Sessions and Pipelines for Structured Service Programming *

Michele Boreale[1], Roberto Bruni[2], Rocco De Nicola[1], and Michele Loreti[1]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze
{boreale,denicola,loreti}@dsi.unifi.it
[2] Dipartimento di Informatica, Università di Pisa
bruni@di.unipi.it

Abstract. Service-oriented computing is calling for novel computational models and languages with primitives for client-server interaction, orchestration and unexpected events handling. We present CaSPiS, a process calculus where the notions of session and pipelining play a central role. Sessions are two-sided and can be equipped with protocols executed by each side. Pipelining permits orchestrating the flow of data produced by different sessions. The calculus is also equipped with operators for handling (unexpected) termination of the partner's side of a session. Several examples are presented to provide evidence for the flexibility of the chosen set of primitives. Our main result shows that in CaSPiS it is possible to program a "graceful termination" of nested sessions, which guarantees that no session is forced to hang forever after the loss of its partner.

1 Introduction

The explosive growth of the Web has led to the widespread use of *de facto* standards for naming schemes (URI, URL), communication protocols (SOAP, HTTP, TCP/IP) and message format (XML). These three components have been used as the basis for building communication centered applications distributed over the web, often referred to as web services, and have put many expectations of the IT community on the growth of a new computational paradigm known as *Service-Oriented Computing* (SOC).

In SOC a main issue is the scalability of the proposed languages, models and techniques. Our belief is that the amount of complexity originated in the so-called global computing applications can be handled by considering well-structured and tightly disciplined approaches to the modeling of interaction. Well studied process algebras, like for instance π-calculus [21], have been used as a foundational model for SOC. However, we see two main problems along this way. The first is separation of concerns: SOC has different first-class aspects that would be mixed up and obfuscated when encoded via π-calculus channels. The second is that π-calculus communication primitives seem too liberal: the lack of structure in the communication topology increases the complexity of the analysis.

Here, we try to center the design of a new process calculus around a few prominent aspects of SOC applications, possibly reusing elegant patterns appeared in different

* Research supported by the Project FET-GC II IST-2005-16004 SENSORIA and by the Italian FIRB Project TOCAI.IT.

proposals. The three aspects that motivated our design choices are service autonomy, client-service interaction, and orchestration. Each aspect is briefly discussed below.

Services are heterogeneous computational entities, that are developed separately and often are scarcely reliable. Each service has *full autonomy* in denying a request or abandoning a pending interaction. A language for SOC should fix some standard mechanism for programming such decisions and to handle their consequences.

A language for SOC should support programming of complex and safe client-service interactions. By *interaction*, we mean the main unit of activity in a service-oriented application, which is essentially a conversation between a client and an instance of a service. The interaction may be *complex* as it will in general comprise both the exchange of several messages and the invocation of subsidiary services. As an example, consider a travel agent service that offers packages for organized trips. We expect that a dialogue takes place between the customer and the service, to let the service learn the customer preferences, let the customer select one among available packages, confirm or cancel the choice, and so on. In the course of this interaction, the service may need to invoke third party services to get, say, up-to-date flight or hotel information. By *safe*, we mean that, in principle, the involved parties should always be able either to complete the interaction or to recover from errors that prevent its completion, like, in the scenario above, one of the third-party services unexpectedly abandoning the conversation.

Orchestration is the process of assembling different services to build a new one or simply to perform a specific task. A central aspect of orchestration is the organization of the data flow among different activities. This flow also determines synchronization of activities. For instance, in the scenario outlined above, upon client's request, the travel agent service can start two new concurrent activities to get hotel and flight information (by invoking two subsidiary services), wait for their results and finally pass them on to the customer.

Motivated by these considerations, we introduce a language where *sessions* and *pipelines* are viewed as natural tools for monitoring the communication graph by structuring client-service interaction and orchestration, respectively. Autonomy is reflected as the ability to leave sessions. We name the new calculus *CaSPiS (Calculus of Sessions and Pipelines)*.

The use of session is a more abstract, alternative solution w.r.t. the W3C proposal of correlation sets. Here we use a name-scoping mechanism à la π-calculus to handle sessions. Pipelines have been inspired by Cook and Misra's Orc [22], a basic and elegant programming model for structured orchestration of services. In this light, they are seen as a convenient mechanism for modeling the flow of data between local processes: it is more general than sequential composition, better suited w.r.t. concurrency and does not require the explicit and improper use of channels for orchestration tasks. CaSPiS evolved from SCC *(Serviced Centered Calculus)* [3], a calculus that arose from a coordinated effort within the EU funded project SENSORIA [23]. The improvements and relationship of our work w.r.t. other proposals is discussed in the concluding section. In the rest of this section, we incrementally describe and motivate the main features of CaSPiS and discuss our results.

CaSPiS in a nutshell. In CaSPiS, service definitions and invocations are written like input and output prefixes in CCS. Thus $\mathtt{sign}.P$ defines a service `sign` that can be invoked by $\overline{\texttt{sign}}.Q$. There is an important difference, though, as the bodies P and Q are not quite continuations, but rather protocols that, within a session, govern interaction between (instances of) the client and the server. As an example:

$$!\texttt{sign}.(?x)(\nu t)\langle\{x,t\}_k\rangle \qquad \text{and} \qquad \overline{\texttt{sign}}.\langle\mathsf{plan}\rangle(?y)\langle y\rangle^{\uparrow}$$

are respectively: a (replicated and thus persistent) service whose instance waits for a digital document x, generates a fresh nonce t and then sends back both the document and the nonce signed with a key k; and a client that passes the argument plan to the service, then waits for the signed response from the server and returns this value outside the session as a result.

Synchronization of $s.P$ and $\overline{s}.Q$ leads to the creation of a new session, identified by a fresh name r that can be viewed as a private, synchronous channel binding caller and callee. Since client and service may be far apart, a session naturally comes with two sides, written $r \rhd P$ and $r \rhd Q$, with r bound somewhere above them by (νr). Values produced by P can be consumed by Q, and vice-versa: this permits description of interaction patterns more complex than the usual *one-way* and *request-response*. Rules governing creation and scoping of sessions are based on those of the restriction operator in the π-calculus. Note that multiple invocations to the same persistent service will yield separate sessions and that hierarchies of nested sessions, like $r_1 \rhd (r_2 \rhd P_2 | r_3 \rhd P_3)$ can arise if services are invoked within a running session. In the above case of service $\texttt{sign}$ the triggered session is

$$!\texttt{sign}.(?x)(\nu t)\langle\{x,t\}_k\rangle \quad | \quad (\nu r)\big(\, r \rhd (?x)(\nu t)\langle\{x,t\}_k\rangle \quad | \quad r \rhd \langle\mathsf{plan}\rangle(?y)\langle y\rangle^{\uparrow}\,\big)\,.$$

Here, after one reduction step where the value $\langle\mathsf{plan}\rangle$ available on the client-side is transmitted to the pending request $(?x)$ on the service-side (with x substituted by plan in the continuation), we get:

$$!\texttt{sign}.(?x)(\nu t)\langle\{x,t\}_k\rangle \quad | \quad (\nu r,t)\big(\, r \rhd \langle\{\mathsf{plan},t\}_k\rangle \quad | \quad r \rhd (?y)\langle y\rangle^{\uparrow}\,\big)\,.$$

Then, the digitally signed value $\{\mathsf{plan},t\}_k$ is computed at the service-side (for some fresh nonce t) and sent back to the client-side:

$$!\texttt{sign}.(?x)(\nu t)\langle\{x,t\}_k\rangle \quad | \quad (\nu r,t)\big(\, r \rhd \mathbf{0} \quad | \quad r \rhd \langle\{\mathsf{plan},t\}_k\rangle^{\uparrow}\,\big)\,.$$

The remaining activity will be then performed by the client protocol: $r \rhd \langle\{\mathsf{plan},t\}_k\rangle^{\uparrow}$ will emit $\{\mathsf{plan},t\}_k$ outside the (client-side of the) session, becoming the inert process $r \rhd \mathbf{0}$ (as already happened to the service side). In fact, values can be returned outside a session to the enclosing environment using the return operator, $\langle\cdot\rangle^{\uparrow}$.

