Under consideration for publication in Math. Struct. in Comp. Science

CaSPiS: A Calculus of Sessions, Pipelines and Services[†]

MICHELE BOREALE,[1] ROBERTO BRUNI,[2] ROCCO DE NICOLA[1,3] and MICHELE LORETI[1]

[1] *Dipartimento di Sistemi e Informatica, Università di Firenze Viale Morgagni 65, I-50134 Firenze, Italy*
Email: {michele.boreale,michele.loreti}@unifi.it
[2] *Dipartimento di Informatica, Università di Pisa Largo Bruno Pontecorvo 3, I-56127 Pisa, Italy*
Email: bruni@di.unipi.it
[3] *IMT- Institute for Advanced Studies Lucca Piazza S. Ponziano, 6 I-55100 Lucca, Italy*
Email: rocco.denicola@imtlucca.it

Received 26 September 2011

Service-oriented computing is calling for novel computational models and languages with well disciplined primitives for client-server interaction, structured orchestration and unexpected events handling. We present CaSPiS, a process calculus where the conceptual abstractions of sessioning and pipelining play a central role for modelling service-oriented systems. CaSPiS sessions are two-sided, uniquely named and can be nested. CaSPiS pipelines permit 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 of the flexibility of the chosen set of primitives. One key contribution is a fully abstract encoding of Misra *et al.*'s orchestration language Orc. Another 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.

Contents

[†] Research supported by the Project FET-GC II IST-2005-16004 Sensoria.

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).

1.1. *Service-Oriented Computing (*SOC*)*

In SOC one of the 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. It is widely recognised that process calculi stay to concurrent computing as lambda-calculus stays to sequential computing; in fact, they lay abstract, rigorous foundations for the analysis of interactive, communicating systems. Well studied process algebras, like for instance π-calculus (Milner et al., 1992), 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. These considerations challenge the quest for novel abstractions tailored to the well-disciplined handling of first-class aspects, like long running interactions, orchestration, and handling of unexpected events.

Here, we try to center the design of a new process calculus, called CaSPiS, around a few prominent aspects of SOC applications, while reusing different patterns put forward in different 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 conversation 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 be able either to complete the interaction or to recover from errors that prevent its completion.

In SOC, the process of assembling different services to build a new one or simply to perform a specific structured task is called *orchestration*. 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 communications 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*, where suitable fields (called *correlation tokens*) of the messages exchanged during a conversation are matched to identify the specific instance of a business process in a runtime environment where several instances of the process are running (see Section 7 for references to calculi based on 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 (Misra and Cook, 2007), 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 and better suited than sequential composition to deal with concurrently executed processes and does not require the improper use of communication channels for encoding standard orchestration tasks. The main features of Orc are summarized in Section 5.1.

CaSPiS evolved from SCC *(Serviced Centered Calculus)* (Boreale et al., 2006; Wirsing and Hölzl, 2011), a calculus that arose from a coordinated effort within the EU funded project Sensoria (Sensoria Project, 2010). 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 overview our achievements.

1.2. *CaSPiS in a nutshell*

We introduce and illustrate CaSPiS primitives in an incremental way by exploiting a toy example of digitally signed documents.

Service definition and invocation. In CaSPiS, *service definitions* and *invocations* are written like (dual) action prefixes in CCS. Thus $\mathtt{sign}.P$ denotes a service $\mathtt{sign}$ whose body is process P and that can be invoked by $\overline{\mathtt{sign}}.Q$. There is an important difference w.r.t. CCS prefixes, 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 protocols. In fact, after the session has been established, we say that P and Q run at two opposite session sides (see below). As a matter of terminology, in $s.P$ and $\overline{s}.Q$, we say that s is a *service name*, $s.P$ is a *service definition* and $\overline{s}.Q$ is a *service invocation*. Slightly abusing the notation, when no confusion can arise, we refer to s and $s.P$ by the term *service*. 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* of web services (see Section 3).

In CaSPiS, services are one-shot, in the sense that when invoked, a new instance serving the request is created, but the service is no longer available. Moreover, it is possible to have different definitions $s.P_1$ and $s.P_2$ available at the same time for the same service name s. Replication (or recursion) can be used to specify persistent services, like $!s.P$. Within services and client bodies P and Q, we write $(?x)$ to input some data on variable x from the opposite side of the session and $\langle v\rangle$ to output the value v to the opposite side of the session. Values can also be returned outside the session by writing $\langle v\rangle^{\uparrow}$. Moreover, the π-calculus-like restriction operator (νn) is used to introduce fresh local names (i.e. names that are not known to the environment). As an example, the process:

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

is the parallel composition of:

— 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.

Session sides. 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 that binds 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). The name of the session has not to be mentioned in P and Q; it is handled implicitly by the operational semantics rules. Each protocol can contain other service definitions and invocations, which in turn can establish nested sessions with other peers. Sometimes, especially when type systems are considered, *polarities* $+$ and $-$ are attached to session sides in order to mark the caller and the callee respectively. In the example above, we could have written, e.g., $r^{+} \rhd P$ and $r^{-} \rhd Q$.

Rules governing creation and scoping of sessions are based on those of the restriction operator in the π-calculus. In the above case of service `sign` the triggered session is

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

Together with the session that will guarantee private communication between caller and callee, a fresh instance of the persistent service is also present. If the operator ! was missing, the service would not be persistent and the leftmost parallel component above would be absent. Note that multiple invocations to the same persistent service create separate sessions. For example $!s.P\,|\,\overline{s}.Q_1\,|\,\overline{s}.Q_2$ will evolve after two steps to $!s.P\,|\,\overline{s}.Q_1\,|\,(\nu\, r_1)(\nu\, r_2)(r_1 \rhd P\,|\,r_2 \rhd P\,|\,r_1 \rhd Q_1\,|\,r_2 \rhd Q_2)$. Hence, there is no risk of interference, because the two interactions are served by separate instances of the same service. Moreover, hierarchies of nested sessions can arise if services are invoked within an already running session side protocol (e.g. $\overline{s}_1.(P_1\,|\,\overline{s}_2.P_2)$ can lead to $r_1 \rhd (P_1\,|\,r_2 \rhd P_2)$).

Intra-session communication. After the session sides have been created, 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:

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

where we write $(\nu\, r,t)$ to abbreviate $(\nu\, r)(\nu\, t)$. Then, the digitally signed value $\{\mathsf{plan},t\}_k$ is computed at the service-side and sent back to the client-side (where y is replaced by $\{\mathsf{plan},t\}_k$):

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

Now the service side protocol is completed ($\mathbf{0}$ denotes the nil process). 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). The *return prefix* $(\langle\cdot\rangle^{\uparrow}\cdot)$ can be exploited to make the responses obtained upon some service invocation available to the parent session, as explained next.

Pipelining. 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 $P > Q$, a generalised form of Orc sequencing operator, which allows to feed Q with all values produced by P: for each value, a fresh instance of Q will be activated, running in parallel with $P > Q$. A pipeline can be seen as some sort of redirection for the outputs available in P: instead of making them available to the peer of the current session, they are fed as input to Q, which is typically guarded by some abstraction prefix. For example, Orc sequencing operator $P > x > Q$ corresponds to $P > (?x)Q$. For example, what follows is a client that invokes the service `sign` twice and then stores the obtained signed documents by invoking a suitable service `store`:

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

Let us stress the difference between the two terms $A\,|\,r \rhd R$ and $B\,|\,r \rhd R$ where $A = r \rhd (P > Q)$ and $B = (r \rhd P) > Q$. In A, outputs from P flow to Q while in B they flow to R, the other partner session side. In both cases, P inputs from R. In combination with the return operator, pipeline allows to make the responses obtained upon some service invocation available locally, to some suitable continuation.

Cancellation. 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.

