Lemma 3.** *Let $P$ be a typable process. If $P$ has type* $\mathsf{end}$ *then it has no top-level enabled prefixes, otherwise it has exactly one top-level enabled prefix, and this has index* $1$.

*Proof.* By induction on the typing proof of $P$. The thesis follows trivially for rules T-SEND, T-RECEIVE, T-DEF, T-CALL, T-SESS-S, T-SESS-C, T-FEED, T-READ, and T-NIL. For rule T-RES, it follows by inductive hypothesis. For rules T-PAR-L, T-PAR-R, T-STREAM-L, and T-STREAM-R, it follows from the observation that one of the two sides has no enabled prefix, thus inductive hypothesis can be applied on the other side.

**Definition 8.** *Given a transition $P \xrightarrow{\alpha} Q$ and a prefix in $P$ we say that the prefix is consumed if, in the derivation of the transition, rule* L-SEND *or* L-RECEIVE *is applied to the prefix.*

Notice that the consumed prefix does not occur in $Q$.

**Lemma 4.** *Let $P_0 = (\nu\, a)\mathcal{D}[\![a \Rightarrow P, a \Leftarrow Q]\!]$ be a typed process such that $a$ does not occur in $\mathcal{D}[\![\bullet,\bullet]\!]$. Suppose $P_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{n-1}} P_n$. Then either:*

- $P_n = (\nu\, a)\mathcal{D}'[\![a \Rightarrow P, a \Leftarrow Q]\!]$ *for some $\mathcal{D}'[\![\bullet,\bullet]\!]$ or*

– $P_n = (\nu\, r)\mathcal{D}'[\![r \rhd P', r \lhd Q']\!]$ *for some* $\mathcal{D}'[\![\bullet, \bullet]\!]$*, processes* $P'$ *and* $Q'$ *and session name* $r$*.*

*Furthermore, whenever the* $i$*-th prefix in* $P$ *is consumed the* $i$*-th prefix in* $Q$ *is consumed too, and the two are synchronised.*

*Proof.* The proof is by induction on $n$. The base case ($n = 0$) is trivial. Let us consider the inductive case. In order to prove the second condition we show that the following property holds: at each step either (i) all the prefixes preserve their indices, or (ii) prefixes with index $1$ are consumed and the indices of all other prefixes decrease by $1$.

Let us consider the two possible forms for $P_i$. In both the cases if the transition involves only the context then the thesis follows trivially (case (i)). In the first case the only other possible transition is the interaction between the service invocation and the service definition in the two holes (since $a$ does not occur in $\mathcal{D}[\![\bullet, \bullet]\!]$). This leads to a process of the second form. Also, the indices are preserved. Let us consider the second case. For transitions not involving prefixes the thesis follows trivially (case (i)). Suppose now that *e.g.,* in $P'$ an output prefix is consumed (the case where the prefix is an input prefix or the consumed prefix is in $Q'$ is symmetric). Thus $P' \xrightarrow{\uparrow v} P''$ and $r \rhd P' \xrightarrow{r \rhd \uparrow v} r \rhd P''$. Since $r$ is private, this label should interact with a label of the form $r \lhd \downarrow v$. For the condition about well-formed processes this can be generated only by $r \lhd Q'$, and thanks to Lemma 3 this must be the prefix with index $1$. Thus in both $P''$ and $Q''$ prefix indices are equal to the indices in $P'$ and $Q'$ minus $1$. This proves the thesis.

We now have all the tools required to prove the correctness of the session breaking technique.

**Theorem 3.** *Let* $(\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x).P]\!], a \Rightarrow \mathcal{C}'[\![v.Q]\!]]\!]$ *be a typed process such that* $a$ *does not occur in* $\mathcal{D}[\![\bullet, \bullet]\!]$*. Suppose that there exists* $i$ *such that* $(x).P$ *and* $v.Q$ *are the* $i$*-th top-level prefixes in* $\mathcal{C}[\![(x).P]\!]$ *and in* $\mathcal{C}'[\![v.Q]\!]$*, respectively. Let* $y \notin \mathrm{fn}(P)$*,* $b \notin \mathrm{fn}(Q)$*. Then:*

$$(\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x).P]\!], a \Rightarrow \mathcal{C}'[\![v.Q]\!]]\!] \approx_{\mathrm{f}} \\ (\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x,y).y \Leftarrow P]\!], a \Rightarrow \mathcal{C}'[\![(\nu\, b)\langle v, b\rangle.b \Rightarrow Q]\!]]\!]$$

