Disciplining Orchestration and Conversation in Service-Oriented Computing

Ivan Lanese
Vasco T. Vasconcelos
Francisco Martins
António Ravara

DI–FCUL TR–07–3

March 2007

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Disciplining Orchestration and Conversation in Service-Oriented Computing

Ivan Lanese* Vasco T. Vasconcelos† Francisco Martins†
António Ravara‡

March 2007

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 give an hint on how the structuring mechanisms of the language can be exploited for static analysis we present a simple type system guaranteeing the compatibility between the protocols for service definition and for service invocation, and ensuring the sequentiality of each protocol.

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 functionality 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, 3, 4, 11, 15] 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 this programming model 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 satisfy the required functionalities. These techniques should be able to deal with the different aspects of services (seen in the abstract context of global computing [13]), including their dynamic behavior.

In this realm, and abstracting the concepts of service-oriented architectures (SOA), we have identified the need for *defining and for invoking services*, for *isolating conversations* between requesters and providers of services, and for *orchestrating services*, i.e. to make use of existent

*Computer Science Department, University of Bologna, Italy
†Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal
‡Security and Quantum Information Group, Institute of Telecommunications, and Department of Mathematics, IST, Technical University of Lisbon, Portugal

services as building blocks to accomplish complex tasks. This paper proposes SSCC (Stream-based Service Centered Calculus), a calculus for modeling service-based systems, inspired by SCC [5] and Orc [17, 21].

The calculus provides primitives for *service definition and invocation* by simply composing a service name with an arbitrary process describing the service protocol. In order to *isolate conversations*, the calculus syntactically segregates value exchanges between two specific requester-provider parties (resulting from particular a service invocation) in separate *sessions*. Finally, regarding *service orchestration*, the most challenging aspect of SOA, we propose the concept of *stream* as the vehicle to compose services. Streams are shared by two processes that run in parallel; one of the parties writes in the stream, the other reads from it. The writing operation anonymous (feeds to the nearest enclosing stream), whereas reading is named; this decision allows describing complex patterns of interaction [24], while keeping the language simple and amenable to various static analysis.

SSCC departs from SCC [5] in significant ways: persistent services are replaced by the more flexible mechanism of recursion over arbitrary processes, service provider and service invocation have become symmetric, and, most importantly, the **return** primitive of SCC was replaced by the stream operations. The **return** operator makes the values produced by a service invocation available at the upper level in the session nesting (where they get mixed with the remaining outputs in the running session), making it difficult to get the results from a service invocation available where they are needed. The programming style thus induced is both difficult to use and to analyze.

Another source of inspiration was Orc [21], 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 [12, 24] 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 looked for a few basic primitives that, when composed, would yield all the required coordination patterns (trying however to be less general than, e.g., pi-calculus [20]), while ensuring a clean and structured programming style and helping in the analysis of program properties.

The calculus comes equipped with a simple type system, inspired in works on session types [14, 16, 25] and on protocol compatibility [10]. Our setting is however simpler for two reasons. Conversations within sessions do not explicitly mention session names, hence we do not require polarities on session names [14, 25]. Also, sessions cannot be passed (neither in conversations nor in streams). Introducing such a feature is straightforward. Future work includes exploring the additional flexibility introduced by such a mechanism.

The three calculi proposals described below are strongly driven by existing standards or technologies, albeit achieving different levels of abstraction. Instead we tried to isolate a few primitives allowing to describe both the orchestration and the conversation in a language amenable to static analysis.

Carbone et al. [9] 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. Such projection should be sound and complete, in the sense that the global behavior is realized as communications among endpoints. The main contribution is the identification of principles for global descriptions which induce a type-preserving endpoint projection that is sound and complete with respect to their operational semantics.

Lapadula, Pugliese, and Tiezzi introduce CŏWS [18], a processes calculus for Web service orchestration that permits to express Web services in a primitive form, with special attention for describing the interactions among Web service instances. For isolating interaction between partners, CŏWS uses message correlation, the approach of WS-BPEL [2].

Busi et al. [7] propose SOCK, a process calculus that addresses the basic mechanisms of service communication and composition, inspired in Web services specifications. In SOCK a service is defined by means of an automaton that distinguishes the set of internal actions from the set of

external actions. So, service definition captures the dependencies between services in terms of what is needed and what is offered by the service under definition. SOCK also uses message correlation to define client-server interactions.

The contributions of this paper can be summarized as follows.

Clear separation of concerns: conversation and orchestration. The calculus allows for describing service interaction and service orchestration using distinct mechanisms. The conversation between parties engaged in a service interaction is described by a series of value send/receive, isolated inside a session, while the orchestration of services is performed using the stream operations. Notice that Orc [21] lacks the conversation primitives, and that both Orc and SCC [5] feature insufficient orchestration.

Flexible programming. Service orchestration and service conversation are both easily structured in SSCC. We were able to encode all van der Aalst workflow patterns [24] (apart from the ones that require termination), in intelligible code.

A type discipline. We provide a simple static analysis tool to check the compatibility between service definition and service invocation, as well as protocol sequentiality.

The outline of the paper is as follows. The next section motivates the language via an example. Sections 3 and 4 present the syntax, semantics and type system for the calculus. Further examples, attesting the flexibility of the language and encoding of van der Aalst workflow patterns [24], can be found in Section 6. Section 7 concludes the papers and points directions for further work. The appendixes contain detailed proofs of the technical results. In Appendix A we show that the labeled transition system coincides with the reduction semantics; in Appendix B we show Subject-Reduction holds for our type system; finally, Appendix C shows that our type system is safe.

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-database-to-obtain-price>.price
```

Here, the parenthesis 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-database>.price
```

A client is supposed to meet the expectations of the service by providing a date and requesting a price. We write it as follows.

```
bologna ⇐ 31Dec2006.(price) <do-something-with-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>. As mentioned in the introduction, 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 in Orc [21] (the various abbreviations used in this paper are summarized in Figure 11.)

```
(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 (bologna in the example above). The downside it that the client does not know which hotel offers the best price; we leave it to the reader to adapt the example as required.

Using **call** and P>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 [24]).

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

Our language is equipped with a notion of types, allowing to statically filter programs that may incur in *conversation errors*, such as: the service provider expects a value and so does the client (or the client is terminated). Returning to the hotel example, we can easily see that the conversation between the service provider (⇒) and the client (⇐) is, from the point of view of the provider, as follows: expect a date; send a price; terminate. The whole process of querying the hotel database to obtain the price is opaque to the client, and does not show up in the type. We write the type for an hotel as:

```
bologna :: ?Date.!Price.end
```

The protocol with the broker is somewhat more complex, yet its interface with the client is exactly the same.

```
broker :: ?Date.!Price.end
```

All values in a stream are required to be of the same type. The type of a process is a pair describing the conversation it engages into and the values it writes into its stream. Considering the part **stream** P **as** f **in** Q of the broker example, we have that P is of type (**end**, Price), meaning that P does not engage in any interaction with the client, and that it feeds Price values into the stream. On the other hand, Q is of type (!Price.**end**, T), since it communicates a price to the client (the type of the stream is arbitrary, given that Q does not feed into its stream).

Further examples can be found in Section 6, after presenting the syntax, semantics, and type system for the language.

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
u, v	$::=$	*Values*
	a	Service name
$\mid$	unit	Unit value

Figure 1: The syntax of SSCC

3 SSCC

This section presents the syntax and the semantics of SSCC.

3.1 Syntax

Processes are built using three kinds of identifiers: *service names*, *stream names*, and *process variables*. Service names are ranged over by $a, b, \ldots$. Values, ranged over by $u, v, \ldots$ can be either service names or the unit value[1]. Values can also be used as variables (bound by value reception or read from stream), and we use $x, y, \ldots$ in this case. Stream names are ranged over by $f, g, \ldots$. Process variables are ranged over by $X, Y, \ldots$ and used to define recursive processes.

Definition 3.1 (Syntax)**.** *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 (notice that 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 symmetric (as opposed to [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 our calculus. 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 $f(x).Q'$.

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.

[1] Basic values such as integers and strings can be easily added, and will be used in the examples.

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

Figure 2: The run-time syntax of SSCC

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

Figure 3: Structural congruence

Definition 3.2 (Runtime syntax)**.** *The grammar in Figure 2 defines the* syntax of runtime processes*.*

We use $r \bowtie P$ to denote either $r \triangleleft P$ or $r \triangleright P$, and we assume that when multiple $\bowtie$ appear in the same rule they are instantiated in the same way, and that $\overline{\bowtie}$ 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 $::$.