Return values can be consumed by other sessions, or used to invoke other services, to start new activities. This is achieved using the pipeline operator $P > Q$. Here, a new instance of process Q is activated each time P emits a value that Q can consume. Notably, the new instance will run within the same session as P, not in a fresh one. For instance, what follows is a client that invokes the service $\texttt{sign}$ twice and then stores the obtained signed documents by invoking a suitable service $\texttt{store}$:

$$\big(\, \overline{\texttt{sign}}.\langle\mathsf{plan}_1\rangle(?y)\langle y\rangle^{\uparrow} \quad | \quad \overline{\texttt{sign}}.\langle\mathsf{plan}_2\rangle(?y)\langle y\rangle^{\uparrow}\,\big) \qquad > \qquad (?z)\overline{\texttt{store}}.\langle z\rangle\,.$$

The above description collects the main features of what we call the close-free fragment of CaSPiS that also includes guarded sums and input prefixes with pattern matching.

The distinguishing feature of our calculus is the presence of novel primitives to explicitly program session termination, to handle (unexpected or programmed) session termination and to garbage-collect terminated sessions. As explained before, session units must be able to autonomously decide to abandon the session they are running in. But since sessions units model client-server interactions, their termination must be programmed carefully, especially in the presence of nesting. The command close is used to terminate the enclosing session side. A terminated session enters the special state $\blacktriangleright P$ that recursively terminates any other session side nested in P. Note that the execution of a close can depend on some local choice as well as be guarded by the input of some data from the opposite session side.

The idea is that upon termination of a session side, the opposite session side will be informed and take some proper counteraction if needed. To achieve this, upon creation of a session, one associates with the fresh session r a pair of names (k_1,k_2), identifying a pair of *termination handlers*, one for each side. Then, right after execution of close a signal $\dagger(k_i)$ is sent to the termination-handler service k_i listening at the *opposite* side of the session. This handler will manage the appropriate actions. Since the name k_i must be known to the current side of the session, the more general syntax for sessions is $r \rhd_k P$ where the subscript k refers to the termination handler of the opposite side. To sum up the above discussion:

$$r \rhd_k \big(\mathsf{close}\,|P\big) \qquad \text{may evolve to} \qquad \dagger(k)|\blacktriangleright P\,.$$

Information about which termination handlers must be used is established at invocation time. To this purpose, the more general syntax for invocation is $\overline{s}_{k_1}.Q$. It mentions a name k_1 at which the termination handler of the client-side is listening. Symmetrically, the more general syntax for service definition is $s_{k_2}.P$, which mentions a name k_2 at which the termination handler of the service-side is listening. Then

$$\overline{s}_{k_1}.Q|s_{k_2}.P \qquad \text{can evolve to} \qquad (\nu r)(r \rhd_{k_2} Q|r \rhd_{k_1} P).$$

This way, if Q terminates with close, the termination handler k_2 of the callee will be activated, and vice versa, if P terminates then k_1 will be activated.

The mechanism of termination handlers is very expressive and flexible. Even if it may look overcomplicated to use, we emphasize that, up to our knowledge, this is the only proposal able to guarantee a disciplined termination of nested sessions. We conjecture that any mechanism of this kind would be very complicated to handle in say π-calculus.

Structure of the paper. For the sake of presentation, we introduce CaSPiS in two steps. First, we present the fragment without session-closing primitives, along with its labelled transition system semantics and well-formedness criteria (Section 2), with several nice examples in Section 3. Then we consider the full version of the calculus with session-closing primitives (Section 4). Our main result shows that, with the given primitives, it is possible to program what we call "graceful termination" of sessions (Section 5).

Specifically, we define a notion of *balanced* process (where all session-sides are pairwise balanced) and prove that any unbalanced state reachable from a balanced one can still become balanced in a finite number of steps. Final remarks, related work and future research avenues are exposed in Section 6.

2 The close-free fragment of CaSPiS

In this section, we introduce the fragment of CaSPiS without the constructs for handling session termination. The full calculus will be formalized only after the reader has gained some familiarity with the base constructs.

2.1 Syntax

Let $\mathcal{N}_{\mathrm{srv}}$ and $\mathcal{N}_{\mathrm{sess}}$ be two disjoint countable sets, respectively of *service* names $s, s', \ldots$ and of *session* names $r, r' \ldots$. We assume a countable set of *names* $\mathcal{N}$ ranged over by $n, n', \ldots$ that contains $\mathcal{N}_{\mathrm{srv}} \cup \mathcal{N}_{\mathrm{sess}}$ and such that $\mathcal{N} \setminus (\mathcal{N}_{\mathrm{srv}} \cup \mathcal{N}_{\mathrm{sess}})$ is infinite, and let $x, y, \ldots, u, v\ldots$ range over $\mathcal{N} \setminus \mathcal{N}_{\mathrm{sess}}$. We also assume a signature Σ of *constructors* $f, f', \ldots$, each coming with a fixed arity, such that Σ is disjoint from $\mathcal{N}$. We shall use $\tilde{\cdot}$ to denote sequences of items. The syntax of the basic fragment of CaSPiS is reported in Figure 1, where operators are listed in decreasing order of precedence.

To improve usability, structured values V can be built via Σ, and selection patterns F can be used to guard choices (via pattern matching). For simplicity, we consider as basic values only value expressions V built out of constructors in Σ and names $x, y, \ldots, u, v, \ldots$, the latter playing the role of variables or basic values depending on the context. We leave the signature Σ unspecified, but in several examples we shall assume Σ contains tuple constructors $\langle \cdot, \ldots, \cdot \rangle$ of arbitrary arity. Richer languages of expressions, comprising specific data values and evaluation mechanisms, are easy to accommodate. Finally, it is worth to note that session names $r, r', \ldots$ do *not* appear in values or patterns: this implies that they cannot be passed around, as it will be evident from the operational semantics (see Section 2.2).

As expected, in $(\nu n)P$, the restriction (νn) binds free occurrences of n in P, while in $(F)P$ an $?x$ in the pattern F binds the free occurrences of name x in P. We denote by $\mathrm{bn}(F)$ the set of names x such that $?x$ occurs in F. Processes are identified up to alpha-equivalence. The guarded sum with $I = \emptyset$ will also be denoted by $\mathbf{0}$. Trailing $\mathbf{0}$'s will often be omitted.

We will let $\sigma, \sigma', \sigma_1, \ldots$ range over substitutions, that is, finite partial functions from $\mathcal{N}$ to $\mathcal{N}$. In particular, we let $[^u/_x]$ denote the substitution that maps x to u. For any term T, we let $T\sigma$ denote the result of the capture-avoiding substitution of the free occurrences of x by $\sigma(x)$, for each $x \in \mathrm{dom}(\sigma)$.

Let us anticipate that the process grammar defined in Figure 1 should be pragmatically considered as a run-time syntax. In particular, sessions $r \rhd P$ can be generated at run-time, upon service invocation, but a programmer is not expected to explicitly use them. These considerations will lead us to impose some constraints on the somewhat too liberal syntax of Figure 1 and to introduce a notion of well-formedness (see Section 2.3).

$P,Q ::= \sum_{i\in I}\pi_i P_i$	Guarded Sum	$\pi ::= (F)$	Abstraction		
$\mid s.P$	Service Definition	$\mid \langle V\rangle$	Concretion		
$\mid \overline{s}.P$	Service Invocation	$\mid \langle V\rangle^{\uparrow}$	Return		
$\mid r \rhd P$	Session				
$\mid P > Q$	Pipeline	$V ::= u \mid f(\tilde{V})$	Value ($f \in \Sigma$)		
$\mid P\vert Q$	Parallel Composition				
$\mid (\nu n)P$	Restriction	$F ::= u \mid ?x \mid f(\tilde{F})$	Pattern ($f \in \Sigma$)		
$\mid !P$	Replication				

Fig. 1. Syntax of processes.

$$
\begin{array}{rclrclrcl}
(P|Q)|R & \equiv & P|(Q|R) & (\nu n)(\nu m)P & \equiv & (\nu m)(\nu n)P & P|(\nu n)Q & \equiv & (\nu n)(P|Q) \text{ if } n \notin \mathrm{fn}(P) \\
P|Q & \equiv & Q|P & (\nu n)\mathbf{0} & \equiv & \mathbf{0} & ((\nu n)Q) > P & \equiv & (\nu n)(Q > P) \text{ if } n \notin \mathrm{fn}(P) \\
P|\mathbf{0} & \equiv & P & !P & \equiv & P|!P & r \rhd (\nu n)P & \equiv & (\nu n)(r \rhd P) \text{ if } r \neq n
\end{array}
$$

Fig. 2. Structural congruence.