However, processes must be able to abandon their current sessions in full autonomy. 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 session side that encloses it. A terminated session enters the special state $\blacktriangleright P$ that recursively terminates any other session side nested in P. Of course the execution of a close may depend on some local choice as well as be guarded by the input of some data from the opposite session side.

Closure notification. 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 on one side, a *signal* $\dagger(k_i)$ is sent to the termination handler k_i listening at the opposite side. This handler will make sure the appropriate actions are taken. 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:

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

that amounts to evolving to a state that sends the notification and closes inner sessions.

Information about the termination handlers to 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

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

Note that the caller and the callee exchange the names of their termination handlers when creating the session. If Q terminates with close, the signal $\dagger(k_2)$ is emitted to activate the termination handler k_2 of the callee; vice versa, if P terminates then k_1 will be activated.

The final ingredient is the possibility to define suitable *termination listeners* $k \cdot P$ that are used to handle termination signals $\dagger(k)$. Then

$$\dagger(k) \mid (r \rhd_{k'} P|k \cdot Q) \qquad \text{can evolve to} \qquad r \rhd_{k'} P|Q.$$

For example, $s_k.(P \mid k{\cdot}\mathsf{close})$ denotes a service that will close its session side when k is notified. Moreover, to make sure that no other listener is ever active on k, we can write $(\nu\, k)s_k.(P \mid k{\cdot}\mathsf{close})$ (assuming k not appearing in P).

Note that the emitted notification $\dagger(k)$ is essentially *asynchronous*, i.e., we have no guarantee as to when the listener at the opposite side will catch $\dagger(k)$. For example, before $\dagger(k)$ 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. While dangling $\dagger(k)$ cannot be avoided in general, simple patterns can avoid the worse situation of dangling session sides pending forever.

The mechanism of termination handlers is very expressive and flexible. We emphasize that, up to our knowledge, there are no similar mechanisms 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 plain π-calculus.

Pattern matching and guarded choice. Last but not least, CaSPiS interactions is empowered by pattern-matching facilities that can be suited, e.g., to deal with XML-like data typical of web service scenarios (Acciai and Boreale, 2008c). Roughly, this is obtained by allowing the use of structured values in output and return prefixes together with pattern matching in input prefixes. Together with ordinary prefix-guarded choices, the presence of patterns makes it possible to manage and route messages on the basis of their contents. For example, a pipeline like $P > (\mathsf{pdf}(?x))Q + (\mathsf{ps}(?x))R$ can be used to handle in different ways the documents produced by P depending on whether they are in Portable Document Format or in PostScript.

1.3. *Structure of the paper*

We start by presenting the syntax and reduction semantics of the close-free fragment of CaSPiS in Section 2 together with many examples (see Section 3) and some important behavioural equivalences in Section 4. Section 5 presents our first main result, namely the fully abstract encoding of Orc in CaSPiS. The cancellation primitives are detailed in Section 6, where the second main result of the paper is given, namely a class of processes is defined for which it is guaranteed that, even in the presence of cancellation primitives, no session side will hang forever after the closure of the opposite side. We call such propety *graceful termination*. Related work is discussed in Section 7, while concluding remarks and future work are summarized in Section 8. A few technical proofs have been confined to Appendix A.

Origin of the material.

This paper is the full version of (Boreale et al., 2008), where graceful termination was first considered. Here we extend that paper with the definition of the abstract semantics in Section 4, the encoding of Orc, which is original to this paper and the formal proofs of graceful termination theorems. The e-shop example in Section 3 is inspired to an example in (Bodei et al., 2009). This paper substantially extends and revisits the examples in the summer school tutorial (Bruni, 2009), where CaSPiS was discussed at the informal level.

2. CaSPiS: the close-free fragment

2.1. *Syntax*

We start by presenting the fragment of CaSPiS without cancellation and closure notification. The actual syntax is in Fig. 1. There we let $\mathcal{N}_{\mathrm{srv}}$ and $\mathcal{N}_{\mathrm{sess}}$ be two disjoint countable sets, respectively of *service* names, ranged over by s and of *session* names, ranged over by r. We assume $\mathcal{N}_{\mathrm{srv}}$ and $\mathcal{N}_{\mathrm{sess}}$ are included in a larger set of *names* $\mathcal{N}$, ranged over by n. Finally, we let

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

Fig. 1. Syntax of close-free CaSPiS

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

Fig. 2. Structural congruence laws

$x, y, ..., u, v...$ range over $\mathcal{N} \setminus \mathcal{N}_{\text{sess}}$. We write $\tilde{t}$ for the tuple (of names, values, patterns, ...) $t_1, ..., t_n$. The operators in Fig. 1 are listed in decreasing order of precedence. Their informal meaning has been already discussed in Section 1.2. The empty sum is the *nil* process, denoted $\mathbf{0}$. Trailing $\mathbf{0}$'s will often be omitted. When the arguments of prefixes are void or inessential, we abbreviate them as $()P$, $\langle\rangle P$ and $\langle\rangle^{\uparrow} P$. As usual, we write $(\nu n_1, n_2, ..., n_m)P$ or just $(\nu \tilde{n})P$ as a shorthand for $(\nu n_1)(\nu n_2)...(\nu n_m)P$.

As expected, in $(\nu n)P$, the restriction (νn) binds free occurrences of n in P, while in $(F)P$ any $?x$ 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. The usual notion of *free names* $\mathrm{fn}(P)$ and *bound names* $\mathrm{bn}(P)$ of a process P are defined in the standard way. We say a name n is *fresh* for P if n does not appear in P, i.e. $n \notin \mathrm{fn}(P) \cup \mathrm{bn}(P)$.

Service definition $s.[\,\cdot\,]$ and invocation $\overline{s}.[\,\cdot\,]$, prefix $\pi_i[\,\cdot\,]$, left-sided pipeline $P > [\,\cdot\,]$ and replication $![\,\cdot\,]$ are called *dynamic* operators, because when they are applied to some process P, then P cannot execute until they are removed. The remaining operators are called *static*, because when they are applied to some process P, then P is immediately active and the static operators are not removed while P evolves.

The *structural congruence* relation $\equiv$ is defined as the least congruence that includes alpha-equivalence (i.e. equivalence up to the capture-avoiding renaming of some bound names) and the laws in Fig. 2. This set of laws comprises the structural rules for parallel composition and restriction, and the obvious extension of restriction's scope extrusion law to pipelines and sessions. Roughly, structural congruence allow us to abstract away from the particular order in which processes are composed in parallel and in which we restrict some of their names.

2.2. *Reduction semantics*

The operational semantics of CaSPiS processes is defined by exploiting suitable contexts surrounding the active redexes. We let a *context*, written $\mathbb{C}[\,\cdot\,]$, be a process term with one hole $[\,\cdot\,]$. We write $\mathbb{C}[\,P\,]$ for the process where the hole is textually replaced by process P. We say that the term Q *occurs in* (or is a subterm of) P if there is a context $\mathbb{C}[\,\cdot\,]$ such that $P = \mathbb{C}[\,Q\,]$. Sometimes, two-holes contexts $\mathbb{C}[\,\cdot,\cdot\,]$ are also used. The contexts we are interested in are the static ones.

Definition 2.1 (Static and immune contexts). A context is *static* if its hole does not occur under a dynamic operator. Moreover, we say that a static context is *session-immune* if its hole does not occur under a session operator, and *pipeline-immune* if its hole does not occur under in the right-hand side of a pipeline operator.

Intuitively, static contexts are characterised by the fact that the hole occurs in an actively running position and it is ready to interact (e.g. it is not under a prefix). In the following we let $\mathbb{C}[\,\cdot\,]$ range over static contexts, $\mathbb{S}[\,\cdot\,]$ over session-immune contexts, and $\mathbb{P}[\,\cdot\,]$ over contexts that are both session- and pipeline-immune. Roughly, a session-immune context $\mathbb{S}[\,\cdot\,]$ cannot "intercept" abstraction and return prefixes, while a session- and pipeline-immune context $\mathbb{P}[\,\cdot\,]$ cannot "intercept" concretion prefixes either. Analogous definitions apply to the case of two-holes contexts $\mathbb{C}[\,\cdot,\cdot\,]$.