3.2 Semantics

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 $\mathsf{rec}\, X.P$. Notation $\mathrm{fn}(P)$ (resp. $\mathrm{bn}(P)$) denotes the set of free (resp. bound) service or session names in P. We require processes to have no free process variables.

As usual, to help the definition of the semantics we use a structural congruence relation. The relation is standard, simply adding to that of the the π-calculus axioms that deal with scope extrusion for the session and the stream construct (notice that session names are static, thus there is no need of extrusion rules for them).

Definition 3.3 (Structural congruence)**.** *The rules in Figure 3, together with the commutative*

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

Figure 4: Contexts

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

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

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

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

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

Figure 5: Reduction relation

monoid rules for $(P, |, \mathbf{0})$ *and with the* α*-conversion axiom, inductively define the* structural congruence *relation on processes.*

Interactions can happen in different active contexts. Since all our interactions are binary, we find it useful to introduce also two-holes contexts, which we call double contexts.

Definition 3.4 (Active 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.

We are now in a position to introduce the reduction semantics.

Definition 3.5 (Reduction semantics)**.** *The rules in Figure 5, together with symmetric rules of* R-COMM *(swapping the processes in the two holes of double context and/or the client and the server sessions) 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. 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. Symmetric rules are used to take care of all the possible combinations of value sends/receives and client/server session. 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 a semantics based on a labeled transition system (henceforth LTS) is more convenient for proofs. Labels are as follows.

μ	::=	*Labels*
	$\uparrow v$	Value output
$\mid$	$\downarrow v$	Value input
$\mid$	$a \Rightarrow (r)$	Service definition activation
$\mid$	$a \Leftarrow (r)$	Service invocation
$\mid$	$\Uparrow v$	Stream feed
$\mid$	$f \Downarrow v$	Stream read
$\mid$	$r \rhd \uparrow v$	Server session output
$\mid$	$r \rhd \downarrow v$	Server session input
$\mid$	$r \lhd \uparrow v$	Client session output
$\mid$	$r \lhd \downarrow v$	Client session input
$\mid$	$r\tau$	Conversation step
$\mid$	τ	Internal step
$\mid$	$(a) \uparrow a$	Value extrusion
$\mid$	$(a)r \rhd \uparrow a$	Server session extrusion
$\mid$	$(a)r \lhd \uparrow a$	Client session extrusion
$\mid$	$(a) \Uparrow a$	Stream feed extrusion

Figure 6: The syntax of labels

Definition 3.6 (Transition labels)**.** *The grammar in Figure 6 defines the syntax of* labels*.*

We define an LTS in early style.

Definition 3.7 (LTS semantics)**.** *The rules in Figure 7, together with symmetric version of rule* L-SERV-COM-STREAM*, inductively define the LTS semantics of* SSCC*.*

The rules are quite simple. We just explain the meaning of labels and highlight a few more tricky points. We use μ as metavariable for labels, and extend $\mathrm{fn}(-)$ and $\mathrm{bn}(-)$ 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 the output of value v. Dually, $\downarrow v$ is the input of value v. We use $\updownarrow v$ to denote either $\uparrow v$ or $\downarrow v$, and we assume that when multiple $\updownarrow v$ appear in the same rule they are instantiated in the same way, and that $\overline{\updownarrow} v$ denotes the opposite instantiation. Also, $a \Leftarrow (r)$ and $a \Rightarrow (r)$ denote respectively the invocation and the reception of an invocation of a service a. Here r is the (bound) name of the new session to be created. 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, as we are working with 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 the name of the session 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.

$$v.P \xrightarrow{\uparrow v} P \qquad (x)P \xrightarrow{\downarrow v} P[v/x] \qquad \text{(L-SEND, L-RECEIVE)}$$

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

$$\mathsf{feed}\, v.P \xrightarrow{\Uparrow v} P \qquad f(x).P \xrightarrow{f \Downarrow v} P[v/x] \qquad \text{(L-FEED, L-READ)}$$

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

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

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

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

$$\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P|Q \xrightarrow{\mu} P'|Q} \qquad \text{(L-PAR)}$$

$$\frac{P \xrightarrow{\updownarrow v} P'}{r \bowtie P \xrightarrow{r \bowtie \updownarrow v} r \bowtie P'} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \notin \{\updownarrow v, (v) \uparrow v\} \quad r \notin \mathrm{bn}(\mu)}{r \bowtie P \xrightarrow{\mu} r \bowtie P'} \qquad \text{(L-SESS-VAL, L-SESS-PASS)}$$

$$\frac{P \xrightarrow{r \bowtie \updownarrow v} P' \quad Q \xrightarrow{r \overline{\bowtie} \updownarrow v} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{r\tau} \mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \qquad \text{(L-SESS-COM-STREAM)}$$

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

$$\frac{P \xrightarrow{r \bowtie \updownarrow v} P' \quad Q \xrightarrow{r \overline{\bowtie} \updownarrow v} Q'}{P|Q \xrightarrow{r\tau} P'|Q'} \qquad \text{(L-SESS-COM-PAR)}$$

$$\frac{P \xrightarrow{a \Rightarrow (r)} P' \quad Q \xrightarrow{a \Leftarrow (r)} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')} \qquad \text{(L-SERV-COM-PAR)}$$

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

$$\frac{P \xrightarrow{r\tau} P'}{(\nu r)P \xrightarrow{\tau} (\nu r)P'} \qquad \frac{P \xrightarrow{\mu} P',\ P \equiv Q,\ P' \equiv Q'}{Q \xrightarrow{\mu} Q'} \qquad \text{(L-SESS-RES, L-STRUCT)}$$

Figure 7: LTS semantics

Some processes, such as $r \lhd r \lhd P$, can be written using the runtime syntax, but they are not reachable from processes written in the basic syntax of Definition 3.1. We consider these processes ill-formed, and therefore make the following assumption, which is necessary for most of the results.

Assumption 1. *From now on, we will consider only processes that either are in the syntax of Definition 3.1, or can be obtained from them via reductions or LTS transitions.*

The reduction and the LTS semantics coincide. A detailed proof is in Appendix A.

Theorem 3.8. *For each P and Q, $P \to Q$ iff $P \xrightarrow{\tau} Q$.*

4 Type system

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

Figure 8: The syntax of types

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

Figure 9: Complement of a protocol

SSCC have been developed keeping in mind typing issues. We present here a simple type system to show the kind of properties (e.g., protocol compatibility) that our language allows to express, but we do not go into refined typing techniques for proving, e.g., deadlock freedom. This will be the topic of future work.

Definition 4.1 (Types)**.** *The grammar in Figure 8 defines the* syntax of types*.*

The term Unit denotes the only basic type [2], 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 $\{T\}$ 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 [22], not distinguishing between a type $\mathsf{rec}\ X.U$ and its unfolding $T[\mathsf{rec}\ X.U/X]$. We are interested on *contractive* (not including subterms of the form $\mathsf{rec}\, X.\mathsf{rec}\, X_1 \ldots \mathsf{rec}\, X_n.X$) types only [22].

Since we are interested in protocol compatibility, we need to find whether two protocols are complementary. To this end 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,

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

where Γ is a map with entries $a\colon T$, $r\colon T$, $f\colon \{T\}$, and $X\colon (U, T)$. The typing system is defined by the rules in Figure 4.

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

[2] The extension with, say, integers and strings is trivial.

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

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

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

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

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

$$\Gamma \vdash \mathbf{0}\colon (\mathsf{end},T) \qquad \frac{\Gamma, n\colon \_ \vdash P\colon (U,T)}{\Gamma \vdash (\nu n)P\colon (U,T)}$$
(T-NIL, T-RES)

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

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

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

$$\frac{}{\Gamma, X\colon (U,T) \vdash X\colon (U,T)} \qquad \frac{\Gamma, X\colon (U,T) \vdash P\colon (U,T)}{\Gamma \vdash \mathsf{rec}\, X.P\colon (U,T)}$$
(T-VAR, T-REC)

Figure 10: Typing rules

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. Choices in protocols can be added following the ideas in [16]. Notice that this does not forbid, e.g., to have two concurrent service invocations, since sequentiality is only enforced in protocols.

The type system enjoys subject reduction and prevents erroneous behaviours in typable processes (a result commonly known as type safety). Appendices B and C contain the proofs of these results.