Structural congruence. Structural congruence $\equiv$ is defined as the least congruence relation induced by the laws in Figure 2. This set of laws comprises the structural rules for parallel composition and restriction from the π-calculus, plus the obvious extension of restriction's scope extrusion law to pipelines and sessions.

2.2 Operational semantics

We let $\xrightarrow{\lambda}$ be the labelled transition relation induced by the rules in Figure 4. Labels λ have the syntax and informal meaning defined in Figure 3, where τ is the silent action. We call a *reduction* any silent transitions $P \xrightarrow{\tau} Q$. Note that we shall often abbreviate $(\nu n_1)\cdots(\nu n_l)\lambda$ as $(\nu\tilde{n})\lambda$ where $\tilde{n} = n_1,\ldots,n_l$. We define $\mathrm{n}(\lambda)$, $\mathrm{fn}(\lambda)$ and $\mathrm{bn}(\lambda)$ as expected; in particular $\mathrm{bn}(s(r)) = \mathrm{bn}(\overline{s}(r)) = \{r\}$.

Service Definition and Invocation. Rule (DEF) describes the behaviour of a service definition: it says that a service $s.P$ is ready to establish a new session named r. Rule (CALL) describes the complementary behaviour of a service invocation $\overline{s}.Q$. Rule (SYNC) describes session creation as the result of $s.P$ and $\overline{s}.Q$ synchronizing and, in doing so, agreeing on a fresh session name r. The new session has two ends, one client's side where protocol Q is running and one at service's side where protocol P is running. A value produced by a concretion at one side can be consumed by an abstraction at the other side.

Communication prefixes. Rule (OUT) models the behaviour of concretion $\langle V\rangle P$ that can evolve to P with a label $\langle V\rangle$, denoting emission of value V. Rule (IN) models the behaviour of an abstraction $(F)P$ that can be seen as a form of guarded command that relies on pattern-matching: $(F)P$ can evolve to $P\sigma$ with (V), indicating consumption of a value, only provided the pattern F matches up the value V. This leads to a substitution

$$
\begin{array}{llllll}
\lambda ::= \tau & \lambda_i ::= s(r) & \text{(service definition)} & \lambda_o ::= \overline{s}(r) & \text{(service invocation)} \\
\quad \mid \lambda_i & \quad \mid (V) & \text{(value consumption)} & \quad \mid (\nu\tilde{n})\langle V\rangle & \text{(value production)} \\
\quad \mid \lambda_o & \quad \mid r:(V) & \text{(consumption within } r\text{)} & \quad \mid (\nu\tilde{n})r:\langle V\rangle & \text{(production within } r\text{)} \\
 & & & \quad \mid (\nu\tilde{n})\uparrow V & \text{(value return)}
\end{array}
$$

Fig. 3. Transition labels.

σ such that $\mathrm{match}(F,V)=\sigma$. Here, the pattern-matching function match is defined as expected: $\mathrm{match}(F,V)=\sigma$, if σ is the (only) substitution such that $\mathrm{dom}(\sigma)=\mathrm{bn}(F)$ and $F\sigma=V$. Rule $(\textsc{Ret})$ models the behaviour of the return primitive $\langle V\rangle^{\uparrow}P$ that can be used to return a value *outside* the current session if the enclosing environment is capable of consuming it. Guarded choice has the expected meaning: $\sum_{i\in I}\pi_i P_i$ evolves with λ to P' if there is an $i\in I$ such that $\pi_i P_i$ evolves with λ to P' (see rule $(\textsc{Sum})$).

Communication inside sessions. Rules $(\textsc{S-In})$ and $(\textsc{S-Out})$ add the name of the innermost enclosing session to the labels for intra-session communication. Rule $(\textsc{S-Sync})$ finalizes communication inside session r for complementary action labels.

Communication outside sessions. Rule $(\textsc{S-Ret})$ transforms a return performed inside a session r into the output of a value for the enclosing environment. Rule $(\textsc{S-Pass})$ propagates all session-transparent activities, namely $s(r')$, $\overline{s}(r')$, $r':(V)$, $(\nu\tilde{n})r':\langle V\rangle$ and τ. In fact, services can be accessed and invoked independently from the hierarchy of sessions, and intra-session communication is always bound to the innermost enclosing session.

Pipelining. Rule $(\textsc{P-Sync})$ expresses that in a pipeline $P>Q$ all the values V produced by P and that can be consumed by Q will trigger a new instance Q' of Q. (Note that, after this reduction, Q is again ready to consume the next value produced by P, if any.) Rule $(\textsc{P-Pass})$ indicates that the pipeline is transparent to all the other transitions of P, which are thus propagated to the enclosing context. Also note that Q is idle until a value produced by P activates one instance of it.

Restriction, parallel and structural congruence. Rules $(\textsc{Open})$, $(\textsc{R-Pass})$ and $(\textsc{Par})$ are the standard ones for restriction and parallel. However, thanks to the rule for structural congruence $(\textsc{Struct})$, we need not a *close* rule for names extruded via rule $(\textsc{Open})$, because all steps can be performed in normalized processes, with all restrictions moved to the top.

Additional Comments. Sessions, service definitions and service invocations can of course be nested at arbitrary depth. Note that no activity can take place under the scope of a dynamic operator (see Definition 1 below). On the contrary, when considering static contexts (see Definition 2 below), concurrent activities can take place at any level of the session hierarchy. Also note that sessions are completely transparent with respect to actions different from value production/consumption/return, that is, service invocation and silent steps.

$$(\textsc{Def})\ \frac{r \notin \mathrm{fn}(P)}{s.P \xrightarrow{s(r)} r \rhd P} \qquad (\textsc{Call})\ \frac{r \notin \mathrm{fn}(P)}{\overline{s}.P \xrightarrow{\overline{s}(r)} r \rhd P} \qquad (\textsc{Sync})\ \frac{P \xrightarrow{s(r)} P' \quad Q \xrightarrow{\overline{s}(r)} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')}$$

$$(\textsc{Out})\ \langle V\rangle P \xrightarrow{\langle V\rangle} P \qquad (\textsc{Ret})\ \langle V\rangle^{\uparrow} P \xrightarrow{\uparrow V} P \qquad (\textsc{In})\ \frac{\mathrm{match}(F,V) = \sigma}{(F)P \xrightarrow{(V)} P\sigma}$$

$$(\textsc{Sum})\ \frac{\pi_i P_i \xrightarrow{\lambda} P'}{\sum_{i\in I} \pi_i P_i \xrightarrow{\lambda} P'} \qquad (\textsc{S-In})\ \frac{P \xrightarrow{(V)} P'}{r \rhd P \xrightarrow{r:(V)} r \rhd P'} \qquad (\textsc{S-Out})\ \frac{P \xrightarrow{\langle V\rangle} P'}{r \rhd P \xrightarrow{r:\langle V\rangle} r \rhd P'}$$

$$(\textsc{S-Sync})\ \frac{P \xrightarrow{r:\langle V\rangle} P' \quad Q \xrightarrow{r:(V)} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad (\textsc{S-Ret})\ \frac{P \xrightarrow{\uparrow V} P'}{r \rhd P \xrightarrow{\langle V\rangle} r \rhd P'}$$

$$(\textsc{S-Pass})\ \frac{P \xrightarrow{\lambda} P'}{r \rhd P \xrightarrow{\lambda} r \rhd P'}\ \begin{array}{l}\lambda ::= \overline{s}(r') \,|\, s(r') \,|\, \tau \,|\, r' : (V) \,|\, r' : \langle V\rangle \\ r' \neq r\end{array}$$

$$(\textsc{P-Sync})\ \frac{P \xrightarrow{\langle V\rangle} P' \quad Q \xrightarrow{(V)} Q'}{P > Q \xrightarrow{\tau} (P' > Q) \,|\, Q'} \qquad (\textsc{P-Pass})\ \frac{P \xrightarrow{\lambda} P'}{P > Q \xrightarrow{\lambda} P' > Q}\ \lambda \neq \langle V\rangle$$

$$(\textsc{Open})\ \frac{P \xrightarrow{\lambda} P' \quad n \notin \mathrm{bn}(\lambda)}{(\nu n)P \xrightarrow{(\nu n)\lambda} P'}\ \begin{array}{l}\lambda ::= (\nu\tilde{n}')\langle V\rangle \,|\, (\nu\tilde{n}') \uparrow V \,|\, (\nu\tilde{n}')r : \langle V\rangle \\ n \in \mathrm{n}(V)\end{array} \qquad (\textsc{R-Pass})\ \frac{P \xrightarrow{\lambda} P' \quad n \notin \mathrm{n}(\lambda)}{(\nu n)P \xrightarrow{\lambda} (\nu n)P'}$$

$$(\textsc{Par})\ \frac{P \xrightarrow{\lambda} P'}{P|Q \xrightarrow{\lambda} P'|Q}\ \mathrm{fn}(Q) \cap \mathrm{bn}(\lambda) = \emptyset \qquad (\textsc{Struct})\ \frac{P \equiv Q \quad Q \xrightarrow{\lambda} Q' \quad Q' \equiv P'}{P \xrightarrow{\lambda} P'}$$

Fig. 4. Labelled Operational Semantics.