Lemma 2.1. Let P, Q be such that $P \equiv \mathbb{P}[\,Q\,]$ for some session- and pipeline-immune context $\mathbb{P}[\,\cdot\,]$. Then, $P \equiv (\nu\,\tilde{n})(Q\sigma\,|\,R)$ for some names $\tilde{n}$, process R and renaming σ.

Proof. The thesis follows immediately by noting that the hole in $\mathbb{P}[\,\cdot\,]$ can occur only under restriction and parallel composition and that the restrictions can be moved to the top by structural congruence laws (exploiting alpha-equivalence when needed, which justifies the presence of σ). $\square$

We say that a name n is *bound* by the context $\mathbb{C}[\,\cdot\,]$ if the hole is in the scope of a binder $(\nu\,n)$. Moreover, we say that the name n is *coupled* in the two-holes context $\mathbb{C}[\,\cdot,\cdot\,]$ if whenever one hole is in the scope of a binder $(\nu\,n)$ then also the other hole is in the scope of the same binder, i.e., each binder $(\nu\,n)$ appearing in $\mathbb{C}[\,\cdot,\cdot\,]$ either binds the name n in both holes or in none of them. From now on we shall assume that in all two-holes contexts $\mathbb{C}[\,\cdot,\cdot\,]$ mentioned in the reduction rules all names are coupled. Intuitively, this means that if a restricted name is communicated, then its scope has already been extruded to the recipient by exploiting structural congruence.

Below we let $\mathbb{C}_r[\,\cdot,\cdot\,]$ be a context of the form $\mathbb{C}[\,r \rhd \mathbb{P}[\,\cdot\,], r \rhd \mathbb{S}[\,\cdot\,]\,]$ for some static context $\mathbb{C}[\,\cdot\,]$ some session-immune context $\mathbb{S}[\,\cdot\,]$ and some session- and pipeline-immune context $\mathbb{P}[\,\cdot\,]$. (A context of the form $\mathbb{C}_r[\,\cdot,\cdot\,]$ captures the most general situation in which intra-session communication can happen.)

The first reduction rule regards the handshake between a service definition and a service invocation.

$$(\textsc{sync})\,\frac{r\ \text{ fresh for }\ \mathbb{C}[\,P, Q\,]}{\mathbb{C}[\,\overline{s}.P, s.Q\,] \xrightarrow{\tau} (\nu\,r)\mathbb{C}[\,r \rhd P, r \rhd Q\,]}$$

The second reduction rule regards intra-session communication. Pattern-matching is accounted for by a substitution $\sigma = \text{match}(F, V)$, defined as the (only) substitution such that $\text{dom}(\sigma) = \text{bn}(F)$ and $F\sigma = V$.

$$(\textsc{Ssync})\,\frac{\sigma = \text{match}(F, V)}{\mathbb{C}_r[\,\langle V\rangle P + \sum_i \pi_i P_i\,,\ (F)Q + \sum_j \pi_j Q_j\,] \xrightarrow{\tau} \mathbb{C}_r[\,P\,,\ Q\sigma\,]}$$

Intra-session communication can be triggered also by a return prefix in a subsession of r. The corresponding reduction rule is:

$$(\textsc{SRsync})\frac{\sigma = \text{match}(F, V)}{\mathbb{C}_r[\, r_1 \rhd \mathbb{S}_1[\, \langle V\rangle^{\uparrow}P + \sum_i \pi_i P_i \,]\,,\ (F)Q + \sum_j \pi_j Q_j \,] \xrightarrow{\tau} \mathbb{C}_r[\, r_1 \rhd \mathbb{S}_1[\, P\,]\,,\ Q\sigma\,]}$$

There are two reduction rules for pipeline orchestration, handling the “redirection” of concretions and returns.

$$(\textsc{Psync})\frac{Q = \mathbb{S}[\,(F)Q' + \sum_j \pi_j Q_j\,] \quad \sigma = \text{match}(F, V)}{\mathbb{C}[\, \mathbb{P}[\, \langle V\rangle P + \sum_i \pi_i P_i \,] > Q\,] \xrightarrow{\tau} \mathbb{C}[\, \mathbb{S}[\, Q'\sigma\,] \,|\, (\mathbb{P}[\, P\,] > Q)\,]}$$

$$(\textsc{PRsync})\frac{Q = \mathbb{S}[\,(F)Q' + \sum_j \pi_j Q_j\,] \quad \sigma = \text{match}(F, V)}{\mathbb{C}[\, \mathbb{P}[\, r \rhd \mathbb{S}_1[\, \langle V\rangle^{\uparrow}P + \sum_i \pi_i P_i \,]\,] > Q\,] \xrightarrow{\tau} \mathbb{C}[\, \mathbb{S}[\, Q'\sigma\,] \,|\, (\mathbb{P}[\, r \rhd \mathbb{S}_1[\, P\,]\,] > Q)\,]}$$

Finally, we have the rule for structural congruence, familiar from the π-calculus.

$$(\textsc{Struct})\frac{P \equiv P' \qquad P' \xrightarrow{\tau} Q' \qquad Q' \equiv Q}{P \xrightarrow{\tau} Q}$$

We write $P(\xrightarrow{\tau})^*Q$ if there is a (possibly empty) finite sequence of reductions $P_0 \xrightarrow{\tau} P_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} P_n$ such that $P \equiv P_0$ and $Q \equiv P_n$, i.e. $(\xrightarrow{\tau})^*$ denotes the reflexive and transitive closure of $\xrightarrow{\tau}$.

An alternative operational semantics based on a labelled transition system with also non silent transitions has been defined in (Boreale et al., 2008), together with the Lemma that reconciles the silent transitions of the two semantics. For the aim of this paper, presenting both semantics would be redundant and therefore we prefer to give the reduction semantics only, that is more concise.

2.3. *Well-formedness*

As anticipated, we introduce a well-formedness criterion to impose some discipline on the way SCC 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 $r \notin \text{fn}(P')$, for each P', r.

The key technical ingredient needed for our definition of well-formed process is a scope containment relation on session names.

Definition 2.2. Given P and two session names $r, r' \in \text{fn}(P)$, we write $r \prec_P r'$ if and only if $P \equiv \mathbb{C}[\, r \rhd \mathbb{S}[\, r' \rhd Q\,]\,]$ for some static context $\mathbb{C}[\,\cdot\,]$, session-immune $\mathbb{S}[\,\cdot\,]$ and process Q.

In other words, $r \prec_P r'$ if, in P, up to structural congruence, a side of session r' is immediately within the scope of some session side $r \rhd [\,\cdot\,]$.

Definition 2.3 (well-formedness). Assume $P \equiv (\nu\, \tilde{r})Q$, where Q has no top bound sessions and $\tilde{r} \subseteq \text{fn}(Q)$. Then, process P is *well-formed* if: (a) the relation $\prec_Q$ is acyclic, (b) for each name

$r_i \in \tilde{r}$, ther are at most two occurrences $r_i \rhd P_1$ and $r_i \rhd P_2$ occurring in Q and they are not 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 $\prec_Q$ (meaning that $\prec_Q^+$ is irreflexive) rules out vicious situations like a session side running inside its partner's side (e.g. $r \rhd (P_1 | r \rhd P_2)$ is not well-formed). 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 6). 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 2.2. Well-formedness is preserved by structural congruence and reductions.

3. Examples

It can be easily shown 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 into π-calculus (details can be found in (Boreale et al., 2006)).