Theorem 4.2 (Subject reduction)**.** *Let P be a process such that $\Gamma \vdash P\colon (U,T)$ and $P \to P'$. Then $\Gamma \vdash P'\colon (U,T)$.*

Protocols, in general, are not exempt from errors. 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 $r\colon [\mathsf{end}]$ 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.

$$
\begin{array}{rcl}
\textbf{call}\ \mathsf{a}(\mathsf{x1}\ ,\ldots,\ \mathsf{xn}) & \triangleq & \mathsf{a} \Leftarrow \mathsf{x1}\ldots\mathsf{xn}.(\mathsf{y})\ \textbf{feed}\ \mathsf{y} \\
\mathsf{P} >^{\mathsf{n}} \mathsf{x1}\ \ldots\ \mathsf{xn} > \mathsf{Q} & \triangleq & \textbf{stream}\ \mathsf{P}\ \textbf{as}\ \mathsf{f}\ \textbf{in}\ \ \mathsf{f}(\mathsf{x1}\ )\ldots\ \mathsf{f}(\mathsf{xn})\mathsf{Q} \\
\mathsf{P} > \mathsf{x} > \mathsf{Q} & \triangleq & \textbf{stream}\ \mathsf{P}\ \textbf{as}\ \mathsf{f}\ \textbf{in}\ \ \textbf{rec}\ \mathsf{X}.\mathsf{f}(\mathsf{x})(\mathsf{P} \mid \mathsf{X}) \\
\mathsf{a}\ \textbf{*} \Rightarrow \mathsf{P} & \triangleq & \textbf{rec}\ \mathsf{X}.\ \mathsf{a} \Rightarrow (\mathsf{P} \mid \mathsf{X}) \\
\textbf{if}\ \ \mathsf{b}\ \textbf{then}\ \mathsf{P} & \triangleq & \mathsf{b} \Leftarrow (\mathsf{x})(\mathsf{y})\ \mathsf{x} \Leftarrow \textbf{feed unit} >^{\mathbf{1}} > \mathsf{P} \\
\textbf{if}\ \ \neg\mathsf{b}\ \textbf{then}\ \mathsf{P} & \triangleq & \mathsf{b} \Leftarrow (\mathsf{x})(\mathsf{y})\ \mathsf{y} \Leftarrow \textbf{feed unit} >^{\mathbf{1}} > \mathsf{P} \\
\textbf{if}\ \ \mathsf{b}\ \textbf{then}\ \mathsf{P}\ \textbf{else}\ \mathsf{Q} & \triangleq & \textbf{if}\ \ \mathsf{b}\ \textbf{then}\ \mathsf{P} \mid \ \textbf{if}\ \ \neg\mathsf{b}\ \textbf{then}\ \mathsf{Q} \\
[?\mathsf{T1}\ldots?\mathsf{Tn}.!\mathsf{T}.\textbf{end}] & \triangleq & \mathsf{T1} \to \ldots \to \mathsf{Tn} \to \mathsf{T} \\
[!\,\mathsf{T}.\textbf{end}] & \triangleq & \varepsilon \to \mathsf{T} \\
\mathsf{Bool} & \triangleq & [![\,\textbf{end}\,].![\,\textbf{end}].\textbf{end}]
\end{array}
$$

Figure 11: Derived constructs

Theorem 4.3 (Type Safety)**.** *Let P be a typable process. Then P has no subterm of the following forms.*

Protocol:

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

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

Sequentiality:

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

where in all cases $\mathcal{D}[\![,]\!]$ does not contain sessions around the $\bullet$.

5 Further examples

This section explores examples that highlight the versatility of SSCC. Services in SSCC are ephemeral: they don't survive invocation. Recursion can be used to provide for persistent services: a service $\mathsf{a} \Rightarrow \mathsf{P}$ can be made persistent by writing instead $\textbf{rec}\ \mathsf{X}.\mathsf{a} \Rightarrow (\mathsf{P} \mid \mathsf{X})$, which we abbreviate to $\mathsf{a}\ \textbf{*} \Rightarrow \mathsf{P}$. Figure 11 gathers all the abbreviations used in the paper.

The first example shows that naming streams can be handy. *Fork-join* is a pattern that 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 :: ?(!T1.end).?(!T2.end).!T1.!T2.end
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 in Orc [21, 17], 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). Notice however that the declared type makes sure that services a and b produce each a single value.

The second example describes an idiom where for each value x produced by a process P, a second process Q is started. If process P produces its values by feeding into its stream, then, in the process below a new copy of process Q is spawned for each value read from the stream. Process

```
stream P as f in rec X.f(x).(Q | X)
```

can be abbreviated to $P > x > Q$ (x can be dropped if it does not occur in Q), so that a service that reads news from sites CNN and BBC and emails each to a given address can be written as:

```
email-news :: ?Address.end
email-news *⇒ (a)((call CNN | call BBC) > x > email ⇐ a.x)
```

The example and the short syntax is again from Orc. In this case we are faithful to the Orc semantics.

The third example describes stateful services, that is services that produce values influenced over time by other computations. 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. Here we concentrate on a rather distilled example: a one place buffer-cell with read and write operations. Inspired in the encoding of objects in the pi-calculus [23], 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 :: !Int.end
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 :: ?Int.end
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 :: !(!Int.end).!(?Int.end).end
cell *⇒ (ν buffer, get, set).get.set.(buffer ⇒ 0 |
  get *⇒ call buffer >1 v > (v | buffer ⇒ v) |
  set *⇒ call buffer >1 > (w)(buffer ⇒ w))
```

The last example simulates buffers that can be read and written on the same side of the **stream** construct, thus overcoming the apparent limitation of anonymous buffer writing. A back service relays the values from the right to the left part of a stream construct, where they are fed into the stream. The technique is embodied in the *interleaved parallel routing* pattern of van der Aalst [24], where 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, as witnessed by their types. Contrary to Orc [12], SSCC is expressive enough to describe the pattern within the language.

```
interleave :: ?(!T1.end)...?(!Tn.end).end
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))
```

6 Programming workflow patterns in SSCC

In this section we illustrate the expressiveness of SSCC by implementing the Workflow Patterns (WP) from Van der Aalst et. al [24]. This allows to contrast our approach with SCC and Orc [12], which have similar aims. While Workflow Patterns are an interesting benchmark, they are aimed at workflow description languages, not at calculi for SOC. For these reason some of the patterns are not meaningful (WP11) in our contexts, while others are redundant (e.g., WP12 is analogous to WP2, since process calculi can obviously handle multiple instances). Also, some patterns require the ability to kill processes, which has not yet been introduced in SSCC, and thus are out of our possibilities. On the contrary WPs consider only "activities", i.e., services that receive one value and give back one result, while our calculus can model complex protocols.

All patterns (in reference [24]) are described as services; we also present their types. Those that have multiple entry points (the various merges, for example) are modeled with a vector of boolean values, describing which services should be invoked.

An *activity* is a service that writes at most a value on the client side (replies at most a value). The simplest activity is the null service.

```
nullService :: ε → Unit
nullService *⇒ unit
```

Most of the patterns below allow definitions in SSCC that do not directly use neither the stream operations (**stream**, **feed**, and f(x)P) nor recursion. To allow a comparison we also show how the patterns can be implemented in SCC. Services in SCC have always one parameter: we exploit it as first input for the server if the server protocol should start with an input, and we assume it is unused otherwise and use **unit** as invocation value.

In what follows we give a brief description of each workflow pattern and present an illustrative example, both taken from [24].

WP1: Sequence

"An activity in a workflow process is enabled after the completion of another activity in the same process. Example: an insurance claim is evaluated after the client's file is retrieved."

```
seq :: (ε → T1) → (ε → T2) → T2
seq *⇒ (a1)(a2) call a1 >1> call a2 >1> x > x
```

In Orc the implementation is similar. In SCC the most direct implementation is:

```
seq ⇒ (a1)(a2) a2 ⇐ a1 ⇐ unit
```

This implementation is fine for activities (actually here a2 is invoked with the value from a1 rather than of **unit**), but if a1 is not an activity then a2 is called for each value returned by a1, and this is not the expected semantics. In SSCC this can not happen since the remaining values returned by a1 stay forever in the stream. If one wants to enforce correct behavior one should write:[3]

```
seq ⇒ (a1)(a2)(νr)(r ▷ (a1 ⇐ unit | (res) return res) |
                   r̄ ▷ (v) a2 ⇐ unit)
```

Also the problem of which value to use for invoking a2 is solved in SSCC. Notice that the most intuitive encoding of this pattern in SCC uses a conversation (a process of the form $r \triangleright P|\overline{r} \triangleright Q$), which we view as runtime syntax in SSCC. However sessions can be avoided also in SCC using "fake" service invocations and definitions (however service definitions stay there afterward since they are persistent).

[3]This can be done also by type checking the protocol for a1: this feature is not yet available in SCC but it can be easily transferred there.

WP2: Parallel Split

"A point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order. Example: after registering an insurance claim two parallel subprocesses are triggered: one for checking the policy of the customer and one for assessing the actual damage."

Parallel composition is built-in. The same in SCC and in Orc.

WP3: Synchronization

"A point in the workflow process where multiple parallel subprocesses/activities converge into one single thread of control, thus synchronizing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed only once. Example: insurance claims are evaluated after the policy has been checked and the actual damage has been assessed."