Remark 1 (About reduction semantics.). It has become common to present the operational semantics of newly proposed calculi by means of reductions instead of labelled transitions, with the advantage of a simpler and compact presentation. However, in the case of CaSPiS, the nesting and interplay of sessions and pipelines introduces some contextual dependencies on the reductions rules that makes the labelled transitions more appealing. The interested reader may check Lemma 3 to see all the different kinds of reductions that must be taken into account (the lemma is given for the full CaSPiS).

2.3 Well-formedness

As anticipated, we introduce a well-formedness criterion to impose some discipline on the way CaSPiS primitives can be used and rule out many pathologically wrong process designs. To this aim, we first introduce some terminology and technical constraints. We say P *has no top bound sessions* if $P \equiv (\nu r)P'$ implies $r \notin \mathrm{fn}(P')$, for each P', r.

A *context* is a process term with one occurrence of a distinct process variable, say X (representing the "hole"). In what follows, we shall indicate by $C[\cdot]$ a generic context, and by $C[P]$ the process obtained when textually replacing X by P in $C[\cdot]$. The term Q

occurs in (or is a subterm of) P if there is a context $C[\cdot]$ such that $P = C[Q]$. The notion of context can be generalized to n-holes contexts as expected. In particular, generic 2-holes contexts will be denoted by $C[\cdot,\cdot]$, with $C[P,Q]$ defined as obvious.

Definition 1. Dynamic *operators are service definition* $s.[\cdot]$ *and invocation* $\overline{s}.[\cdot]$*, any prefix* $\pi_i[\cdot]$*, right-hand side term of a pipeline* $P > [\cdot]$ *and replication* $![\cdot]$*. The remaining operators are* static.

Definition 2. *A context* $C[\cdot]$ *is* static *if its hole does not occur in the scope of a dynamic operator. Moreover, we say that* $C[\cdot]$ *is* session-immune *if its hole does not occur in the scope of a session operator.*

We introduce the final ingredient needed for our definition of well-formed process.

Definition 3. *Given* P *and two session names* $r, r' \in \mathrm{fn}(P)$*, we write* $r \prec_P r'$ *if and only if* $P \equiv C[r \rhd C'[r' \rhd P']]$ *for some static* $C[\cdot]$*, session-immune* $C'[\cdot]$ *and* P'.

In other words, $r \prec_P r'$ if, in P, up to structural congruence, an occurrence of r' is immediately within the scope of some session side $r \rhd [\cdot]$. Well-formedness is formally defined below:

Definition 4 (well-formedness). *Assume* $P \equiv (\nu\tilde{r})Q$*, where* Q *has no top bound sessions and* $\tilde{r} \subseteq \mathrm{fn}(Q)$*. Then, process* P *is* well-formed *if: (a) relation* $\prec_Q$ *is acyclic (that is,* $\prec_Q^+$ *is irreflexive), (b) modulo alpha conversion, for each* r*,* $r\rhd$ *occurs at most twice in* P *and never in the scope of a dynamic operator, (c) in any summation* $\sum_i \pi_i$*, all prefixes* π_i *are of one and the same kind (either all abstractions, or all concretions or all returns).*

The acyclicity of $\prec_Q$ rules out vicious situations like $r \rhd (P_1 | r \rhd P_2)$. For the rest, distinct service invocations, even of the same service, should give rise to distinct and fresh session names, and, of course, each session should normally have no more than two sides (one-sided sessions make sense once we allow one side to autonomously close, a scenario that we shall consider in Section 4). Also, for technical reasons, it is desirable to forbid mixed sums, that is sums where prefixes of different kinds occur.

In the remainder of this paper, *all processes are assumed to be well-formed*. By inspection, it is straightforward to check the validity of the following result.

Lemma 1. *Well-formedness is preserved by structural congruence and reductions.*

3 CaSPiS at work

In this section we present some simple examples that aim at showing how CaSPiS can be used for specifying behaviours of structured services. The examples expose some important patterns for service composition, for which we find it convenient to introduce a set of derived operators.

In the following we will assume the following services are available. Service `emailMe` when invoked with argument *msg* has the effect of sending a message *msg* to one's email address. Services `ANSA`, `BBC` and `CNN`, upon invocation, return a possibly infinite sequence of values representing pieces of news (disregarding the identity of these news, these services resemble $!\mathtt{ANSA}.!(\nu n)\langle n\rangle$, etc.).

$$\overline{s}\langle V\rangle \triangleq \overline{s}.\langle V\rangle \quad \text{(One Way)} \qquad P \gg \overline{s} \triangleq P > (?x)\overline{s}(x) \quad \text{(Simple Pipe)}$$

$$\overline{s}(V) \triangleq \overline{s}.\langle V\rangle(?x)\langle x\rangle^{\uparrow} \quad \text{(Request Response)} \qquad \overline{s}(!) \triangleq \overline{s}.!(?x)\langle x\rangle^{\uparrow} \quad \text{(Get All Responses)}$$

Fig. 5. CaSPiS derivable constructs

Invocation patterns. Recurrent service invocation patterns are reported in Figure 5:

- $\overline{s}\langle V\rangle$ invokes the service s and then sends the value V over the established session;
- $\overline{s}(V)$ invokes s, then sends value V over the established session, then waits for a value from the service (the result of the service invocation) and publishes it locally (outside the established session);
- $P \gg \overline{s}$ is a pipeline based composition to invoke service s on all values produced by P;
- $\overline{s}(!)$ invokes s, then keeps retrieving and publishing all responses from s.

Using the request-response and one-way patterns we can rewrite the example in the Introduction as: $\overline{\texttt{sign}}(\mathsf{plan}) > (?z)\overline{\texttt{store}}\langle z\rangle$. Simple pipe can be used to write the signing of a document by two different authorities as $\langle\mathsf{plan}\rangle \gg \overline{\texttt{sign}_1} \gg \overline{\texttt{sign}_2}$. Usage examples of the get-all-responses pattern are reported below for invoking news services.

Selection. Command **select** permits collecting the first n values emitted by a set of processes running in parallel and satisfying a given sequence of patterns.
Formally, we define **select** $F_1,\ldots,F_n$ **from** P as follows:

$$\textbf{select } F_1,\ldots,F_n \textbf{ from } P \triangleq (\nu s)\big(s.(F_1).\ldots.(F_n)\langle\hat{F}_1,\ldots,\hat{F}_n\rangle^{\uparrow} \mid \overline{s}.P\big)$$

where for each pattern F_i, $\hat{F}_i$ denotes the value V_i obtained from F_i by replacing each $?x$ with x. A useful variation of the above command is **select** $F_1,\ldots,F_n$ **from** P **in** Q, defined as follows:

$$\textbf{select } F_1,\ldots,F_n \textbf{ from } P \textbf{ in } Q \triangleq \textbf{select } F_1,\ldots,F_n \textbf{ from } P > (F_1,\ldots,F_n)Q$$

As an example the process

$$\textbf{select } ?x,?y \textbf{ from } \big(\overline{\texttt{ANSA}}(!) \mid \overline{\texttt{BBC}}(!) \mid \overline{\texttt{CNN}}(!)\big) \textbf{ in } \overline{\texttt{emailMe}}\langle x,y\rangle$$

will send to the specified email address a pair of the first two pieces of news among those arriving from ANSA, BBC and CNN, no matter who has actually produced them.