*Proof.* We show that the following is a full weak bisimulation:

$$\begin{aligned}\{&((\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x).P]\!], a \Rightarrow \mathcal{C}'[\![v.Q]\!]]\!], \\ &\quad (\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x,y).y \Leftarrow P]\!], a \Rightarrow \mathcal{C}'[\![(\nu\, b)\langle v, b\rangle.b \Rightarrow Q]\!]]\!]) \\ &((\nu\, r)\mathcal{D}[\![r \lhd \mathcal{C}[\![(x).P]\!], r \rhd \mathcal{C}'[\![v.Q]\!]]\!], \\ &\quad (\nu\, r)\mathcal{D}[\![r \lhd \mathcal{C}[\![(x,y).y \Leftarrow P]\!], r \rhd \mathcal{C}'[\![(\nu\, b)\langle v, b\rangle.b \Rightarrow Q]\!]]\!]) \\ &((\nu\, r)\mathcal{D}[\![r \lhd \mathcal{C}[\![P]\!], r \rhd \mathcal{C}'[\![Q]\!]]\!], (\nu\, r, b)\mathcal{D}[\![r \lhd \mathcal{C}[\![b \Leftarrow P]\!], r \rhd \mathcal{C}'[\![b \Rightarrow Q]\!]]\!]) \\ &((\nu\, r)\mathcal{D}[\![r \lhd \mathcal{C}[\![P]\!], r \rhd \mathcal{C}'[\![Q]\!]]\!], (\nu\, r, r')\mathcal{D}[\![r \lhd \mathcal{C}[\![r' \lhd P]\!], r \rhd \mathcal{C}'[\![r' \rhd Q]\!]]\!])\}\end{aligned}$$

where all the names and processes are universally quantified, $y \notin \mathrm{fn}(P)$, $b \notin \mathrm{fn}(Q)$, and $(x).$ and $v.$ have the same index.

Processes in the first pair can move only to processes of the same shape or to processes in the second pair because of Lemma 4. Similarly processes in the second pair can move to processes of the same form or to processes in the third pair (notice in fact that prefixes with the same index should be consumed at the same time). In the third pair the only transition that can change the structure of the processes is the invocation of service $b$ on the right, but since this is a $\tau$ step, thus the left part can answer by staying idle.

Notice that the technique above can be extended so to break protocols of services with more than one definition/invocation: simply break all of them at the same point.

## 6 Correctness of the Transformations

We now prove that the transformations presented in Section 3 are actually correct with respect to full weak bisimilarity. Interestingly, they are also transparent for process $\mathsf{A}$, i.e. $\mathsf{A}$ needs not to be changed when the transformation is applied, and binder $(\nu\,\mathsf{b})$. To prove this we show that the three equations below hold.

$$(\nu\,\mathsf{c})(\mathsf{B}\,|\,\mathsf{C}) \approx_{\mathrm{f}} (\nu\,\mathsf{c})(\mathsf{E}\,|\,\mathsf{C}) \tag{9}$$

$$(\nu\,\mathsf{c})(\mathsf{E}\,|\,\mathsf{C}) \approx_{\mathrm{f}} (\nu\,\mathsf{c})(\mathsf{F}\,|\,\mathsf{C}) \tag{10}$$

$$(\nu\,\mathsf{c})(\mathsf{F}\,|\,\mathsf{C}) \approx_{\mathrm{f}} (\nu\,\mathsf{c})(\mathsf{G}\,|\,\mathsf{D}) \tag{11}$$

The correctness of the whole transformations, *i.e.*, $\mathrm{SC} \approx_{\mathrm{f}} \mathrm{SC}' \approx_{\mathrm{f}} \mathrm{SC}'' \approx_{\mathrm{f}} \mathrm{SC}'''$ follows from closure under contexts from the equations above.

Note that the transformation of auxiliary communications (passing value $\mathsf{w}$ from $\mathsf{r}^{\mathsf{B}}$ to $\mathsf{s}^{\mathsf{B}}$ and value $\mathsf{v}$ from $\mathsf{s}^{\mathsf{B}}$ to $\mathsf{r}^{\mathsf{B}}$) into normal communications are actually correct only up to weak full bisimilarity. In fact, auxiliary communications require a few more steps, and leave behind them empty sessions and streams, which have to be garbage collected. The correctness of garbage collection is based on Equations 1 and 2.