```
sync :: (ε → T) → ... → (ε → T) → Unit
sync *⇒ (a1)...(an)(call a1 | ... | call an) >^n> unit
```

Orc uses the **where** and SCC uses sessions (or "fake services"):

```
sync ⇒ (a1)...(an)(νr)(r ▷ (a1 ⇐ unit | ... | an ⇐ unit) |
                        r̄ ▷ (x1)...(xn) return unit)
```

WP4: Exclusive Choice

"A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen. Example: based on the workload, a processed tax declaration is either checked using a simple administrative procedure or is thoroughly evaluated by a senior employee."

```
xor :: Bool → (ε → T) → (ε → T) → T
xor *⇒ (b)(a1)(a2) if b then call a1 >^1 x > x else call a2 >^1 x > x
```

Notice that if-then-else cannot be typed with the current system unless both branches have the empty protocol (since they occur in parallel). One should add a dedicated rule to exploit the knowledge that only one of the branches is actually executed.

In SCC we can implement true and false in a similar way. Then we have:

```
if b then P = (ν s) s ▷ b {(−) (x)(y) return x} ⇐ unit
              | s̄ ▷ (x1) (ν r) r ▷ x1 {(−)return unit} ⇐ unit
                                | r̄ ▷ (z) P
if ¬b then P = (ν s) s ▷ b {(−) (x)(y) return y} ⇐ unit
              | s̄ ▷ (x1) (ν r) r ▷ x1 {(−)return unit} ⇐ unit
                                | r̄ ▷ (z) P
```

Notice that while in SSCC feeds from P are not intercepted by the if context, in SCC the returns are lost since P is executed inside a subsession. To forward the results to the caller extra programming effort is required. Actually since P can not be executed at top level (since in order to start it when a trigger coming from a subsession is received, the trigger should be transmitted using a return, that either goes to the other side, or executes P inside a session) a forward of values is needed, but we are able to specify only a finite amount of forwarding. Thus if P can give back an unbounded number of replies this can not be programmed. Anyway in the following example we always suppose to have the if with forwarding of the results. Since we deal only with activities (one result) then it can be implemented.

The if-then-else is as in SSCC.

```
xor ⇒ (b)(a1)(a2) if b then a1 ⇐ unit else a2 ⇐ unit
```

Because of the above observation the xor in SCC (with the above implementation of if) gives back no value.

WP5: Simple Merge

"A point in the workflow process where two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel. Example: after the payment is received or the credit is granted the car is delivered to the customer."

```
merge :: Bool → (ε → T) →...→ Bool → (ε → T) → Unit
merge *⇒ (b1)(a1)...(bn)(an)
         (if b1 then call a1 | ... | if bn then call an) >¹> unit
```

More in line with van der Aalst [24] than patterns in Orc, since the fact that only some of the activities are activated is modeled. Notice that $>^1>$ can be replaced by $>>$, given the assumptions.

WP6: Multi-Choice

"A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen. Example: after executing the activity *evaluate_damage* the activity *contact_fire_department* or the activity *contact_insurance_company* is executed. At least one of these activities is executed. However, it is also possible that both need to be executed."

```
multiChoice *⇒ (b1)(a1)...(bn)(an)
               (if b1 then call a1 >¹ x > x | ... |
                if bn then call an >¹ x > x)
```

Not an activity (multiple replies). Not typable since there are many parallel outputs.
A similar implementation is possible in Orc and in SCC.

WP7: Synchronizing Merge

"A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete. Example: extending the example of WP6 (Multi-choice), after either or both of the activities *contact_fire_department* and *contact_insurance_company* have been completed (depending on whether they were executed at all), the activity *submit report* needs to be performed (exactly once)."

```
syncMerge:: (ε → Bool) → (ε → Unit) →...→
            (ε → Bool) → (ε → Unit) → Unit
syncMerge *⇒ (b1)(a1)...(bn)(an)
            call sync(ifSignal_b1_a1 ,... , ifSignal_bn_an) >ⁿ > unit
ifSignal_bi_ai:: ε → Unit
ifSignal_bi_ai ⇒ IfSignal(bi, call ai >¹ x > x)
```

where

```
IfSignal(b,P) = if b then P else unit
```

Similar to Orc and SCC.

WP8: Multi-Merge

"A point in a workflow process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started *for every activation of every incoming branch*. Example: two activities *audit_application* and *process_application* running in parallel which should both be followed by an activity *close_case*."

Replace, in WP5, $>^1 >$ by $> >$. Not an activity (multiple replies). Not typable, since there are many outputs in parallel (remember that $> >$ unfolds in a recursion with a parallel composition inside).

Similar to the Orc implementation. In SCC we can use the technique of WP1. Notice also that now the behavior of the synchronization is the expected one (one instance is launched for each value).

WP9: Discriminator

"The discriminator is a point in a workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and "ignores" them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again (which is important otherwise it could not really be used in the context of a loop). Example: to improve query response time, a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored."

```
discriminator :: (ε → T) →...→ (ε → T) → Unit
rec X. discriminator ⇒ (a1)...(an)
                       stream call a1 | ... | call an as f in
                       f(x1).unit.f(x2)...f(xn).X
```

In SCC, we can not control the point where the service discriminator becomes available again.

```
discriminator ⇒ (a1)...(an)
                (ν r) r ▷ a1 ⇐ unit | ... |an ⇐ unit
                      r̄ ▷ (x1).return unit.(x2)...(xn)
```

Here the Orc implementation supposes the existence of a basic site S, with methods put and get, acting as a buffer. This site can not be described in Orc (Orc does not deal with site programming). We think that sites should deal only with computation, while all the coordination should be done at the coordination language level. This implementation fails to satisfy this separation of concerns. We are not aware of better implementations in Orc.

WP10: Arbitrary Cycles

"A point in a workflow process where one or more activities can be done repeatedly."

Arbitrary cycles can be obtained via mutual invocations among services.

We show here how an example of structured cycle can be programmed: call service a while service c returns true.

```
while :: (ε → Bool) → (ε → T) → Unit
while *⇒ (c)(a) call c >^1 b >
                    IfSignal(b, call a >^1 > call while (c,a)) >^1 x > x
```

Programmed as in Orc.

WP11: Implicit Termination

"A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock)."

This is not a real pattern. Processes should be terminated only when they have finished their activity, not when a final state is reached by one of their components. This is what happens in our case and in SCC. This is the standard thing in calculi, as opposed to workflow managers.

WP12: Multiple Instances without Synchronization

"Within the context of a single case (i.e., workflow instance) multiple instances of an activity can be created, i.e., there is a facility to spawn new threads of control. Each of these threads of control is independent of other threads. Moreover, there is no need to synchronize these threads. Example: a customer ordering a book from an electronic bookstore such as Amazon may order multiple books at the same time. Many of the activities (e.g., billing, updating customer records, etc.) occur at the level of the order. However, within the order multiple instances need to be created to handle the activities related to one individual book (e.g., update stock levels, shipment, etc.). If the activities at the book level do not need to be synchronized, this pattern can be used."

Multiple instances of the same service can be executed concurrently without any particular problem. Thus, this is the same as WP2. The same in SCC.

WP13: Multiple Instances with a Priory Design Time Knowledge

"For one process instance an activity is enabled multiple times. The number of instances of a given activity for a given process instance is known at design time. Once all instances are completed some other activity needs to be started. Example: the requisition of hazardous material requires three different authorizations."

```
sync_n :: (ε → T) → Unit
sync_n *⇒ (a) call sync (a,...,a) >1 x > x
```

There are n arguments to service sync. The number of instances (calls to) of service a is known to be n.

In SCC:

```
sync_n ⇒ (a) sync {a. ... .a.(x) return x} ⇐ a
```

WP14: Multiple Instances with a Priory Runtime Knowledge

"For one case an activity is enabled multiple times. The number of instances of a given activity for a given case varies and may depend on characteristics of the case or availability of resources, but is known at some stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started. Example: when booking a trip, the activity *book_flight* is executed multiple times if the trip involves multiple flights. Once all bookings are made, the invoice is to be sent to the client."

We treat this case as particular example of WP15. See below for the discussion.

WP15: Multiple Instances without a Priory Runtime Knowledge

"For one case an activity is enabled multiple times. The number of instances of a given activity for a given case is not known during design time, nor is it known at any stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started. The difference with WP14 is that even while some of the instances are being executed or already completed, new ones can be created. Example: for the processing of an insurance claim, zero or more eyewitness reports should be handled. The number of eyewitness reports may vary. Even when processing eyewitness reports for a given insurance claim, new eyewitnesses may surface and the number of instances may change."

Invoke service a as long as service c replies true. Instances are executed in parallel: the first instance is launched in parallel with parloop_c_a. Termination of an instance is checked together with the termination of the parloop launched together.