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

$$[\![a(x).P]\!] \triangleq a.(?x)\langle x\rangle^{\uparrow} > (?x)[\![P]\!] \qquad [\![\overline{a}v.P]\!] \triangleq \overline{a}.\langle v\rangle\langle\rangle^{\uparrow} > ()[\![P]\!]$$

Note that in both cases a value is returned (possibly void) that serves to activate the suitable continuation, i.e. the (encoded) prefixed process. The encoding can then be extended homomorphically to parallel composition, restriction and replication.

In this section we present some further 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 `!ANSA.!`$(\nu n)\langle n\rangle$, etc.).

Invocation patterns. Recurrent service invocation patterns are reported in Figure 3:

— $\overline{s}\langle V\rangle$ 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);
— $P > \overline{s}$ 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.

$$\begin{array}{rcll@{\qquad}rcll}
\overline{s}\langle V\rangle & \triangleq & \overline{s}.\langle V\rangle & \text{(One Way)} & P > \overline{s} & \triangleq & P > (?x)\overline{s}(x) & \text{(Simple Pipe)}\\
\overline{s}(V) & \triangleq & \overline{s}.\langle V\rangle(?x)\langle x\rangle^{\uparrow} & \text{(Request Response)} & \overline{s}(!) & \triangleq & \overline{s}.!(?x)\langle x\rangle^{\uparrow} & \text{(Get All Responses)}
\end{array}$$

Fig. 3. SCC derivable constructs

Using the request-response and one-way patterns we can rewrite the example in the Introduction as: $\overline{\texttt{sign}}(\textsf{plan}) > (?z)\overline{\texttt{store}}\langle z\rangle$. Simple pipe can be used to write the signing of a document by two different authorities as $\langle\textsf{plan}\rangle > \overline{\texttt{sign}_1} > \overline{\texttt{sign}_2}$. 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

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

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:

$$\begin{aligned}
\textbf{wait } F_1, \ldots, F_n \textbf{ from } P_1, \ldots, P_n \ \triangleq\ (\nu\, s)(\ & s.(t_1(F_1)).\ldots.(t_n(F_n))\langle \hat{F}_1, \ldots, \hat{F}_n\rangle^{\uparrow}\\
& \mid \overline{s}.(\ \textbf{select } F_1 \textbf{ from } P_1 \textbf{ in } \langle t_1(\hat{F}_1)\rangle \mid \cdots\\
& \qquad \mid \textbf{select } F_n \textbf{ from } P_n \textbf{ in } \langle t_n(\hat{F}_n)\rangle\ )\ )
\end{aligned}$$

We can also consider the following variation:

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

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:

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

After invocation, one message among three possible kinds of requests is expected: $\mathit{fly}(V)$, $\mathit{hotel}(V)$ and $\mathit{fly}\&\mathit{hotel}(V)$, where value V contains trip information. For example, in 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 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.

Proxy. To show some name passing capabilities, we consider 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 in response:

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

E-shop. We conclude this section by revisiting the electronic commerce example from (Bodei et al., 2009), where it was used to illustrate a static analysis machinery for the detection of logic flaws in service applications, i.e. to prevent the so-called *application logic attacks* that exploit the vulnerabilities of the specific functionality of the application (e.g., by violating the business logic) rather than the ones of the underlying platform.

We model a simple e-shop application S that exchanges information with customers C and the data base D that stores item prices. Service *price*, which is used to retrieve item prices, is private to S and D. Essentially, a honest customer invokes service *buy*, chooses an item, receives its price inside an order form and if interested in finalising the order, must fill in a payment form with personal data and credit card information. In the same form are reported: the transaction code, the chosen item, and the received price of the purchase.

$$HC \quad \triangleq \quad \overline{\mathit{buy}}.\langle \mathsf{item}_k\rangle(\mathsf{orderForm}(?x_{code}, \mathsf{item}_k, ?x_{price_k}))\langle \mathsf{payForm}(x_{code}, \mathsf{item}_k, x_{price_k}, \mathsf{name}, \mathsf{cc})\rangle$$

However, a malicious user may try to finalise the transaction sending a forged copy of the payment form, where the price field has been abusively discounted (like when downloading a web order form associated with an e-shopping cart, editing some hidden field outside the browser and resubmitting it in place of the original one).

$$MC \triangleq \overline{buy}.\langle \mathsf{item}_k\rangle(\mathsf{orderForm}(?x_{code}, \mathsf{item}_k, ?x_{price_k}))\langle \mathsf{payForm}(x_{code}, \mathsf{item}_k, \mathsf{5cents}, \mathsf{name}, \mathsf{cc})\rangle$$

In the specification shown below, the application S exploits, for each item, two concurrent processes OF_i and PF_i, respectively for sending the order form to the customer and for receiving the cancellation or the payment of the order. This way it cannot check if the form sent by the customer contains the right price.

$$\begin{array}{rcl}
ESHOP & \triangleq & (\nu\, price)(D \,|\, S) \\
D & \triangleq & !price.\, \sum_i (\mathsf{item}_i)\langle \mathsf{price}_i\rangle \\
S & \triangleq & !buy.\, \sum_i (\mathsf{item}_i)(\nu\, code)(OF_i \,|\, PF_i) \\
OF_i & \triangleq & \overline{price}.\langle \mathsf{item}_i\rangle\ (?x_{price_i})\langle \mathsf{orderForm}(code, \mathsf{item}_i, x_{price_i})\rangle^{\uparrow} \\
PF_i & \triangleq & (\mathsf{cancel})\mathbf{0} + (\mathsf{payForm}(code, \mathsf{item}_i, ?y_{price_i}, ?y_{name}, ?y_{cc}))PAY
\end{array}$$

Then both the honest customer HC and the malicious customer MC shown above are capable to interact with the application, each fulfilling their purposes.

We can exploit pipeline redirection to redesign the e-shop application S in such a way that the price indicated by the customer in the payment form is matched against the one provided by the data base. We just modify processes S, OF_i and PF_i as follows:

$$\begin{array}{rcl}
S & \triangleq & !buy.\, \sum_i (\mathsf{item}_i)(\nu\, code)S_i \\
S_i & \triangleq & (\overline{price}.\langle \mathsf{item}_i\rangle(?x_{price_i})\langle x_{price_i}\rangle^{\uparrow}\ ) > (?x_{price_i})(OF_i \,|\, PF_i) \\
OF_i & \triangleq & \langle \mathsf{orderForm}(code, \mathsf{item}_i, x_{price_i})\rangle \\
PF_i & \triangleq & (\mathsf{cancel})\mathbf{0} + (\mathsf{payForm}(code, \mathsf{item}_i, x_{price_i}, ?y_{name}, ?y_{cc}))PAY
\end{array}$$

4. Behavioural Relations

CaSPiS structural congruence allow us to ignore inessential details in process terms, like the exact order in which processes are composed in parallel or the exact order in which names are restricted. However, there are many other ways in which processes with completely different structures behave essentially the same. In this section we define some *behavioural relations* over processes that will be handful to prove the correctness of Orc's encoding.

The first ingredient is a notion of an *observable* (also called *barb*), i.e., some notable capability of a process to interact with its environment, like the names of the services that can be invoked, or the kind of values that can be given in output, taken in input or returned one level up to the parent session. In analogy with Orc, we are only interested in the possible values that a process can output. In fact, the encoding of any Orc expression will be a process that cannot perform any input or return (see Lemma 5.1).