*Proof (of Equation 9).* The proof can be easily obtained by exhibiting a bisimulation including the two processes. For lack of space we will not show it, but just highlight a few important points. The two processes can mimic each other even if the first one is non deterministic, since the nondeterminism comes from $\tau$ steps, whose order is not important, since the processes are confluent. Also, as mentioned before, garbage collection Equations 1 and 2 are used in the proof. After some steps, the two processes have evolved to:

$$(\nu\,\mathsf{s})(r \rhd \mathsf{Q}[{}^{w}\!/\!_{x}][{}^{v'}\!/\!_{y}] \mid s \lhd \mathsf{R}[{}^{w}\!/\!_{x}][{}^{w'}\!/\!_{z}][{}^{v'}\!/\!_{y}] \mid s \rhd \mathsf{S}[{}^{w}\!/\!_{x}][{}^{v}\!/\!_{y}])$$

$$(\nu\,\mathsf{s})(r \rhd (s \lhd \mathsf{R}[{}^{w}\!/\!_{x}][{}^{w'}\!/\!_{z}][{}^{v'}\!/\!_{y}]) \mid \mathsf{Q}[{}^{w}\!/\!_{x}][{}^{v'}\!/\!_{y}] \mid s \rhd \mathsf{S}[{}^{w}\!/\!_{x}][{}^{v}\!/\!_{y}])$$

respectively. These processes can be proved equivalent using structural congruence (which is included in full bisimilarity, according to Lemma 2), session independence (Equation 3) and closure under contexts.

*Proof (of Equation 10).* To prove the correctness of Equation 10 it is enough to prove $E \approx_{\mathrm{f}} F$, then the thesis follows from closure under contexts. Actually, in general we can prove

$$(\nu\,\mathsf{a})(\mathcal{C}'[\![a \Uparrow v.P]\!] \,|\, \mathcal{C}''[\![a \Downarrow (y).Q]\!]) \approx_{\mathrm{f}} \mathsf{stream}\,\mathcal{C}'[\![\mathsf{feed}\,v.P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!] \tag{12}$$

provided that neither $a$ nor $f$ occur elsewhere and $P$ and $\mathcal{C}'$ contain no feeds.

The proof shows that the three pairs below, together with a few other pairs differing from these because of $\tau$ transitions (corresponding to intermediate steps) form a full bisimulation.

$$\begin{array}{c}
((\nu\,\mathsf{a})(\mathcal{C}'[\![a \Uparrow v.P]\!] \,|\, \mathcal{C}''[\![a \Downarrow (y).Q]\!]), \mathsf{stream}\,\mathcal{C}'[\![\mathsf{feed}\,v.P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!]) \\
(\mathcal{C}'[\![P]\!] \,|\, \mathcal{C}''[\![\mathsf{stream}\,\mathbf{0}\,\mathsf{as}\,f' = \langle v\rangle\,\mathsf{in}\,f'(y).Q]\!], \\
\qquad\qquad \mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f = \langle v\rangle\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!]) \\
(\mathcal{C}'[\![P]\!] \,|\, \mathcal{C}''[\![Q]\!], \mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![Q]\!])
\end{array}$$

In each process considered in the relation, $a$, $f$, and $f'$ do not occur elsewhere, and $P$ and $\mathcal{C}'$ do not contain feeds.

The only difficult part is when the feed and read from stream are executed (and, correspondingly, the two auxiliary communications). Actually both transitions amount to $\tau$ actions.

$$\begin{array}{l}
\mathsf{stream}\,\mathcal{C}'[\![\mathsf{feed}\,v.P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!] \xrightarrow{\tau} \\
\qquad\qquad\qquad \mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f = \langle v\rangle\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!]
\end{array}$$

$$\mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f = \langle v\rangle\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!] \xrightarrow{\tau} \mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![Q[{}^{v}\!/\!_{y}]]\!]$$

The first transition is as follows (where we used the definitions of auxiliary communications in the first step).

$$\begin{array}{l}
(\nu\,\mathsf{a})\mathcal{C}'[\![a \Uparrow v.P]\!] \,|\, \mathcal{C}''[\![a \Downarrow (y).Q]\!] = \\
\qquad (\nu\,\mathsf{a})\mathcal{C}'[\![\mathsf{stream}\,a \Leftarrow v.\mathsf{feed}\,u\,\mathsf{as}\,f''\,\mathsf{in}\,f(x).P]\!] \,| \\
\qquad\qquad \mathcal{C}''[\![\mathsf{stream}\,a \Rightarrow (z)\mathsf{feed}\,z\,\mathsf{as}\,f'\,\mathsf{in}\,f(y).Q]\!] \xrightarrow{\tau}{}^{*} \\
(\nu\,\mathsf{a},\mathsf{r})\mathcal{C}'[\![\mathsf{stream}\,r \lhd \mathbf{0}\,\mathsf{as}\,f''\,\mathsf{in}\,P]\!] \,|\, \mathcal{C}''[\![\mathsf{stream}\,r \rhd \mathbf{0}\,\mathsf{as}\,f' = \langle v\rangle\,\mathsf{in}\,f(y).Q]\!] \sim_{\mathrm{f}} \\
\qquad\qquad \mathcal{C}'[\![P]\!] \,|\, \mathcal{C}''[\![\mathsf{stream}\,\mathbf{0}\,\mathsf{as}\,f' = \langle v\rangle\,\mathsf{in}\,f(y).Q]\!]
\end{array}$$