```
parloop_c_a :: ε → Unit
parloop_c_a *⇒ call c >1 b >
              IfSignal(b, call sync(a, parloop_c_a)) >1 x > x
```

For simplicity we have chosen a loop service specific for a and c. To write a generic loop service that accepts two parameters (c and a) we have to customize sync to invoke services with parameters. We leave the exercise to the reader.

Similar implementations can be done in Orc and in SCC.

As far as WP 14 is concerned, the main choice is how to represent the runtime knowledge about the required number of instances to be executed, i.e. how to represent state. Possibilities include taking advantage of the number of values in a stream, of the number of instances of a service available, or of the number of values in a protocol.

WP16: Deferred Choice

"A point in the workflow process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment. However, in contrast to the AND-split, only one of the alternatives is executed. This means that once the environment activates one of the branches the other alternative branches are withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible. Example: after receiving products there are two ways to transport them to the department. The selection is based on the availability of the corresponding resources. Therefore, the choice is deferred until a resource is available."

Requires a means to kill unwanted computations (cf. FIRST in [12]).

WP17: Interleaved Parallel Routing

"A set of activities is executed in an arbitrary order: each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time). Example: the Navy requires every job applicant to take two tests: *physical_test* and *mental_test*. These tests can be conducted in any order but not at the same time."

See the last example of Section 5. Ephemeral services are crucial here. This cannot be implemented in SCC (even with the kill, since the kill can delete the resource but not atomically with the service invocation, thus concurrent invocations may succeed).

Orc here exploits a basic site M implementing a lock with methods acquire and release. This site cannot be programmed inside Orc (see discussion in WP9).

7 Conclusion and further work

SSCC is a typed language aiming at flexibly describing *services*, *conversation*, and *orchestration*, with a restricted set of constructors. The expressivity of the language is witnessed by the simple implementation of all workflow patterns in [24] (except for the ones that require some form of explicit process termination).

There is a close relationship between the calculus here proposed (and of SCC [5] without session termination) and the pi-calculus with session types [16, 25]. However, the emphasis of the pi-calculus with sessions is on conversations and not on orchestration. Rather than using the full pi-calculus as a coordination tool, our approach is more constrained (for example, streams are never communicated), what should help analysis.

Future work includes the incorporation of termination and compensation primitives to model long-running transactions. Some existent process calculi proposals include basic primitives to interrupt running conversations and program compensations [5, 6, 7, 8, 18, 19].

Also, we have clearly separated in the language three potential sources of deadlock—service invocation, protocols (that is conversations within sessions), and streams—hoping to establish a basis suitable to develop further analysis tools. Finally, the labeled transition system here developed may be used as basis to develop equivalences and logics for the world of services.

Acknowledgments. This work was partially supported by the EU FEDER and the Portuguese FCT (via the Center for Logic and Computation and the project SpaceTimeTypes, POSC/EIA/-55582/2004), and 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, 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 *Proc. of WS-FM'06*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[6] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. of POPL'05*, pages 209–220. ACM Press, 2005.

[7] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *Proc. of ICSOC'06*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[8] M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *Proceedings of 25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2005.

[9] M. Carbone, K. Honda, N. Yoshida, and R. Milner. Structured communication-centred programming for web services. In *Proc. of ESOP'07*, Lecture Notes in Computer Science. Springer, 2007. To appear.

[10] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for web services. In *Proc. of WS-FM'06*, volume 4184 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2006.

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

[12] W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in orc. In *Proc. of COORDINATION'06*, volume 4038 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2006.

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

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

[15] 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.

[16] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proc. of ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 22–138. Springer, 1998.

[17] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *Proc. of CONCUR'06*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006.

[18] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP'07*, Lecture Notes in Computer Science. Springer, 2007. To appear.

[19] M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *Proc. of WS-FM'06*, volume 4184 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2006.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[21] 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.

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

[23] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Proc. of TPPP'94*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer, 1995.

[24] W. van der Aalst, B. Hofstede, and A. Kiepuszewski. Advanced workflow patterns. In *Proc. of CoopIS'00*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer, 2000.

[25] 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 *Proc. of 1st International Workshop on Security and Rewriting Techniques*, ENTCS. Elsevier, 2006.

A Equivalence between LTS and reduction semantics

Theorem A.1. *For each P and Q, $P \to Q$ if and only if $P \xrightarrow{\tau} Q$.*

Proof. To prove the forward implication we have to show that for each reduction we have a corresponding derivation with label τ. The proof is by induction on the length of the derivation of the reduction.

We have a case analysis for the different rules.

R-comm: From rule L-SEND we have $v.P \xrightarrow{\uparrow v} P$ while from rule L-RECEIVE we have $(x)Q \xrightarrow{\downarrow v} Q[v/x]$. By case analysis one can see that both these labels can traverse all active contexts satisfying the side conditions using rules L-RES (actually it is not required that the context does not bind r), L-PAR, L-STREAM-PASS-P and L-STREAM-PASS-Q, thus the two contexts have the same transitions. Then using rule L-SESS-VAL one can derive transitions with labels $r\triangleright \uparrow v$ and $r\triangleleft \downarrow v$. Again these transitions can traverse active contexts satisfying the side conditions (same rules as before). When the toplevel double context is reached then there are two cases according to its form. If the topmost operator is a parallel composition then rule L-SESS-COM-PAR is used, otherwise rule L-SESS-COM-STREAM is used. This allows to derive a transition with label $r\tau$. Finally rule L-SESS-RES can be applied to have the desired transition.

R-sync: The structure of the derivation is similar to the one above. Using rule L-CALL one can derive $a \Leftarrow P \xrightarrow{a \Leftarrow (r)} r \triangleleft P$ and using rule L-DEF one can derive $a \Rightarrow Q \xrightarrow{a \Rightarrow (r)} r \triangleright Q$. Again, these labels can traverse all active contexts satisfying the side conditions (using rules L-RES, L-PAR, L-STREAM-PASS-P, L-STREAM-PASS-Q and L-SESS-PASS). When the toplevel double context is reached we have three cases corresponding to rules L-SERV-COM-PAR, L-SERV-COM-STREAM and its symmetric.