Definition 4.1 (CaSPiS barbs). We write $P \downarrow (\nu\tilde{n})V$, and say that P *has barb* $(\nu\tilde{n})V$, if (i) $P \equiv \mathbb{P}[\,\langle V\rangle.Q + R\,]$ or (ii) $P \equiv \mathbb{P}[\,r \rhd \mathbb{S}[\,\langle V\rangle^{\uparrow}.Q + R\,]\,]$ for some processes Q and R, some session name r, some session- and pipeline-immune context $\mathbb{P}[\cdot]$ and some session-immune context $\mathbb{S}[\cdot]$, such that $n_i \in \tilde{n}$ iff n_i is bound in $\mathbb{P}[\cdot]$ or $\mathbb{S}[\cdot]$. Moreover, if case (i) applies we also write $P \xrightarrow{(\nu\tilde{n})\langle V\rangle} P'$ for any $P' \equiv \mathbb{P}[\,Q\,]$, while if (ii) applies we write $P \xrightarrow{(\nu\tilde{n})\langle V\rangle} P''$ for any $P'' \equiv \mathbb{P}[\,r \rhd \mathbb{S}[\,Q\,]\,]$.

A weak barb defines the possibility to output a value after a finite number of reductions.

Definition 4.2 (CaSPiS weak barbs). We write $P \Downarrow (\nu\tilde{n})V$, and say that P *has weak barb* $(\nu\tilde{n})V$, if there is a process Q such that $P(\xrightarrow{\tau})^{*}Q \downarrow (\nu\tilde{n})V$.

The second ingredient is the way in which we compare two processes based on the capabilities they possess.

Definition 4.3 (Strong barbed bisimilarity). We let *strong barbed bisimilarity* over CaSPiS processes, denoted $\dot{\sim}$, be the largest binary symmetric relation such that whenever $P \dot{\sim} Q$ then:

1 whenever $P \downarrow (\nu\tilde{n})V$ and $\tilde{n} \cap \mathrm{fn}(Q) = \emptyset$ then $Q \downarrow (\nu\tilde{n})V$, and
2 whenever $P \xrightarrow{\tau} P'$ then there is Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \dot{\sim} Q'$.

Definition 4.4 (Weak barbed bisimilarity). We let *weak barbed bisimilarity* over CaSPiS processes, denoted $\dot{\approx}$, be the largest binary symmetric relation such that whenever $P \dot{\approx} Q$ then:

1 whenever $P \downarrow (\nu\tilde{n})V$ and $\tilde{n} \cap \mathrm{fn}(Q) = \emptyset$ then $Q \Downarrow (\nu\tilde{n})V$, and
2 whenever $P \xrightarrow{\tau} P'$ then there is Q' such that $Q(\xrightarrow{\tau})^{*}Q'$ and $P' \dot{\approx} Q'$.

We remark that $\dot{\sim}$ and $\dot{\approx}$ are equivalence relations (i.e. reflexive, symmetric and transitive), with $\dot{\sim}\subseteq\dot{\approx}$ and that they trivially include structural congruence.

Below, as a matter of notation, we write $P \xrightarrow{\hat{\tau}} Q$ if either $P \xrightarrow{\tau} Q$ or $P \equiv Q$.

Definition 4.5 (Expansion preorder). We let *expansion preorder* over CaSPiS processes, denoted $\dot{\gtrsim}$, be the largest binary relation such that whenever $P \dot{\gtrsim} Q$ then:

1 whenever $P \downarrow (\nu\tilde{n})V$ and $\tilde{n} \cap \mathrm{fn}(Q) = \emptyset$ then $Q \downarrow (\nu\tilde{n})V$, and
2 whenever $Q \downarrow (\nu\tilde{n})V$ and $\tilde{n} \cap \mathrm{fn}(P) = \emptyset$ then $P \Downarrow (\nu\tilde{n})V$, and
3 whenever $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \xrightarrow{\hat{\tau}} Q'$ and $P' \dot{\gtrsim} Q'$, and
4 whenever $Q \xrightarrow{\tau} Q'$ then there is P' such that $P(\xrightarrow{\tau})^{*}P'$ and $P' \dot{\gtrsim} Q'$.

When $P \dot{\gtrsim} Q$ we say that P *expands to* Q.

We remark that $\dot{\gtrsim}$ is a preorder (i.e. reflexive and transitive) and that $\dot{\sim} \subseteq \dot{\gtrsim} \subseteq \dot{\approx}$

The above defined relations are not congruences in general. A standard way to tackle this drawback is to close them under all possible (static) contexts. This gives rise to three finer relations, that we denote with the symbols $\sim$, $\gtrsim$, and $\approx$. For example, we let $P \approx Q$ if and only if, for each static context $\mathbb{C}[\ ]$, (i) $\mathbb{C}[\,P\,]$ is well-formed iff $\mathbb{C}[\,Q\,]$ is well-formed and (ii) $\mathbb{C}[\,P\,] \dot{\approx} \mathbb{C}[\,Q\,]$.

The following auxiliary lemmas assert some properties of $\gtrsim$ and $\sim$ that will serve to prove the correspondence results between Orc processes and their encoding in CaSPiS. Their proofs are tedious but standard, as they go along the lines of analogous results for the π-calculus (see e.g. (Boreale and Sangiorgi, 1998).

Lemma 4.1. For any process P, we have $\mathbf{0} > P \sim \mathbf{0}$.

We say that a process is *return free* if any occurrence of $\langle V\rangle^{\uparrow}$ in P appears under a service definition or a session side. Similarly, we say that a process is *input free* if any occurrence of (F) in P appears under a service definition or a session side or at the top of the right-hand side of a pipeline. We say that P is an *emitter* if it is both return and input free.

We say that a service name s is *not provided* by P if: (i) if no service definition $s.Q$ occurs in P and (ii) s does not appear in any occurrence of $\langle V\rangle$ and $\langle V\rangle^{\uparrow}$ in P.

Lemma 4.2. For any service name s, any process P such that s is not provided by P and any process Q such that $P(\xrightarrow{\tau})^{*}Q$, then the name s is not provided by Q.

Lemma 4.3. Let P, Q, R be processes not providing s. We have that:

1 $(\nu s)(\, P \,|\, Q \,|\, !s.R\,) \sim (\nu s)(\, P \,|\, !s.R\,) \,|\, (\nu s)(\, Q \,|\, !s.R\,)$;

2 if R is return free then $(\nu s)(\,(P > Q) \,|\, !s.R\,) \sim (\nu s)(\, P \,|\, !s.R\,) > (\nu s)(\, Q \,|\, !s.R\,)$;

3 if R is return free then $(\nu s)(\,(r \rhd P) \,|\, Q \,|\, !s.R\,) \sim (\, r \rhd (\nu s)(\, P \,|\, !s.R\,)\,) \,|\, (\nu s)(\, Q \,|\, !s.R\,)$.

Lemma 4.4. For any return free processes P, Q and session name r, $(\nu r)(\, r \rhd P \,|\, r \rhd !s.Q\,) \gtrsim !s.Q$.

5. Encoding Orc

In (Misra and Cook, 2007; Cook et al., 2006; Kitchin et al., 2006), Cook and Misra have proposed a basic programming language for structured orchestration, called Orc, whose primitives join simplicity with generality and are quite different from the communication centric primitives found in process calculi literature, where control and data flows are always encoded in terms of interaction. Orc neatly separates orchestration from computation: Orc expressions e are considered as scripts to be invoked, e.g., within imperative programming languages using assignments such as $z :\in e$, where z is a variable and the Orc expression e can involve wide-area computation over multiple servers. (The assignment symbol $:\in$ (due to Tony Hoare) makes it explicit that e can return zero or more results, one of which is assigned to z.) Even if Orc looks quite different from ordinary process calculi, it relies anyway on hidden mechanisms for name handling (creation and passing) and for atomic distributed termination.

For the above reasons, we think it is interesting to see if and how elegantly can CaSPiS formally encode Orc constructs, which is the main goal of this section.

We first recap Orc basics, borrowing definitions from (Misra and Cook, 2007) and then define the encoding of Orc expressions as CaSPiS processes and prove it is adequate by establishing a tight semantics correspondence.