Waiting. When waiting for values produced by a set of concurrent activities, it may be useful not only to constrain the patterns of the expected values, but also to fix the exact binding between patterns and activities, that is, to specify which activity is expected to produce what. For instance, in the previous example, one might want to receive a mail with three pieces of news, the first coming from ANSA, the second from BBC and the third from CNN. This also implies that a mail will be sent only when a piece of news has been received from each of these services. We let **wait** $F_1,\ldots,F_n$ **from** $P_1,\ldots,P_n$ be a process that emits tuple $\langle V_1,\ldots,V_n\rangle$, where V_i is the first value emitted by P_i and matching F_i, for $i = 1,...,n$. We have that:

$$\begin{array}{rl}\textbf{wait}\ F_1,\ldots,F_n\ \textbf{from}\ P_1,\ldots,P_n \stackrel{\triangle}{=} & (\nu s)\big(\ s.(t_1(F_1)).\ldots.(t_n(F_n))\langle \hat{F}_1,\ldots,\hat{F}_n\rangle^{\uparrow} \\ & \mid \overline{s}.(\ \textbf{select}\ F_1\ \textbf{from}\ P_1\ \textbf{in}\ \langle t_1(\hat{F}_1)\rangle \mid \cdots \\ & \quad \mid \textbf{select}\ F_n\ \textbf{from}\ P_n\ \textbf{in}\ \langle t_n(\hat{F}_n)\rangle\ )\ \big)\end{array}$$

We can also consider the following variation:

$$\textbf{wait}\ F_1,\ldots,F_n\ \textbf{from}\ P_1,\ldots,P_n\ \textbf{in}\ Q \stackrel{\triangle}{=} \textbf{wait}\ F_1,\ldots,F_n\ \textbf{from}\ P_1,\ldots,P_n > (F_1,\ldots,F_n)Q$$

Then, **wait** $?x,?y,?z$ **from** $\overline{\texttt{ANSA}}(!)$, $\overline{\texttt{BBC}}(!)$, $\overline{\texttt{CNN}}(!)$ **in** $\overline{\texttt{emailMe}}\langle x,y,z\rangle$ will send an email containing the first pieces of news distributed by each of `ANSA`, `BBC` and `CNN`.

Travel Agent Service. We put at work macros *select* and *wait* defined above for describing a classical example of service composition: a Travel Agent Service. A travel agent offers its customers the ability to book packages consisting of services offered by various providers. For instance: Flight and Hotel Booking. Three kinds of booking are available: Flight only, Hotel only, or both. A customer contacts the service and then provides it with appropriate information about the planned travel (origin and destination, departing and returning dates,...). When a request is received, the service contacts appropriate services (for instance `Lufth` and `Alit` for flights, and `HInn` and `BWest` for hotels) and then compares their offers to select the most convenient (this is actually done by invoking a sub-service `compare`), which is returned to the customer.

This service can be specified in CaSPiS as follows:

$$\begin{array}{rl}!\texttt{ta}. & (\mathit{fly}(?x)).\textbf{wait}\ ?y,?z\ \textbf{from}\ \overline{\texttt{Lufth}}\langle x\rangle,\overline{\texttt{Alit}}\langle x\rangle\ \textbf{in}\ \overline{\texttt{compare}}(y,z) \\ + & (\mathit{hotel}(?x)).\textbf{wait}\ ?y,?z\ \textbf{from}\ \overline{\texttt{BWest}}\langle x\rangle,\overline{\texttt{HInn}}\langle x\rangle\ \textbf{in}\ \overline{\texttt{compare}}(y,z) \\ + & (\mathit{fly\&hotel}(?x)).\textbf{wait}\ ?y,?z\ \textbf{from}\ \overline{\texttt{ta}}(fly(x)),\overline{\texttt{ta}}(hotel(x))\ \textbf{in}\ \langle y,z\rangle\end{array}$$

After invocation, one message among three possible kinds of requests is expected: *fly*(V), *hotel*(V) and *fly*&*hotel*(V), where value V contains trip information. For example, the first case, two values, each containing an offer for a flight, are obtained from `Lufth` and `Alit`. The pair of these values is passed to service `compare` that selects the most convenient, and then immediately returned on to the client. Note that, upon receiving a *fly*&*hotel* request, service `TA` recursively invokes itself for determining the best offers for flight and hotel. These values are then returned to the client.

π-calculus channels. By analogy with SCC [3], it can be easily inferred that the close-free fragment of CaSPiS is expressive enough to model (lazy) λ-calculus and that, in the absence of pattern matching, it can be encoded in π-calculus.

The close-free fragment of CaSPiS is also expressive enough to encode π-calculus. Indeed, input and output over a channel a can be encoded in CaSPiS as follows:

$$a(x).P \stackrel{\triangle}{=} a.(?x)\langle x\rangle^{\uparrow} > (?x)P \qquad \overline{a}v.P \stackrel{\triangle}{=} \overline{a}.\langle v\rangle\langle\bullet\rangle^{\uparrow} > (\bullet)P$$

A proxy service. We conclude this section by considering the description of a simple *proxy service* that, once received a service name s and a value x, invokes s with parameter x and sends back to the caller all the values emitted by s:

$$!proxy.(?s,?x)\overline{s}.\langle x\rangle!(?y)\langle y\rangle^{\uparrow}$$

4 The full calculus

The calculus we have presented in the preceding sections offers no primitives for handling session closing. These primitives might be useful to garbage-collect terminated sessions. Most important, one might want to explicitly program session termination, in order to implement cancellation workflow patterns [24], or to manage abnormal events, or timeouts.

Sessions are units of client-server cooperation and as such their termination must be programmed carefully. At least, one should avoid situations where one side of the session is removed abruptly and leaves the other side dangling forever. Also, subsessions should be informed and, e.g., close in turn. The full CaSPiS we are going to introduce comprises mechanisms for programming disciplined closing sessions. The mechanism of session termination we shall adopt has been informally described in the Introduction. Before introducing this extension formally, we discuss a couple of additional issues below.

An important aspect of our modelling is that, when shutting down a side, the emitted signal $\dagger(k)$ is (realistically) *asynchronous*. In other words, we have no guarantee as to when the termination handler at the opposite side will be reached by $\dagger(k)$. So, for example, by the time $\dagger(k)$ reaches its destination, the other side might in turn have entered a closing state $\blacktriangleright Q$ on its own, or be closed right away, as a result of the closing of a parent session. These aspects must be taken into account when defining what "well-programmed" closing means (see Section 5). In general, dangling $\dagger(k)$ cannot be avoided, but we will be able to avoid sessions dangling forever.

It is worth mentioning that there are at least two obvious alternatives to the mechanism we have chosen. One would be to use close as a primitive for terminating instantaneously *both* the client-side and service-side sessions. But, as discussed above, this strategy conflicts with the two parties being in charge for the closing of their own session sides. A second alternative would be to use close as a synchronization primitive, so that the client-side and service-side sessions are terminated when close is encountered on one side and $\overline{\text{close}}$ on the other side. This strategy conflicts with parties being able to decide autonomously when to end their own sessions. The use of termination handlers looks a reasonable compromise: each party can exit a session autonomously but it is obliged to inform the other party.

4.1 Syntax and operational semantics of the full calculus

Syntax. In what follows, we assume a new countable set $\mathcal{K}$ of *signal names* $k, k', \ldots$, disjoint from session and service names. The syntax of full CaSPiS is reported in Figure 6. In addition to the extended primitives $s_k.P$, $\overline{s}_k.P$ and $r \rhd_k P$ and to the new primi-

$P,Q ::=$	$\sum_{i\in I}\pi_i P_i$	Guarded Sum	$\mid$	$\dagger(k)$	Signal
	$\mid s_k.P$	Service Definition	$\mid$	$r \rhd_k P$	Session
	$\mid \overline{s}_k.P$	Service Invocation	$\mid$	$\blacktriangleright P$	Terminated Session
	$\mid P > Q$	Pipeline	$\mid$	$P\vert Q$	Parallel Composition
	$\mid \mathsf{close}$	Close	$\mid$	$(\nu n)P$	Restriction
	$\mid k\cdot P$	Listener	$\mid$	$!P$	Replication

Fig. 6. Syntax of full CaSPiS.

tives close, $\dagger(k)$ and $\blacktriangleright P$ discussed in the Introduction, note the presence of termination listeners $k\cdot P$ that are used to handle termination signals $\dagger(k)$.