R-feed: Using rule L-FEED one can derive $\mathsf{feed}\, w.P \xrightarrow{\Uparrow w} P$. This label can traverse all contexts satisfying the side conditions (using the same rules as above). When the stream context is reached one can apply rule L-STREAM-FEED.

R-read: Using rule L-READ one can derive $f(x).Q \xrightarrow{f \Downarrow w} Q[w/x]$. This can traverse each context satisfying the side conditions (note that L-STREAM-PASS-Q can be applied if the stream name is different from f, while if it is f then the context binds f against the hypothesis). The derivation can be concluded using rule L-STREAM-CONS.

R-cong: It is enough to check that all the active contexts are transparent to labels τ.

R-str: The same structural congruence can be used also in the LTS, thus there is nothing to prove.

To prove the opposite direction one has to check that for each way to produce τ in the LTS there is a corresponding reduction.

Again we have an induction on the length of the derivation, and a case analysis according to the last used rule.

L-send, L-receive, L-call, L-def, L-feed, L-read: these rules can not produce τ, thus the theorem is trivially true.

L-sess-val, L-sess-com-stream, L-sess-com-par, L-extr: these rules can not produce τ too, thus the theorem is trivially true.

L-stream-pass-P, L-stream-pass-Q, L-par, L-sess-pass, L-res: these rules can produce transitions with label τ only if the premise has a transition with label τ, thus they can be simulated by context closure.

L-stream-feed: This rule requires that the argument has a transition with label $\Uparrow v$. This label can be produced only by $\mathsf{feed}\, v.P \xrightarrow{\Uparrow v} P$. Notice also that this label is propagated exactly by the contexts that satisfy the side conditions of rule R-FEED. Thus R-FEED can be used to generate the required reductions.

L-stream-cons: This rule requires that the argument has a transition with label $f \Downarrow v$. This label can be produced only by $f(x).P \xrightarrow{f \Downarrow v} P[v/x]$. Notice also that this label is propagated exactly by the contexts that satisfy the side conditions of rule R-READ. Thus R-READ can be used to generate the required reductions.

L-serv-com-stream: This rule requires that the two arguments have transitions with labels $a \Rightarrow (r)$ and $a \Leftarrow (r)$ respectively (the opposite for the symmetric, but the proof is similar). These labels can be produced only by $a \Rightarrow P \xrightarrow{a \Rightarrow (r)} r \rhd P$ and $a \Leftarrow P \xrightarrow{a \Leftarrow (r)} r \lhd P$ respectively and propagated by the contexts satisfying the side conditions of rule R-SYNC concerning $\mathcal{D}[\![,]\!]$ (which are actually applied to the unary subcontexts, since the toplevel context is a the stream in rule L-SERV-COM-STREAM). Thus the term has the structure required to apply rule R-SYNC.

L-serv-com-par: the proof is analogous to the one above, with the only difference that now the toplevel double context is a parallel composition (and structural congruence can be used to swap the arguments if needed).

L-sess-res: This rule requires the argument to have a transition with label $r\tau$. Two cases are possible: either it is produced by rule L-SESS-COM-STREAM or by rule L-SESS-COM-PAR. Let us consider the first case.

Rule L-SESS-COM-STREAM requires that the two arguments have transitions with labels $r \bowtie\!\updownarrow v$ and $r\overline{\bowtie}\!\updownarrow v$ respectively. These labels can be produced only by two sessions r with opposite polarities applied to labels $\uparrow v$ and $\downarrow v$ respectively, and propagated by the contexts satisfying the side conditions of rule R-COMM concerning $\mathcal{D}[\![,]\!]$ (which are actually applied to the unary subcontexts, since the toplevel context is the stream in rule L-SESS-COM-STREAM).

Labels $\uparrow v$ and $\downarrow v$ can be produced only by $v.P \xrightarrow{\uparrow v} P$ and $(x)Q \xrightarrow{\downarrow v} Q[v/x]$ respectively and propagated by the contexts satisfying the side conditions of rule R-COMM concerning $\mathcal{C}[\![]\!]$ and $\mathcal{C}'[\![]\!]$. Thus the term has the structure required to apply rule R-COMM.

The proof for rule L-SESS-COM-PAR is analogous to the one above, with the only difference that now the toplevel double context is a parallel composition.

L-struct: structural congruence is available also for reduction semantics, thus the proof is trivial.

□

B Subject reduction

Lemma B.1. *For each session r and each process P, at most two session constructs appear in P, and if they are exactly two then they are not nested, they have opposite polarities and there is a restriction binding them. These are the only allowed occurrences of r in P.*

Proof. By induction on the length of the computation creating P. The thesis is true for computations of length 0 (sessions do not appear in the syntax). When a session is created its name is bound, thus it is checked that it is different from other names, thus different service invocations can not create sessions with the same name. A service invocation can create at most a pair of non nested sessions with opposite polarities (and if two are created then a restriction for the session name is added too), and no other occurrences of the session name are allowed. □

Lemma B.2 (Substitution lemma)**.** *Let* $\Gamma, x\colon T' \vdash P\colon (U,T)$*. If* $\Gamma \vdash v\colon T'$ *then* $\Gamma \vdash P[{}^{v}\!/_{x}]\colon (U,T)$*.*

Proof. By induction on the typing proof. All the cases are simple. □

Lemma B.3. *If* $\Gamma \vdash P\colon (\mathsf{end},T)$ *then* P *has no transitions of the form* $P \xrightarrow{mu} P'$ *with* $\mu \in \{\uparrow v, \downarrow v, (v)\uparrow v\}$*.*

Proof. The only way to have such transitions is to have processes of the form $\mathcal{C}[\![v.P]\!]$ or $\mathcal{C}[\![(x)P]\!]$ where $\mathcal{C}[\![-]\!]$ is composed only by streams, parallel compositions and restrictions. Let us consider four cases according to the toplevel operator in $\mathcal{C}[\![]\!]$.

In the base case we have to use rule T-SEND or T-RECEIVE. These rules do not allow (end,T) as resulting type.

In the case of stream we have to use rule T-STREAM-R or T-STREAM-L. We consider just the first case, the second being symmetric. The stream has type (end,T) only if the second argument has the same type. Since also the first argument has type (end,T') we know by induction that neither of the arguments can do the communication transitions, thus P cannot do them too.

In the case of parallel composition we have to use rule T-PAR-R or T-PAR-L. We consider the first case, the second one being symmetric. P has type (end,T) only if the second argument has the same type. Since also the first argument has type (end,T) we know by induction that neither of the arguments can do the communication transitions, thus P cannot do them too.

In the case of restriction we have to use rule T-RES. P has type (end,T) only if the restricted process has the same type. We know by induction that the argument cannot do the communication transitions, thus P cannot do them too. □

Lemma B.4 (Weakening)**.** *If* $\Gamma \vdash P\colon (U,T)$ *and* $n \notin \mathrm{fn}(P)$ *then* $\Gamma, n\colon T' \vdash P\colon (U,T)$*, for all* T'*.*

Proof. Simple, by induction on the derivation of the typing judgement. □

Lemma B.5 (Strengthening)**.** *If* $\Gamma, n : T' \vdash P\colon (U,T)$ *and* $n \notin \mathrm{fn}(P)$ *then* $\Gamma \vdash P\colon (U,T)$*.*

Proof. Simple, by induction on the derivation of the typing judgement. □

Lemma B.6 (Subject congruence)**.** *If* $\Gamma \vdash P\colon (U,T)$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q\colon (U,T)$*.*

Proof. It is enough to show that structural congruent terms can be given the same type using the same assumptions. This is enough to show this for the LHS and the RHS for each structural congruence rule, then the thesis follows by induction (the congruence axioms are simple). All the cases but the one for recursion are easy. We show just this case. Suppose that $\Gamma \vdash \mathsf{rec}\, X.P\colon (U,T)$. Then by hypothesis $\Gamma, X\colon (U,T) \vdash P\colon (U,T)$. By structural induction on P we can prove that if $\Gamma, X\colon (U,T) \vdash P\colon (U,T)$ then $\Gamma \vdash P[{}^{\mathsf{rec}\, X.P}\!/_{X}]\colon (U,T)$. This holds for the case of $P = X$ and is preserved by all the contexts (notice in fact that the assumptions about different occurrences of the same variable are compatible). The proof is similar in the opposite direction. □