5.1. *The Orc language*

We let Orc expressions be defined by the following grammar:

$$
\begin{array}{llllllllllll}
\text{(Expressions)} & e & ::= & \mathbf{0} & \mid & b & \mid & e_1 \,|\, e_2 & \mid & e_1 > x > e_2 & \mid & e_2 \ \mathbf{where}\ x :\in e_1 \\
\text{(Basic expr.)} & b & ::= & \langle p\rangle & \mid & x\langle \tilde{p}\rangle & \mid & s\langle \tilde{p}\rangle & \mid & E\langle \tilde{p}\rangle & \mid & ?^{s}_{\tilde{v}}
\end{array}
$$

where we assume given the following (pairwise disjoint) sets: a set $\mathcal{V}$ of *values*, ranged over by v, a set X of *variables*, ranged over by x, a finite set $\mathcal{S}$ of *sites*, ranged over by s, and a set $\mathcal{E} \triangleq \{\ E_i(\tilde{x}) \triangleq e_i\ \}_i$ of *defined expressions*, ranged over by E. Moreover, we let the set of *parameters* $\mathcal{P} \triangleq \mathcal{V} \cup \mathcal{S} \cup X$ be ranged over by p. The expressions $e_1 > x > e_2$ and e_2 **where** $x :\in e_1$ bind the occurrences of x in e_2. The occurrences of non-bound variables are free and the set of free variables of an expression e is denoted by fv(e). All defined expressions $E(\tilde{x}) \triangleq e$ are *well-formed*, in the sense that $\mathrm{fv}(e) \subseteq \tilde{x}$.

The basic computational entities orchestrated by Orc expressions are *sites*: a site call can be thought of as an RMI, a call to a monitor procedure, or to a function or to a (web) service. Each invocation to site s elicits at most one value published by s. (Note instead that, in principle, an Orc expression can publish any number of values.) Values are published locally using the primitive *let*(v), here rendered just as $\langle v \rangle$ for brevity. In the following we let $resp(s, \tilde{v})$ denote the (possibly empty) set of responses to the invocation of site s with actual parameters $\tilde{v}$.

Orc semantics is defined in the LTS style, via SOS rules. The basic expressions $?^s_{\tilde{v}}$ must be considered as run-time syntax: they denote response handlers for site invocations. The set of labels include action $\langle v \rangle$ for the local publication of value v, and the silent action τ. We let o range over labels.

Site call is strict, in the sense that actual parameters must have been evaluated before the call is issued. A site call $s\langle \tilde{v} \rangle$ is handled in three phases: first the call is issued, leading to the installation of a response handler $?^s_{\tilde{v}}$; second the response $w \in resp(s, \tilde{v})$, if any, arrives; third the received value is made available (*published*) locally. The corresponding rules are shown below.

$$(\text{CALL})\ \frac{}{s\langle \tilde{v} \rangle \xrightarrow{\tau} ?^s_{\tilde{v}}} \qquad (\text{RESP})\ \frac{w \in resp(s, \tilde{v})}{?^s_{\tilde{v}} \xrightarrow{\tau} \langle w \rangle} \qquad (\text{LET})\ \frac{}{\langle v \rangle \xrightarrow{\langle v \rangle} \mathbf{0}}$$

The evaluation of defined expressions is non-strict. The corresponding inference rule is:

$$(\text{DEF})\ \frac{E_i(\tilde{x}) \triangleq e_i \in \mathcal{E}}{E_i\langle \tilde{p} \rangle \xrightarrow{\tau} e_i\{\tilde{p}/\tilde{x}\}}$$

Orc has three composition principles. The first one is the ordinary parallel composition $e_1 \,|\, e_2$, here called *symmetric parallel* (e.g., the parallel composition of two site calls can produce zero, one or many values). We remark that there is no interaction between e_1 and e_2. The corresponding inference rules are:

$$(\text{LSYM})\ \frac{e_1 \xrightarrow{o} e_1'}{e_1 \,|\, e_2 \xrightarrow{o} e_1' \,|\, e_2} \qquad (\text{RSYM})\ \frac{e_2 \xrightarrow{o} e_2'}{e_1 \,|\, e_2 \xrightarrow{o} e_1 \,|\, e_2'}$$

The second composition principle is called *sequencing* and it takes inspiration from universal quantification: in the sequential expression $e_1 > x > e_2$, a fresh copy $e_2\{v/x\}$ of e_2 is spawned for *any* value v published by e_1, i.e., a sort of pipeline is established between e_1 and e_2. When $x \notin \mathrm{fv}(e_2)$, then we write $e_1 >> e_2$ as a shorthand for $e_1 > x > e_2$ (because x is inessential). The corresponding inference rules are:

$$(\textsc{seq})\ \frac{e_1 \xrightarrow{\tau} e'_1}{e_1 > x > e_2 \xrightarrow{\tau} e'_1 > x > e_2} \qquad (\textsc{pipe})\ \frac{e_1 \xrightarrow{\langle v\rangle} e'_1}{e_1 > x > e_2 \xrightarrow{\langle v\rangle} (e'_1 > x > e_2)\,|\, e_2\{{}^{v}\!/\!_{x}\}}$$

The third and last composition principle is called *asymmetric parallel composition* and takes inspiration from existential quantification. The evaluation of the asymmetric parallel expression e_2 **where** $x :\in e_1$ (written as $e_2 < x < e_1$ in the latest papers on Orc) is lazy: e_1 and e_2 start in parallel, but all sub-expressions of e_2 that depend on the value of x must wait for e_1 to publish *one* value. When e_1 produces a value it is assigned to x and that side of the orchestration is cancelled. The corresponding inference rules are:

$$(\textsc{lasym})\ \frac{e_2 \xrightarrow{o} e'_2}{e_2 \textbf{ where } x :\in e_1 \xrightarrow{o} e'_2 \textbf{ where } x :\in e_1}$$

$$(\textsc{rasym})\ \frac{e_1 \xrightarrow{\tau} e'_1}{e_2 \textbf{ where } x :\in e_1 \xrightarrow{\tau} e_2 \textbf{ where } x :\in e'_1} \qquad (\textsc{pick})\ \frac{e_1 \xrightarrow{\langle v\rangle} e'_1}{e_2 \textbf{ where } x :\in e_1 \xrightarrow{\tau} e_2\{{}^{v}\!/\!_{x}\}}$$

5.2. *Orc examples*

We borrow from (Misra and Cook, 2007) some simple examples of Orc declarations. In the following we assume the existence of two sites *cnn* and *bbc* to be invoked with a date d as argument and that return selected news from date d, and of a site *email* that requires two arguments m and a and sends an email containing message m to the address a without returning any value. Moreover, we write $v_1 :: v_2$ to denote the concatenation of two messages.

— Declaration $\mathit{MailTwice}(a,d) \triangleq (\mathit{cnn}\langle d\rangle | \mathit{bbc}\langle d\rangle) > x > \mathit{email}\langle x,a\rangle$ specifies a service for notifying all news from *cnn and bbc* in two different emails.

— Declaration $\mathit{MailOnce}(a,d) \triangleq \mathit{email}\langle x,a\rangle$ **where** $x :\in (\mathit{cnn}\langle d\rangle | \mathit{bbc}\langle d\rangle)$ specifies a service that notifies address a with only one of the news selected either from *cnn* or from *bbc*.

— Declaration

$$\mathit{MailBoth}(a,d) \quad \triangleq \quad (\langle x_1 :: x_2\rangle > x > \mathit{email}\langle x,a\rangle) \textbf{ where } x_1 :\in \mathit{cnn}\langle d\rangle \textbf{ where } x_2 :\in \mathit{bbc}\langle d\rangle$$

specifies a service that notifies address a with both news selected from *cnn* and from *bbc* in a unique message.

We refer the interested reader to (Wehrman et al., 2008) for more details and examples.

5.3. *Orc weak barbed bisimilarity*

The abstract semantics of Orc can be defined in terms of strong bisimilarity and gives rise to interesting equivalences, some of which are in Fig. 4.