In what follows, we say that a name *n occurs free in P underneath a context* $C[\cdot]$ if there is a term Q such that $n \in \mathrm{fn}(Q)$ and $P = C[Q]$. For instance, k occurs free in $!k\cdot\mathbf{0}$ underneath $![\cdot]$, while it does not occur free in $!(\nu k)(k\cdot\mathbf{0})$ underneath $![\cdot]$.

Well-formedness. Beside those reported in Section 2, we assume the following additional well-formedness conditions on the syntax presented in Figure 6:

- for any signal name k and term P, modulo alpha-equivalence there is at most one subterm of P of the form $r \rhd_k Q$; moreover, k does not occur in concretion prefixes or return prefixes, and does not occur *free* in P underneath any dynamic context;
- Terminated sessions operators $\blacktriangleright$ do not occur within the scope of a dynamic operator.

The above conditions are easily seen to be preserved by the structural rules and by the SOS rules presented below. Note that the second condition above implies that passing of signal names is forbidden. In what follows, we assume all terms are well-formed.

Structural congruence. Nesting of sessions may give rise to subtle race conditions on the order of closings. As an example, consider a situation with two sessions r_1 and r_2, both ready to close and with r_2 nested in r_1:

$$r_1 \rhd_{k_1} \big(\ \mathsf{close}\ \mid\ P\ \mid\ r_2 \rhd_{k_2} (\mathsf{close}\,|Q)\ \big)\,.$$

Suppose the innermost session r_2 closes up first; then, before the signal $\dagger(k_2)$ reaches its listener, also session r_1 closes up. This leads to the situation $\dagger(k_1)\,|\,\blacktriangleright(P\,|\,\dagger(k_2)\,|\,\blacktriangleright Q)$, where one still wants to activate the listener on k_2, despite the fact that $\dagger(k_2)$ lies within a terminated session. This example shows that terminated session operator, $\blacktriangleright$, should stop any activity *but* invocation of listeners, that is signals $\dagger(k)$.

The structural rules listed in Figure 7 enrich the set of rules already introduced for the basic calculus. The law $\blacktriangleright \dagger(k) \equiv \dagger(k)$ has been motivated above. The remaining rules serve the purpose of letting signals $\dagger(k)$ freely move within a term to reach the corresponding listeners, and distributing the terminated session $\blacktriangleright$ over static operators.

Note that, by structural congruence, in any term it is possible to move all restrictions not in the scope of a dynamic operator to top level.

$$
\begin{array}{rclrclrcl}
r \rhd_{k'} (\dagger(k)|P) & \equiv & \dagger(k)|r \rhd_{k'} P & (\dagger(k)|P) > Q & \equiv & \dagger(k)|(P > Q) & \blacktriangleright \dagger(k) & \equiv & \dagger(k) \\
\blacktriangleright r \rhd_k P & \equiv & \blacktriangleright r \rhd_k \blacktriangleright P & \blacktriangleright (P > Q) & \equiv & (\blacktriangleright P) > Q & \blacktriangleright\blacktriangleright P & \equiv & \blacktriangleright P \\
\blacktriangleright P|Q & \equiv & \blacktriangleright P| \blacktriangleright Q & \blacktriangleright (\nu x)P & \equiv & (\nu x) \blacktriangleright P & \blacktriangleright \mathbf{0} & \equiv & \mathbf{0}
\end{array}
$$

Fig. 7. Structural congruence rules for $\dagger(k)$ and $\blacktriangleright$.

$$
(\textsc{Def})\ s_{k_1}.P \xrightarrow{s(r)^{k_2}_{k_1}} r \rhd_{k_2} P \quad (\textsc{Call})\ \overline{s}_{k_2}.P \xrightarrow{\overline{s}(r)^{k_1}_{k_2}} r \rhd_{k_1} P \quad (\textsc{Sync})\ \frac{P \xrightarrow{s(r)^{k_2}_{k_1}} P' \quad Q \xrightarrow{\overline{s}(r)^{k_1}_{k_2}} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')}
$$

$$
(\textsc{Lis})\ k \cdot P \xrightarrow{k} P \qquad (\textsc{Sig})\ \dagger(k) \xrightarrow{\overline{k}} \mathbf{0} \qquad (\textsc{T-Sync})\ \frac{P \xrightarrow{k} P' \quad Q \xrightarrow{\overline{k}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}
$$

$$
(\textsc{End})\ \mathsf{close} \xrightarrow{\mathsf{close}} \mathbf{0} \qquad (\textsc{S-End})\ \frac{P \xrightarrow{\mathsf{close}} P'}{r \rhd_k P \xrightarrow{\tau} \blacktriangleright P'|\dagger(k)} \qquad (\textsc{T-End})\ \blacktriangleright r \rhd_k P \xrightarrow{\tau} \blacktriangleright P|\dagger(k)
$$

Fig. 8. Labelled Operational Semantics of full CaSPiS.

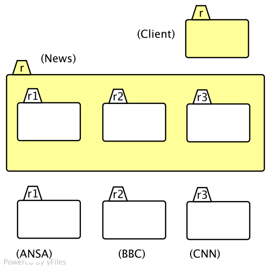
Operational semantics. The SOS rules for the new operators are reported in Figure 8, the rules for the remaining operators are the same as those introduced for the basic calculus (Section 2). Roughly: (1) rules (DEF), (CALL) and (SYNC) have been updated to take into account the exchange of termination handler names, (2) new rules (LIS), (SIG) and (T-SYNC) are used to synchronize a listener with a pending signal for it, (3) new rules (END) and (S-END) are used to close a session from inside and generate a signal towards the handler of the partner, and (4) a new rule (T-END) is used to recursively close session that were nested in a closed session (in a top-down fashion). Garbage collecting rules for guarded sums, service definitions and service invocations, like $\blacktriangleright \overline{s}_k.P \xrightarrow{\tau} \mathbf{0}$, are omitted for simplicity, but they can be added without any relevant consequence on the results in Section 5.

Example 1. Let us consider process *News* defined as follows:

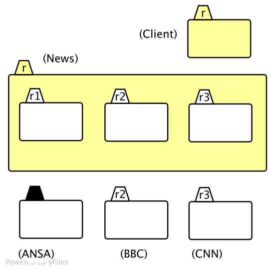
$$
\begin{array}{rl}
News \triangleq\ !(\nu k)\mathtt{collect}_k.\big(\, k \cdot \mathsf{close} & |\ (\nu k_1)\overline{\mathtt{ANSA}}_{k_1}.(!(?x)\langle x\rangle^{\uparrow}\ |\ k_1 \cdot (\mathsf{close}|\dagger(k))) \\
& |\ (\nu k_2)\overline{\mathtt{BBC}}_{k_2}.(!(?x)\langle x\rangle^{\uparrow}\quad |\ k_2 \cdot (\mathsf{close}|\dagger(k))) \\
& |\ (\nu k_3)\overline{\mathtt{CNN}}_{k_3}.(!(?x)\langle x\rangle^{\uparrow}\quad |\ k_3 \cdot (\mathsf{close}|\dagger(k)))\,\big)
\end{array}
$$

News specifies a *news collector* exposing service `collect`. After invocation of this service, a client receives all the news produced by `ANSA`, `BBC` and `CNN`. The established session can be closed: either (i) by the client-side, when an action close on the client's side is performed, as this will yield a signal $\dagger(k)$ able to activate the corresponding service-side listener $k \cdot \mathsf{close}$; or, (ii) when any of the three nested sessions used for interacting with the news services is closed by peer, yielding the signal $\dagger(k_i)$ and hence $\dagger(k)$. The termination of the topmost session will in turn cause the termination of all (not yet terminated) nested news clients.

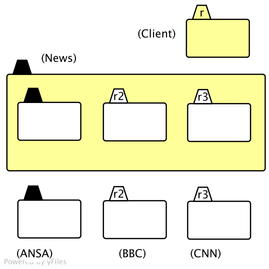
Figure 9 shows a possible propagation of termination. Initially all services have been invoked and the corresponding sessions have been triggered (Figure 9(a)). Suppose the



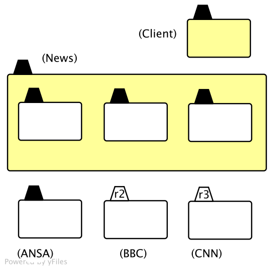
(a) All sides are active.



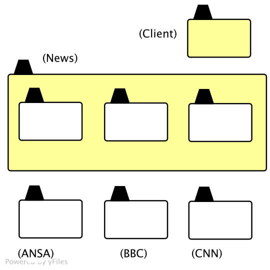
(b) `ANSA`-side terminates.