Let $\Gamma[{}^{[U']}\!/_{r}]$ denote the substitution on Γ of $[U']$ for $\Gamma(r)$.

Theorem B.7. *Let* P *be a process such that* $\Gamma \vdash P\colon (U,T)$*. Then:*

- *if* $P \xrightarrow{\uparrow v} P'$ *then* $U = !T'.U'$*,* $\Gamma \vdash v\colon T'$ *and* $\Gamma \vdash P'\colon (U',T)$*;*
- *if* $P \xrightarrow{(v)\uparrow v} P'$ *then* $U = !T'.U'$ *and* $\Gamma, v\colon T' \vdash P'\colon (U',T)$*;*
- *if* $P \xrightarrow{\downarrow v} P'$ *then* $U = ?T'.U'$ *and* $\Gamma, v\colon T' \vdash P'\colon (U',T)$*;*

- *if* $P \xrightarrow{a \Leftarrow (r)} P'$ *then* $\Gamma \vdash a\colon [U']$ *and* $\Gamma, r\colon [U'] \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{a \Rightarrow (r)} P'$ *then* $\Gamma \vdash a\colon [U']$ *and* $\Gamma, r\colon [U'] \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{\Uparrow v} P'$ *then* $\Gamma \vdash v\colon T$ *and* $\Gamma \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{(v) \Uparrow v} P'$ *then* $\Gamma, v\colon T \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{f \Downarrow v} P'$ *then* $\Gamma \vdash f\colon \{T\}$ *and* $\Gamma, v\colon T \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{r \rhd \uparrow v} P'$ *then* $\Gamma \vdash r\colon [!T'.U']$, $\Gamma \vdash v\colon T'$ *and* $\Gamma[{}^{[U']}\!/_{r}] \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{(v) r \rhd \uparrow v} P'$ *then* $\Gamma \vdash r\colon [!T'.U']$ *and* $\Gamma[{}^{[U']}\!/_{r}], v\colon T' \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{r \rhd \downarrow v} P'$ *then* $\Gamma \vdash r\colon [?T'.U']$ *and* $\Gamma[{}^{[U']}\!/_{r}], v\colon T' \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{r \lhd \uparrow v} P'$ *then* $\Gamma \vdash r\colon [?T'.U']$, $\Gamma \vdash v\colon T'$ *and* $\Gamma[{}^{[U']}\!/_{r}] \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{(v) r \lhd \uparrow v} P'$ *then* $\Gamma \vdash r\colon [?T'.U']$ *and* $\Gamma[{}^{[U']}\!/_{r}], v\colon T' \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{r \lhd \downarrow v} P'$ *then* $\Gamma \vdash r\colon [!T'.U']$ *and* $\Gamma[{}^{[U']}\!/_{r}], v\colon T' \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{r \tau} P'$ *then* $\Gamma \vdash r\colon [!T'.U']$ *or* $\Gamma \vdash r\colon [?T'.U']$ *and* $\Gamma[{}^{[U']}\!/_{r}] \vdash P'\colon (U, T)$;
- *if* $P \xrightarrow{\tau} P'$ *then* $\Gamma \vdash P'\colon (U, T)$;

Proof. The proof is by induction on the derivation of the transition. A case analysis according to the last used rule is needed.

L-send: P has the form $v.P'$. This can be typed only using rule T-SEND and this requires $U = !T'.U'$, $\Gamma \vdash P'\colon (U', T)$ and $\Gamma \vdash v\colon T'$. This is exactly as desired.

L-receive: P has the form $(x)P''$ and $P' = P''[{}^{v}\!/_{x}]$. P can be typed only using rule T-RECEIVE and this requires $U = ?T'.U'$ and $\Gamma, x\colon T' \vdash P'\colon (U', T)$. Thanks to Lemma B.2 we also have $\Gamma, v\colon T' \vdash P'[{}^{v}\!/_{x}]\colon (U', T)$.

L-call: P has the form $a \Leftarrow P''$ and $P' = r \lhd P''$. P can be typed only using rule T-CALL and this requires $U = \mathsf{end}$, $\Gamma \vdash P'' : (U', T)$ and $\Gamma \vdash a : [\overline{U'}]$. Using rule T-SESS-C (and thanks to Lemma B.4) one can derive $\Gamma, r\colon [\overline{U'}] \vdash r \lhd P''\colon (\mathsf{end}, T)$.

L-def: P has the form $a \Rightarrow P''$ and $P' = r \rhd P''$. P can be typed only using rule T-DEF and this requires $U = \mathsf{end}$, $\Gamma \vdash P'' : (U', T)$ and $\Gamma \vdash a : [U']$. Using rule T-SESS-S (and thanks to Lemma B.4) one can derive $\Gamma, r\colon [U'] \vdash r \rhd P''\colon (\mathsf{end}, T)$.

L-feed: P has the form $\mathsf{feed}\, v.P'$. This can be typed only using rule T-FEED and this requires $\Gamma \vdash P'\colon (U, T)$ and $\Gamma \vdash v\colon T$. This is exactly as required.

L-read: P has the form $f(x).P''$ and $P' = P''[{}^{v}\!/_{x}]$. P can be typed only using rule T-READ and this requires $\Gamma, x\colon T' \vdash P''\colon (U, T)$ and $\Gamma \vdash f\colon \{T'\}$. From Lemma B.2 we have $\Gamma, v\colon T' \vdash P''[{}^{v}\!/_{x}]\colon (U, T)$ as required.

L-stream-pass-P: P has the form $\mathsf{stream}\, P''\, \mathsf{as}\, f = \vec{v}\ \mathsf{in}\, Q$ with $P'' \xrightarrow{\mu} P'''$ and we have $P' = \mathsf{stream}\, P'''\, \mathsf{as}\, f = \vec{v}\ \mathsf{in}\, Q$. There are two cases according to the last rule used to type P. We consider rule T-STREAM-R first and rule T-STREAM-L later. Thanks to Lemma B.3 $\mu \notin \{\uparrow v, \downarrow v, (v) \uparrow v\}$. Also, $\mu \notin \{\Uparrow v, (v) \Uparrow v\}$. By hypothesis all the assumptions on f, $\vec{v}$ and Q are satisfied. By inductive hypothesis in all the cases but $r \bowtie\uparrow v$, $(v)r \bowtie\uparrow v$, $r \bowtie\downarrow v$

and $r\tau$ we have that $\Gamma' \vdash P''' \colon (\mathsf{end}, T)$ for some extension Γ' of Γ. Thanks to Lemma B.4 Γ' can be used to derive $\Gamma' \vdash P' \colon (U, T)$ as required. Notice also that the assumptions on Γ' are satisfied by inductive hypothesis since the label is unchanged. For the other cases the problem is that the assumption about r is changed. However, thanks to Lemma B.1 there are two cases. If there is just one occurrence of r, thus the assumption is never used outside P''', Lemma B.5 can be used to drop the old assumption and Lemma B.4 to add the new one, and the thesis follows. If there are three occurrences two should be in opposite session constructs and the third in a restriction binding them. The only label of these that can cross the restriction is $r\tau$, thus no occurrence of r can be in Q, since otherwise we can not obtain this label. Thus r is not used in Q and we can derive $\Gamma' \vdash Q \colon (U, T)$ as required, using again lemmas B.5 and B.4. Thus we can also derive $\Gamma' \vdash P' \colon (U, T)$ and the thesis follows.

Let us consider the second case. Notice that $\mu \notin \{\Uparrow v, (v) \Uparrow v\}$. Now both U and μ are preserved from the premise, thus in most of the cases the thesis follows immediately from the inductive premise (when a new assumption is needed in Γ, such as in extrusions, Lemma B.4 can be used, and the compatibility of the new assumption is guaranteed by the side condition on bound names of the typing rule). The only tricky cases concern labels $r \bowtie\uparrow v$, $(v)r \bowtie\uparrow v$, $r \bowtie\downarrow v$ and $r\tau$, but the same reasoning above applies. The thesis follows.

L-stream-pass-Q: P has the form $\mathsf{stream}\, P''\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q$ with $Q \xrightarrow{\mu} Q'$ and we have $P' = \mathsf{stream}\, P''\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q'$. By hypothesis all the assumptions on P, f and $\vec{v}$ are satisfied. Also, $\Gamma, f : \{T'\} \vdash Q \colon (U, T)$. By inductive hypothesis $\Gamma', f : \{T'\} \vdash Q' \colon (U', T)$ where Γ' and U' are defined by the statement of the theorem. Notice that Γ' verifies all the assumptions of rule T-STREAM-L (resp. T-STREAM-R) since it is either an extension of Γ (and in this case Lemma B.4 can be used), or it changes the assumption about some session r, and in this case the same reasoning done for rule L-STREAM-PASS-P can be used. Thus one can use rule $TStreamL$ (resp. T-STREAM-R) to derive $\Gamma' \vdash P' \colon (U', T)$ as required.

L-stream-feed: P has the form $\mathsf{stream}\, P''\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q$ with $P'' \xrightarrow{\Uparrow v} P'''$ and we have $P' = \mathsf{stream}\, P'''\, \mathsf{as}\, f = v :: \vec{w}\, \mathsf{in}\, Q$. There are two cases corresponding to rules T-STREAM-R and T-STREAM-L. We consider just the first one, the second being similar. By hypothesis $\Gamma \vdash P'' \colon (\mathsf{end}, T')$, $\Gamma, f \colon \{T'\} \vdash Q \colon (U, T)$ and $w' \in \mathrm{Set}(\vec{w}) \Rightarrow \Gamma \vdash w' \colon T'$. By inductive hypothesis $\Gamma \vdash v \colon T'$ and $\Gamma \vdash P''' \colon (\mathsf{end}, T')$. Thus using rule T-STREAM-R we can prove $\Gamma \vdash P' \colon (U, T)$ (notice in particular that the assumption about $v :: \vec{w}$ can be proved from the assumptions about v and $\vec{w}$).