Since our encoding shall introduce some auxiliary reductions to simulate the evolution of Orc expressions, strong bisimilarity is too strict for our pourposes (there would be bisimilar Orc

$$\begin{array}{ll}
e_1 \,|\, (e_2 \,|\, e_3) \sim (e_1 \,|\, e_2) \,|\, e_3 & \\
e_1 > x > (\, e_2 > y > e_3 \,) \sim (e_1 > x > e_2) > y > e_3 & \text{if } x \notin \mathrm{fv}(e_3) \\
e_1 \,|\, e_2 \sim e_2 \,|\, e_1 & \\
\mathbf{0} > x > e \sim \mathbf{0} & \\
(\, e_1 \,|\, e_2 \,) > x > e \sim (\, e_1 > x > e \,) \,|\, (\, e_2 > x > e \,) & \\
e \,|\, \mathbf{0} \sim e & \\
\mathbf{0} \ \mathbf{where}\ x :\in \mathbf{0} \sim \mathbf{0} & \\
(\, e_2 \,|\, e_3 \,) \ \mathbf{where}\ x :\in e_1 \sim (\, e_2 \ \mathbf{where}\ x :\in e_1 \,) \,|\, e_3 & \text{if } x \notin \mathrm{fv}(e_3) \\
(\, e_3 \ \mathbf{where}\ y :\in e_2 \,) \ \mathbf{where}\ x :\in e_1 \sim (\, e_3 \ \mathbf{where}\ x :\in e_1 \,) \ \mathbf{where}\ y :\in e_2 & \text{if } x \notin \mathrm{fv}(e_2) \text{ and } y \notin \mathrm{fv}(e_1) \\
(\, e_2 > y > e_3 \,) \ \mathbf{where}\ x :\in e_1 \sim (\, e_2 \ \mathbf{where}\ x :\in e_1 \,) > y > e_3 & \text{if } x \notin \mathrm{fv}(e_3)
\end{array}$$

Fig. 4. Some strongly bisimilar Orc expressions

expressions whose encoding processes are not bisimilar). However, we can prove the adequacy of the translation with respect to weak barbed bisimilarity. To this purpose, it is necessary to introduce weak barbed bisimilarity for Orc. The definition follows the standard pattern. We let $C[\,\cdot\,]$ range over *static* Orc contexts, that is, contexts given by the grammar below

$$C[\,\cdot\,] ::= [\,\cdot\,] \;\big|\; [\,\cdot\,]|e \;\big|\; e|[\,\cdot\,] \;\big|\; [\,\cdot\,] > x > e \;\big|\; [\,\cdot\,] \ \mathbf{where}\ x :\in e \;\big|\; e \ \mathbf{where}\ x :\in [\,\cdot\,]\,.$$

Let us write $e \downarrow v$ if there exists e' such that $e \xrightarrow{\langle v \rangle} e'$. Similarly, we write $e \Downarrow v$ if there exists e' such that $e(\xrightarrow{\tau})^* \xrightarrow{\langle v \rangle} e'$. We let weak barbed bisimilarity over Orc, denoted $\dot{\approx}^{\mathsf{Orc}}$, be the largest binary relation over closed Orc expressions such that whenever $e \dot{\approx}^{\mathsf{Orc}} f$ then:

1 whenever $e \downarrow v$ then $f \Downarrow v$, and
2 whenever $e \xrightarrow{\tau} e'$ then there is f' such that $f(\xrightarrow{\tau})^* f'$ and $e' \dot{\approx}^{\mathsf{Orc}} f'$

and vice-versa for f, e. We let barbed equivalence over Orc, denoted $\approx^{\mathsf{Orc}}$ the closure of $\dot{\approx}^{\mathsf{Orc}}$ under static contexts, that is: $e \approx^{\mathsf{Orc}} f$ if and only if for each static $C[\,\cdot\,]$, $C[e] \dot{\approx}^{\mathsf{Orc}} C[f]$.

5.4. *From Orc to CaSPiS*

We start by making a few assumptions on the set of Orc values, $\mathcal{V}$. Let us identify the service names $s, s', \ldots$ of CaSPiS with the site names of Orc. We assume a finite signature Σ, disjoint from the set of variables $x, y, \ldots$, site names $s, s', \ldots$, session names $r, r', \ldots$ and expression names $E, E', \ldots$. We will further assume that:

1 $\mathcal{V} = T_\Sigma$, where as usual T_Σ denotes the set of (ground) terms that can be formed using symbols in Σ;
2 for any site name s, $resp(s, \cdot)$ is a function $\mathcal{V} \to \mathcal{P}_{\mathrm{fin}}(\mathcal{V})$ with finite range;
3 for any s, let $\asymp_s$ be the equivalence relation on $\mathcal{V}$ induced by $resp(s, \cdot)$ (that is, $v \asymp_s v'$ iff $resp(s, v) = resp(s, v')$); then $\asymp_s$ has finite index, and each equivalence class $C \in \mathcal{V}/\asymp_s$ can be represented by a CaSPiS pattern $F_{s,C}$, in the sense that $v \in C$ iff $\mathrm{match}(v, F_{s,C}) = \sigma$ for some σ; for any value v, the names occurring in $F_{s,[v]}$, denoted by $\mathrm{n}(F_{s,[v]})$, are the constants occurring in v.

$$\begin{array}{rcl}
[\![E(x) \triangleq e]\!] & \triangleq & !E.(?x)[\![e]\!] \\
[\![\mathbf{0}]\!] & \triangleq & \mathbf{0} \\
[\![\langle p\rangle]\!] & \triangleq & [\![p]\!]_{\mathrm{v}} \\
[\![?^s_v]\!] & \triangleq & \left(\sum_{w\in resp(s,v)} \langle w\rangle\right) > (?x_w)\langle x_w\rangle \\
[\![E(p)]\!] & \triangleq & \overline{E}.\langle p\rangle!(?x_r)\langle x_r\rangle^{\uparrow} \\
[\![s(p)]\!] & \triangleq & [\![p]\!]_{\mathrm{v}} > \sum_{[v]\in\mathcal{V}/\asymp_s} (F_{s,[v]})[\![?^s_v]\!] \\
[\![x(p)]\!] & \triangleq & [\![x]\!]_{\mathrm{v}} > \sum_{s\in\mathcal{S}} (s)[\![s(p)]\!] \\
[\![e_1 \,|\, e_2]\!] & \triangleq & [\![e_1]\!] \,|\, [\![e_2]\!] \\
[\![e_1 > x > e_2]\!] & \triangleq & [\![e_1]\!] > (?x_v)(\nu\, x)([\![e_2]\!] \,|\, !x.\langle x_v\rangle) \\
[\![e_2 \ \mathbf{where}\ x :\in e_1]\!] & \triangleq & (\nu\, r, x)(\ r \rhd [\![e_1]\!] \ \mid\ r \rhd (?y)!x.\langle y\rangle \ \mid\ [\![e_2]\!]\ )
\end{array}$$

Fig. 5. Encoding Orc into CaSPiS

Without loss of generality, we show the encoding for basic expressions that carry one single parameter, instead of a list of parameters. The more general case can be dealt with by including a list constructor in the signature Σ to be exploited in the inductive encoding of lists of parameters, but this would make the notation heavier without introducing any technical challenge.

The encoding of Orc expressions is detailed in Fig. 5. Some points are worth a comment. An Orc expression may depend on a set of expression definitions, $(E_i(x) \triangleq e_i)_{i=1,\ldots,n}$; hence the encoding of an Orc expression will comprise the encoding of such expression definitions as processes composed in parallel, that is the process

$$Exp \triangleq \Pi_{i=1}^{n}[\![E_i(x) \triangleq e_i]\!]\ .$$

We let $\tilde{E}$ denote the tuple of expression names $E_1, ..., E_n$.