(c) `News`-side terminates.



(d) Client-side terminates.



(e) `BBC`/`CNN`-side terminate.

Fig. 9. Propagation of side-termination via listeners.

session running on `ANSA`-side shuts down (Figure 9(b)), then its partner session-side running as a child of the main session of the news collector is informed and terminated. Moreover, the listener causes the termination of the parent session (Figure 9(c)). Termination of the main session causes the termination of the remaining two children sections and of the main client-side session (Figure 9(d)). Finally, the sessions running on `BBC`-side and `CNN`-side are also terminated (Figure 9(e)).

5 Programming graceful termination

The session closing primitives introduced in the preceding section do not guarantee *per se* that forever-dangling, one-sided sessions never arise, in the same way as deadlock can arise in untyped π-calculus processes or termination cannot be guaranteed for all sequential programs. In this section, we show that this undesirable situation can be avoided if one makes sure that termination handlers of the form $k \cdot C[\mathsf{close}]$, for suitable static contexts $C[\cdot]$ and signals k, are installed in the bodies of client invocations and service definitions. We will be rather liberal with the choice of $C[\cdot]$, that may contain extra actions the termination handler may wish to take upon invocation, e.g., further signaling to other listeners (a sort of compensation, in the language of long-running transactions). For example, we anticipate that the process *News* from Example 1 fits our requirements.

The key to the proof is a concept of *balanced* term, roughly, a term with only pairs of session-sides that balance with each other. We shall prove that these terms enjoy what we call a "graceful" termination property: informally, *any possibly unbalanced term reachable from a balanced term can get balanced in a finite number of reductions*. Technically, this result will be achieved in two steps. First, we shall introduce a notion of *quasi*-balanced term, that generalizes that of balanced term. The key property here is that, differently from balanced-ness, quasi-balanced-ness is preserved through reductions. Next, we prove that any quasi-balanced term can reduce to a balanced one in a finite number of reductions. In order to define these concepts, we need to introduce some terminology about contexts.

Updated vocabulary. We revisit the terminology presented in Section 2.3 to deal with the extended syntax of full CaSPiS. A context is *static* if the hole does not occur in the scope of a dynamic operator, and *quasi-static* if the hole does not occur in the scope of a dynamic operator, except possibly the dead-session operator $\blacktriangleright$. In what follows, $D[\cdot], D'[\cdot], \ldots$ will be used to denote generic quasi-static contexts. Note that reductions and execution of close actions are permitted underneath static context but, in general, not underneath quasi-static ones. We say that $C[\cdot]$ is *session-immune* if its hole is not in the scope of a session operator.

Main definitions. In order to ensure that graceful session closing at runtime, well-formedness does not suffice, and we have to restrict our syntax in certain ways. The first step is to ensure that session termination is properly programmed inside all service definitions and invocations. That is, both on client- and service-side, proper listeners are in place.

Definition 5 (graceful property). *We say P is* graceful *if, whenever $P \equiv Q$, Q satisfies the following conditions for each s and k: (a) $s_k.$ and $\overline{s}_k.$ may only occur in Q in subterms of the form $s_k.(C[k \cdot C'[\mathsf{close}]])$ or $\overline{s}_k.(C[k \cdot C'[\mathsf{close}]])$, with $C[\cdot], C'[\cdot]$ static and session immune; (b) in Q there is at most one occurrence of the listener $k\cdot$ and one occurrence of $\rhd_k$.*

For example, obvious "graceful" usages for service invocation and service definition are $(\nu k_1)\overline{s}_{k_1}.(P_1|k_1 \cdot \mathsf{close})$ and $(\nu k_2)s_{k_2}.(P_2|k_2 \cdot \mathsf{close})$, respectively.

Lemma 2. *Let P be graceful and suppose $P \xrightarrow{\tau} Q$. Then Q is graceful.*

The proof of Lemma 2 involves some case analysis based on the following general lemma that allows us to identify active and passive components of any reduction $P \xrightarrow{\tau} Q$. (Note that, by definition, the graceful property is preserved by structural congruence.)

Lemma 3 (context lemma for reductions). *Suppose $P \xrightarrow{\tau} Q$. Then there is a 1- or 2-hole static context, $C[\cdot]$ or $C[\cdot,\cdot]$, such that one of the following cases is true (for some r, k, k', k'', s, V_i's, F_j's, P_i's, R_j's, R, P', σ, static and session-immune $C_0[\cdot], C_1[\cdot], C_2[\cdot]$ such that* $\mathrm{match}(V_l, F_h) = \sigma$*).*

1. (Sync) $P \equiv C[s_k.P,\ \overline{s}_{k'}.R]$
$Q \equiv (\nu r)C[r \rhd_{k'} P,\ r \rhd_k R]$ *r fresh for* $P, C[\cdot], R$

2. (S-Sync) $P \equiv C[r \rhd_k (P' | \sum_i \langle V_i \rangle P_i),\ r \rhd_{k'} C_0[\sum_j (F_j) R_j]]$
$Q \equiv C[r \rhd_k (P' | P_l),\ r \rhd_{k'} C_0[R_h \sigma]]$

3. (S-Sync-Ret) $P \equiv C[r' \rhd_k \big(r \rhd_{k''} C_1[\sum_i \langle V_i \rangle^{\uparrow} P_i] | P'\big),\ r \rhd_{k'} C_2[\sum_j (F_j) R_j]]$
$Q \equiv C[r' \rhd_k \big(r \rhd_{k''} C_1[P_l] | P'\big),\ r \rhd_{k'} C_2[R_h \sigma]]$

4. (P-Sync) $P \equiv C[(\sum_i \langle V_i \rangle P_i | P') > C_1[\sum_j (F_j) R_j]]$
$Q \equiv C[R_h \sigma \,|\, \big((P_l | P') > C_1[\sum_j (F_j) R_j]\big)]$

5. (P-Sync-Ret) $P \equiv C[\big((r \rhd_k C_0[\sum_i \langle V_i \rangle^{\uparrow} P_i]) \,|\, P'\big) > C_1[\sum_j (F_j) R_j]]$
$Q \equiv C[R_h \sigma \,|\, \big((r \rhd_k C_0[P_l] | P') > C_1[\sum_j (F_j) R_j]\big)]$

6. *(T-Sync)* $P \equiv C[\dagger(k) | k \cdot R]$ *and* $Q \equiv C[R]$
7. *(S-End)* $P \equiv C[r \rhd_k C_0[\mathsf{close}]]$ *and* $Q \equiv C[\blacktriangleright C_0[\mathbf{0}] | \dagger(k)]$
8. *(T-End)* $P \equiv C[\blacktriangleright (r \rhd_k R)]$ *and* $Q \equiv C[\blacktriangleright R | \dagger(k)]$

The graceful property is not sufficient to guarantee the main result about we are after, because it says nothing about balancing of existing sessions. In order to define balancing at the level of sessions, we need to introduce some more terminology.

Definition 6 (r-balancing). *Let P be a process. We say P is*

- r-balanced *if either $r \notin \mathrm{fn}(P)$ or, for some static $C[\cdot]$ not mentioning r, k and k' (with $k \neq k'$), $P \equiv C[r \rhd_k A, r \rhd_{k'} B]$ where one of the following holds for some static, session-immune $C'[\cdot]$ and $C''[\cdot]$*
  (a) $A \equiv C'[\mathsf{close}]$
  (b) $A \equiv C'[\dagger(k') | k' \cdot C''[\mathsf{close}]]$
  (c) $A \equiv C'[k' \cdot C''[\mathsf{close}]]$
  and either (a) *or* (b) *or* (c) *holds for B, with k in place of k'.*
- quasi r-balanced *if either P is r-balanced or, for some static $C[\cdot]$ not mentioning r, $P \equiv C[r \rhd_k A]$ with either of* (a) *or* (b) *above holding for A, or $P \equiv C[\blacktriangleright r \rhd_k A]$ with either of* (a) *or* (b) *or* (c) *above holding for A.*

We are now ready to give the definition of (quasi-)balancing for processes.

Definition 7 ((quasi-)balanced processes). *Assume $P \equiv (\nu \tilde{r})Q$, where Q has no top bound sessions and $\tilde{r} \subseteq \mathrm{fn}(Q)$. We say P is* balanced *(resp.* quasi-balanced*) if:*

(a) P is graceful, and
(b) for each $r \in \mathrm{fn}(Q)$, Q is r-balanced (resp. quasi r-balanced).