where in the last step we used Equations 1 and 2. The second transition is matched by:

$$\begin{array}{l}
\mathcal{C}'[\![P]\!] \,|\, \mathcal{C}''[\![\mathsf{stream}\,\mathbf{0}\,\mathsf{as}\,f' = \langle v\rangle\,\mathsf{in}\,f(y).Q]\!] \xrightarrow{\tau} \\
\qquad \mathcal{C}'[\![P]\!] \,|\, \mathcal{C}''[\![\mathsf{stream}\,\mathbf{0}\,\mathsf{as}\,f'\,\mathsf{in}\,Q[{}^{v}\!/\!_{y}]]\!] \sim_{\mathrm{f}} \mathcal{C}'[\![P]\!] \,|\, \mathcal{C}''[\![Q[{}^{v}\!/\!_{y}]]\!]
\end{array}$$

where we used Equation 2 again. This concludes the proof.

*Proof (of Equation 11).* It is easy to verify that $(\nu\,\mathsf{c})(\mathsf{F} \,|\, \mathsf{C})$ can be typed according to the type system for sequentiality. By considering:

$$\mathcal{D}[\![\bullet_1, \bullet_2]\!] = b \Rightarrow (x)(\mathsf{stream}\ \bullet_1\ \mathsf{as}\,f\,\mathsf{in}\,f(y).y.Q) | \bullet_2$$

$$\mathcal{C}[\![\bullet]\!] = x.\bullet \qquad P = v.(y)\textsf{feed}\, y.R$$
$$\mathcal{C}'[\![\bullet]\!] = (x)\bullet \qquad Q = (y)v'.S$$

we can apply Theorem 3 to get the thesis since prefixes $(z)$ and $w'$ have both index 2.

## 7 Conclusions

We have shown how to exploit formal techniques to define correct program transformations relating different styles of programming used in the field of service-oriented systems, namely object-oriented, session-based and request/request-response based. This allows to exploit the different techniques available in each field, and still get a system implemented using the desired technology.

In addition to that, we have illustrated the expressiveness of SSCC [10, 12], also in a field like object-oriented programming, for which it was not conceived. We have also demonstrated the benefit of working with sequential sessions, where more powerful transformations are available.

For future work we intend to investigate the possibility of extending the session breaking transformation to larger classes of systems. We are aware of the fact that parallel communications make the agreement between the client and the server on where to change session more difficult. Towards this end, a promising approach is to perform a preliminary transformation turning arbitrary sessions into sequential ones.

## References


[1] Ambler, S.W.: The Object Primer: Agile Model-Driven Development with UML 2.0. Cambridge University Press, Cambridge (2004)

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

[3] Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: SCC: a service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)

[4] Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: FMOODS 2008. LNCS, Springer, Heidelberg (to appear, 2008)

[5] Bruni, R., Lanese, I., Melgratti, H., Tuosto, E.: Multiparty sessions in SOC. In: COORDINATION 2008. LNCS, Springer, Heidelberg (to appear, 2008)

[6] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)

[7] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 63–81. Springer, Heidelberg (2006)

[8] Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)

[9] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: WSDL: Web Services Definition Language. World Wide Web Consortium (2004)

[10] Cruz-Filipe, L., Lanese, I., Martins, F., Vasconcelos, V.T., Ravara, A.: Bisimulations in SSCC. DI/FCUL TR 07–37, Department of Informatics, Faculty of Sciences, University of Lisbon (2007)

[11] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 22–138. Springer, Heidelberg (1998)

[12] Lanese, I., Martins, F., Vasconcelos, V.T., Ravara, A.: Disciplining orchestration and conversation in service-oriented computing. In: SEFM 2007, pp. 305–314. IEEE Computer Society Press, Los Alamitos (2007)

[13] Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)

[14] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading (1999)

[15] Sands, D.: Total correctness by local improvement in the transformation of functional programs. ACM Trans. Program. Lang. Syst. 18(2), 175–234 (1996)

[16] Sangiorgi, D.: Typed $\pi$-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. Theory and Practice of Object Systems 5(1), 25–34 (1999)

[17] Sangiorgi, D., Walker, D.: The $\pi$-calculus: A theory of mobile processes. Cambridge University Press, Cambridge (2001)

[18] Sensoria project web site, `http://www.sensoria-ist.eu/`

[19] Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)

[20] Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: A model of service oriented computation. In: ESOP 2008. LNCS, Springer, Heidelberg (to appear, 2008)

[21] Wirsing, M., Clark, A., Gilmore, S., Hölzl, M.M., Knapp, A., Koch, N., Schroeder, A.: Semantic-based development of service-oriented systems. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 24–45. Springer, Heidelberg (2006)