L-stream-cons: P has the form $\mathsf{stream}\, P''\, \mathsf{as}\, f = \vec{w} :: v\ \mathsf{in}\, Q$ with $Q \xrightarrow{f \Downarrow v} Q'$ and $P' = \mathsf{stream}\, P''\, \mathsf{as}\, f = \vec{w}\ \mathsf{in}\, Q'$. There are two cases corresponding to rules T-STREAM-R and T-STREAM-L. We consider just the first the second being symmetric. By hypothesis $\Gamma \vdash P'' \colon (\mathsf{end}, T')$, $\Gamma, f \colon \{T'\} \vdash Q \colon (U, T)$ and $w' \in \mathrm{Set}(\vec{w} :: v) \Rightarrow \Gamma \vdash w' \colon T'$. By inductive hypothesis $\Gamma, f \colon \{T'\}, v \colon T' \vdash Q' \colon (U, T)$. Since $\Gamma, v \colon T'$ is an extension of Γ we can use it (thanks to Lemma B.4) in all the premises of rule T-STREAM-R and finally derive $\Gamma, v \colon T' \vdash P' \colon (U, T)$.

L-par: the reasoning is as for rule L-STREAM-PASS-P, but there is no stream here.

L-sess-val: we consider just the cases for $\lhd$, the other being simpler. P has the form $r \lhd P''$. By hypothesis $U = \mathsf{end}$, $\Gamma \vdash P'' \colon (U', T)$ and $\Gamma \vdash r \colon [\overline{U'}]$.

Let us consider the case $P'' \xrightarrow{\uparrow v} P'''$ before. This implies $P' = r \lhd P'''$. By inductive hypothesis $U' = !T'.U''$, $\Gamma \vdash v \colon T'$ and $\Gamma \vdash P''' \colon (U'', T)$. Using rule T-SESS-C we can prove $\Gamma[\overline{[U'']}/r] \vdash r \lhd P''' \colon (\mathsf{end}, T)$ as required since this is the only place where the assumption about r is used inside the term thanks to Lemma B.1, thus it can be changed using lemmas B.5 and B.4.

Let us now consider the case $P'' \xrightarrow{\downarrow v} P'''$. Again $P' = r \lhd P'''$. By inductive hypothesis $U' = ?T'.U''$, $\Gamma, v \colon T' \vdash P''' \colon (U'', T)$. Using rule T-SESS-C we can prove $\Gamma[\overline{[U'']}/r], v \colon T' \vdash$

$r \lhd P'''$ as required since this is the only place where the assumption about r is used inside the term thanks to Lemma B.1, thus it can be changed using lemmas B.5 and B.4.

L-sess-pass: we consider just the cases for $\lhd$, the others being simpler. P has the form $r \lhd P''$ with $P'' \xrightarrow{\mu} P'''$ and $P' = r \lhd P'''$. By hypothesis $U = \mathsf{end}$, $\Gamma \vdash P'' \colon (U', T)$ and $\Gamma \vdash r \colon [\overline{U'}]$. Notice that $\mu \neq \updownarrow v$. Thus for all the cases but session communication labels we have $\Gamma' \vdash P''' \colon (U', T)$ for some extension Γ' of Γ. In the case of session communication labels instead the assumption about r' is changed from Γ to Γ'. Notice that thanks to Lemma B.1 $r \neq r'$, thus in both the cases we can use rule T-SESS-C to derive $\Gamma' \vdash r \lhd P''' \colon (\mathsf{end}, T)$ as required since the label of the new transition is equal to the label of the premise, thus the assumptions on Γ' are the same ones.

L-sess-com-stream: P has the form $\mathsf{stream}\, P''\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q$ with $P'' \xrightarrow{r \triangleright \uparrow v} P'''$, $Q \xrightarrow{r \lhd \downarrow v} Q'$ and $P' = \mathsf{stream}\, P'''\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'$ (the other cases are similar). There are two cases corresponding to rules T-STREAM-R and T-STREAM-L. We consider just the first one, the second being similar. By hypothesis $\Gamma \vdash P \colon (\mathsf{end}, T')$ and $\Gamma, f \colon \{T'\} \vdash Q \colon (U, T)$. By inductive hypothesis on the first transition $\Gamma \vdash r \colon [!T''.U']$, $\Gamma \vdash v \colon T''$ and $\Gamma[{}^{[U']}\!/_{r}] \vdash P''' \colon (U, T)$. From the second transition we have a redundant hypothesis on r and $\Gamma[{}^{[U']}\!/_{r}], f \colon \{T'\}, v \colon T' \vdash Q' \colon (U, T)$. Notice that $\Gamma[{}^{[U']}\!/_{r}], f \colon \{T'\}, v \colon T' = \Gamma[{}^{[U']}\!/_{r}], f \colon \{T'\}$ since $\Gamma[{}^{[U']}\!/_{r}], f \colon \{T'\} \vdash v \colon T'$. Thus we can apply rule T-STREAM-R to derive $\Gamma[{}^{[U']}\!/_{r}] \vdash P' \colon (U, T)$ as required.

L-serv-com-stream: P has the form $\mathsf{stream}\, P''\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q$ with $P'' \xrightarrow{\downarrow a(r)} P'''$, $Q \xrightarrow{\uparrow a(r)} Q'$ and $P' = (\nu r)\mathsf{stream}\, P'''\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'$ (the symmetric case is similar). There are two cases corresponding to rules T-STREAM-R and T-STREAM-L. We consider just the first one, the second being similar. By hypothesis $\Gamma \vdash P'' \colon (\mathsf{end}, T')$ and $\Gamma, f \colon \{T'\} \vdash Q \colon (U, T)$. By inductive hypothesis (on both the transitions) $\Gamma \vdash a \colon [U']$ and $\Gamma, r \colon [U'] \vdash P'' \colon (\mathsf{end}, T')$ and $\Gamma, f \colon \{T'\}, r \colon [U'] \vdash Q' \colon (U, T)$. Using rule T-STREAM-R we can derive $\Gamma, r \colon [U'] \vdash \mathsf{stream}\, P'''\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q' \colon (U, T)$. Then we can use rule T-RES to derive $\Gamma \vdash P' \colon (U, T)$ as desired.

L-sess-com-par: the reasoning is as for rule L-SESS-COM-STREAM, but there is no stream here.

L-serv-com-par: the reasoning is as for rule L-SERV-COM-STREAM, but there is no stream here.

L-res: P has the form $(\nu n)P''$ with $P'' \xrightarrow{\mu} P'''$ and $P' = (\nu n)P'''$. By hypothesis $\Gamma, n \colon \_ \vdash P'' \colon (U, T)$. By inductive hypothesis $\Gamma', n \colon \_ \vdash P''' \colon (U', T)$ where Γ' and U' are as defined by the statement of the theorem. Thus we can apply rule T-RES to derive $\Gamma' \vdash (\nu n)P''' \colon (U', T)$ since the label is unchanged thus Γ' and U' are as before.

L-extr: P has the form $(\nu a)P''$ with $P'' \xrightarrow{\mu} P'$. By hypothesis $\Gamma, a \colon T' \vdash P'' \colon (U, T)$. Thanks to the inductive hypothesis $\Gamma', a \colon T' \vdash P' \colon (U', T)$ where Γ' and U' are as described in the statement of the theorem. This is exactly as required, given the different requirements between each action and the corresponding extruding action.

L-sess-res: P has the form $(\nu r)P''$ with $P'' \xrightarrow{r\tau} P'''$ and $P' = (\nu r)P'''$. By hypothesis $\Gamma, r \colon [U'] \vdash P'' \colon (U, T)$ (the type of r should be a protocol since r is a session). By inductive hypothesis $\Gamma, r \colon [U''] \vdash P''' \colon (U, T)$. Then we can use rule T-RES to derive $\Gamma \vdash P''' \colon (U, T)$ as required.

L-struct: By Lemma B.6.

□

Theorem 4.2 (Subject Reduction)

Proof. The thesis follows from Theorem B.7 and the characterization of reductions as transitions with labels τ given in Theorem 3.8. □

C Type safety

Theorem 4.3 (Type Safety)

Proof. The proofs of all the cases are by contradiction. We suppose that such a subterm exists and we show that it is not typable. We consider the three different cases:

Protocol: let us consider the first case. Here $v.P$ and $u.Q$ have types of the form $([!T.U], T'')$ and $([!T'.U'], T''')$ respectively. One can prove by structural induction on the context that the protocol part of the type is preserved (only the session construct can change it, but the side condition forbids sessions around the hole). Thus the two session constructs require $r : [!T.U]$ and $r : [?T'.\overline{U'}]$ (supposing that the first one is a server session, the symmetric otherwise). Since $\mathcal{D}[\![,]\!]$ does not bind r the assumptions are preserved, and at top level they should agree since the Γ used to type the two sides of parallel composition or stream is the same. This is not the case and we have the required contradiction.

The other cases are similar, with just end protocol for $\mathbf{0}$ and $([?T.U], T'')$ for input.

Sequentiality: in all the cases the two terms inserted into the double context have non end protocol. The property is preserved by the context (since there are no sessions around the hole). At top level we have two non end protocols, but the rules for parallel composition and stream can not be applied because of this. Since no other rules can type a parallel composition or a stream we have the desired contradiction.

□