Clearly balancing implies quasi-balancing.

Theorem 1 (graceful termination). *Let P be balanced. Whenever $P \xrightarrow{\tau}{}^{*} P'$ there exists a balanced process Q such that $P' \xrightarrow{\tau}{}^{*} Q$.*

Proof (Sketch). The proof involves two main steps:

– First, we show that for any quasi-balanced process R, if $R \xrightarrow{\tau} R'$, then R' is also quasi-balanced.
– Second, we prove that for any quasi-balanced process R there is a balanced process R'' such that $R \xrightarrow{\tau}{}^{*} R''$.

Since P is balanced by hypothesis, then it is also quasi-balanced. Therefore P' is quasi-balanced (by straightforward induction, using the first fact above) and we can apply the second fact above to deduce the existence of a suitable Q reachable from P'.

Example 2. It is easy to observe that process *News* defined in Example 1 is balanced. This guarantees that whenever a client closes the session established for interacting with service `collect`, eventually all the nested sessions will be closed. Moreover, if the sessions that interact with the news servers are closed, the main session will be closed and the client will be notified. In fact, the example shows a "graceful" usage pattern for closing the parent session after the unexpected termination of a child session.

6 Conclusion, related work and future work

We have presented CaSPiS, a core calculus for service-oriented applications. One of the key notions of CaSPiS is that of a session, which allows for the definition of arbitrarily structured interaction modalities between clients and services. The presentation of CaSPiS have included the full formalization of the operational semantics in the SOS style and a simple discipline to program graceful session termination. A number of examples witness the expressiveness of our proposal. A recent prototype implementation is also available [2].

The design of CaSPiS has been influenced by the π-calculus [21] and by Orc [22]. Indeed, one could say that CaSPiS combines the dataflow flavour of Orc (pipelines) with the name-scoping mechanisms of the π-calculus (sessions). There are important differences with these two languages, though, that in our opinion make CaSPiS worth studying on its own. There is no notion of a session in Orc: client-service interaction is strictly request-response. Complex interaction patterns can possibly be programmed in Orc by correlating sequences of independent service invocations via some ad-hoc mechanism (e.g. state variables). Asymmetric parallel in Orc offers a way for implementing, e.g., simple cancellation patterns. However, in Orc each site invocation can return at most one value, so that cancellation has a purely local effect. Our graceful termination improves on that mechanism by dealing with nested sessions and informing any side about the termination of its opposite side. Sessions and pipelines can possibly be encoded into π-calculus, but this would certainly cost the use of several levels of "continuation channels" to specify where the results produced by a session should be sent (i.e., whether to a father session, to a pipeline or to the surrounding environment). This would make the resulting code pretty awkward (see also [3]). As our examples show, sessions and pipelines are handy constructs that is worth having as first-class objects. This fact becomes even more evident when dealing with session termination, which has no (obvious) counterpart in the π-calculus.

CaSPiS evolved from SCC [3], because the original proposal turned out to be unsatisfactory in some important respects. In particular, SCC had no dedicated mechanism for orchestrating values arising from different activities, and only had a rudimentary mechanism for handling session termination, that would immediately kill the process (and its subprocess) executing the closing action, without activating any compensation action. The first problem motivated the proposal of a few evolutions of SCC. The one proposed in [16] is *stream*-oriented, in that values produced by sessions are stored into dedicated queues, accessible by their names. The one proposed in [10] has instead dedicated message passing primitives to model communication in all directions (within a session, from inside to outside and vice-versa). As seen, CaSPiS relies solely on the concept of pipeline, but introduces pattern matching. More recently, a location-aware extension of SCC has also been proposed [6] that allows for the dynamic joining of multiparty sessions, but session termination policies are not addressed there.

A number of other proposals have been put forward, in the last couple of years, aiming, like us, at providing process calculi to support specification and analysis of services, see e.g. [17, 9, 19, 11, 25]. We do not enter into a detailed description of these proposals, which are based on rather different concepts and primitives, like *correlation sets*.

Developing safe client-service interactions requires some notion of compliance between conversation protocols. In this respect, the presence of pipelines and nested sessions makes the dynamics of a CaSPiS session quite complex and substantially different from simple type-regulated interactions as found in the π-like languages of, e.g. [14, 15], or in the finite-state contract languages of [4, 12, 13]. Two recent contributions exploring the problem of compliance in the setting of CaSPiS are [1, 8], whose type systems make evident the benefits of the concept of session. A type inference algorithm is proposed in [20].

Regarding future work, the impact of adding a mechanism of *delegation* deserves further investigation. In fact, delegation could be simply achieved by enabling session-name passing that is forbidden in the present version. However, the consequences of this choice on the semantics are at the moment not clear at all. We also plan to investigate the use of the session-closing mechanism for programming long-running transactions and related compensation policies in the context of web applications, in the vein e.g. of [7, 18], and its relationship with the cCSP and the sagas-calculi discussed in [5].

Acknowledgments. We thank the members of the SENSORIA project involved in Workpackage 2, Core Calculi for Service Oriented Computing, for stimulating discussions on Services and Calculi. Lucia Acciai and Leonardo Gaetano Mezzina read the manuscript providing us with suggestions for improvements.

References

1. L. Acciai and M. Boreale. A type system for client progress in a service-oriented calculus. In *Festschrift in Honour of Ugo Montanari, on the Occasion of His 65th Birthday*, volume 5065 of *Lect. Notes in Comput. Sci.* Springer, 2008. To appear.
2. L. Bettini, R. De Nicola, and M. Loreti. Implementing session-centered calculi with IMC. In *Proc. of COORDINATION'08*, Lect. Notes in Comput. Sci. Springer, 2008. To appear.
3. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V.T. Vasconcelos, and G. Zavattaro. SCC: a service centered

calculus. In *Proc. of WS-FM'06*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 38–57. Springer, 2006.
4. M. Bravetti and G. Zavattaro. A theory for strong service compliance. In *Proc. of COORDINATION'07*, volume 4467 of *Lect. Notes in Comput. Sci.*, pages 96–112. Springer, 2007.
5. R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *Proc. of CONCUR'05*, volume 3653 of *Lect. Notes in Comput. Sci.*, pages 383–397. Springer, 2005.
6. R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty sessions in SOC. In *Proc. of COORDINATION'08*, Lect. Notes in Comput. Sci. Springer, 2008. To appear.
7. R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: extending join. In *Proc. of IFIP TCS'04*, pages 367–379. Kluwer Academics, 2004.
8. R. Bruni and L.G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines, 2008. Submitted.
9. 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 *Lect. Notes in Comput. Sci.*, pages 327–338. Springer, 2006.
10. L. Caires, H.T. Viera, and J.C. Seco. The conversation calculus: a model of service oriented computation. Technical Report TR DIFCTUNL 6/07, Univ. Lisbon, 2007.
11. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proc. of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 2–17. Springer, 2007.
12. 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 *Lect. Notes in Comput. Sci.*, pages 148–162. Springer, 2006.
13. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *Proc. of POPL'08*, pages 261–272. ACM, 2008.
14. S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In *Proc. of ESOP'99*, volume 1576 of *Lect. Notes in Comput. Sci.*, pages 74–90. Springer, 1999.
15. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP'98*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 122–138. Springer, 1998.
16. I. Lanese, F. Martins, A. Ravara, and V.T. Vasconcelos. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM'07*, pages 305–314. IEEE Computer Society Press, 2007.
17. C. Laneve and L. Padovani. Smooth orchestrators. In *PRoc. of FoSSaCS'06*, volume 3921 of *Lect. Notes in Comput. Sci.*, pages 32–46. Springer, 2006.
18. C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proc. of FOSSACS'05*, volume 3441 of *Lect. Notes in Comput. Sci.*, pages 282–298. Springer, 2005.
19. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 33–47. Springer, 2007.
20. L.G. Mezzina. How to infer finite session types in a calculus of services and sessions. In *Proc. of COORDINATION'08*, Lect. Notes in Comput. Sci. Springer, 2008. To appear.
21. R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.
22. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 6(1):83–110, 2007.
23. Sensoria Project. Public web site. `http://sensoria.fast.de/`.
24. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
25. World Wide Web Consortium. *Web Services Choreography Description Language Version 1.0*. http://www.w3.org/TR/2005/CR-ws-cdl-10/.