While the call of a site is strict (and thus the actual parameters must have been evaluated), the evaluation of defined expressions is non-strict (and thus parameters can be passed by name). Correspondingly, we define the call by value by letting:

$$[\![s]\!]_{\mathrm{v}} \triangleq \langle s\rangle \qquad [\![v]\!]_{\mathrm{v}} \triangleq \langle v\rangle \qquad [\![x]\!]_{\mathrm{v}} \triangleq \overline{x}.(?x_r)\langle x_r\rangle^{\uparrow}$$

Note that the evaluation of a variable x is encoded as a request for the current value to a service called the *variable manager* of x. Variable managers are introduced by the encoding of sequential composition and asymmetric parallel composition.

Note that the process $[\![?^s_v]\!]$ will evolve non-deterministically in one-step to $\langle w\rangle \,|\, (\mathbf{0} > (?x_w)\langle x_w\rangle)$ for some $w \in resp(s, v)$ (if any), where the process $\mathbf{0} > (?x_w)\langle x_w\rangle$ is clearly inactive.

The most interesting part of the encoding regards the asymmetric parallel composition. The session side $r \rhd [\![e_1]\!]$ is expected to produce one value $\langle v\rangle$ that is passed to the other side of the session, $r \rhd (?y)!x.\langle y\rangle$, after which any other value issued by $r \rhd [\![e_1]\!]$ will become useless (because it cannot be consumed by the opposite side of the session). On the other hand, the first produced value v will be constantly available to e_2 through invocation to the variable manager service $!x.\langle v\rangle$ (now running inside session r).

The correctness of the translation relies on an operational correspondence, established by lemmas 5.4 and 5.5 below, between Orc expressions and their translations.

Let us recall that a process P is return free (resp. input free) if any occurrence of $\langle V\rangle^{\uparrow}$ (resp. (F)) in P is within a service definition or session side (resp. within a service definition or session side or at the top of the right-hand side of a pipeline). The following result ensures that encoded processes are emitters (i.e., both return- and input-free).

Lemma 5.1. For any Orc expression e, the CaSPiS process $[\![e]\!]$ is an emitter.

Proof. Straightforward structural induction on the definition of $[\![e]\!]$. □

The second technical result is a familiar substitution lemma.

Lemma 5.2. For any e, $(\nu\,\tilde{E})(\,(\nu\,x)([\![e]\!]|!x.\langle v\rangle)\,|\,Exp) \gtrsim (\nu\,\tilde{E})(\,[\![e\{^{v}\!/\!_{x}\}]\!]\,|\,Exp)$.

Proof. The proof proceeds by structural induction on the expression e. We show only two main cases, the remaining ones follow analogously.

The most interesting case to consider is when $e = \langle x\rangle$, i.e. we need to show that

$$(\nu\,\tilde{E})(\,(\nu\,x)(\overline{x}.(?x_r)\langle x_r\rangle^{\uparrow}\,|\,!x.\langle v\rangle)\,|\,Exp) \gtrsim (\nu\,\tilde{E})(\,\langle v\rangle\,|\,Exp)$$

which is obvious because $(\nu\,x)(\overline{x}.(?x_r)\langle x_r\rangle^{\uparrow}\,|\,!x.\langle v\rangle) \gtrsim \langle v\rangle$.

The other case we consider is when $e = e_1\,|\,e_2$ for some e_1, e_2. By induction hypothesis we know that $(\nu\,\tilde{E})(\,(\nu\,x)([\![e_i]\!]|!x.\langle v\rangle)\,|\,Exp) \gtrsim (\nu\,\tilde{E})(\,[\![e_i\{^{v}\!/\!_{x}\}]\!]\,|\,Exp)$ for $i \in 1,2$. Therefore, by the congruence property of $\gtrsim$ and by Lemma 4.3(1) we have: $(\nu\,\tilde{E})(\,(\nu\,x)([\![e_1]\!]|[\![e_2]\!]|!x.\langle v\rangle)\,|\,Exp) \gtrsim (\nu\,\tilde{E})(\,[\![e_1\{^{v}\!/\!_{x}\}]\!]\,|\,[\![e_2\{^{v}\!/\!_{x}\}]\!]\,|\,Exp)$. □

The third lemma will allow to create local instances of Exp when needed in the proof of the operational correspondence between Orc expressions and the corresponding CaSPiS processes.

Lemma 5.3. For any Orc expression e and defined expression name E, the name E is not provided by the CaSPiS process $[\![e]\!]$.

Proof. Straightforward structural induction on the definition of $[\![e]\!]$, given that defined expression names are not admitted as values (the encoding can only introduce service invocations to E, not service definitions that are collected in Exp only). □

Lemma 5.4. Let e, e' be closed Orc expressions.

1 whenever $e \xrightarrow{\tau} e'$ then $(\nu\,\tilde{E})([\![e]\!]|Exp) \xrightarrow{\tau}\gtrsim (\nu\,\tilde{E})([\![e']\!]|Exp)$;

2 whenever $e \xrightarrow{\langle v\rangle} e'$ then $(\nu\,\tilde{E})([\![e]\!]|Exp)(\xrightarrow{\tau})^{*} \xrightarrow{\langle v\rangle}\gtrsim (\nu\,\tilde{E})([\![e']\!]|Exp)$.

Proof. The proof is by induction on the derivation of $e \xrightarrow{o} e'$. Most cases follow from induction hypothesis, plus easy manipulations of the encoded expressions according to the laws established by auxiliary lemmas. Here we deal with the two most interesting cases, the other cases being similar or easier. We distinguish the last rule that is applied.

— (Pipe). Assume $e_1 > x > e_2 \xrightarrow{\tau} (e_1' > x > e_2)|e_2\{v/x\}$, where $e_1 \xrightarrow{\langle v\rangle} e_1'$. By induction hypothesis,

$$(\nu\tilde{E})([\![e_1]\!]|Exp)(\xrightarrow{\tau})^* \xrightarrow{\langle v\rangle}\gtrsim (\nu\tilde{E})([\![e_1']\!]|Exp)\,.$$

Let $Q \triangleq (\nu\tilde{E})([\![e_1 > x > e_2]\!]|Exp)$, then we have

$$\begin{array}{rll}
Q & \sim & (\nu\tilde{E})([\![e_1]\!]|Exp) > (\nu\tilde{E})((?x_v)(\nu x)([\![e_2]\!]|!x.\langle x_v\rangle)|Exp) \quad (a)\\
(\xrightarrow{\tau})^* & \gtrsim & (\nu\tilde{E})([\![e_1']\!]|Exp) > (\nu\tilde{E})((?x_v)(\nu x)([\![e_2]\!]|!x.\langle x_v\rangle|Exp))\;|\\
 & & (\nu\tilde{E})((\nu x)([\![e_2]\!]|!x.\langle v\rangle)|Exp) \quad (b)\\
 & \gtrsim & (\nu\tilde{E})([\![e_1']\!]|Exp) > (\nu\tilde{E})((?x_v)(\nu x)([\![e_2]\!]|!x.\langle x_v\rangle|Exp))\;|\\
 & & (\nu\tilde{E})([\![e_2\{v/x\}]\!]|Exp) \quad (c)\\
 & \sim & [\![e_1' > x > e_2\,|\,e_2'\{v/x\}]\!] \quad (d)
\end{array}$$

where the steps of the derivation are justified as follows: (a) by Lemma 4.3(2); (b) by induction hypothesis and congruence properties of $\gtrsim$; (c) by Lemma 5.2 and congruence properties of $\gtrsim$; (d) by Lemma 4.3(1), Lemma 4.3(2) and the definition of the encoding.

— (Pick). Assume e_2 **where** $x :\in e_1 \xrightarrow{\tau} e_2\{v/x\}$, where $e_1 \xrightarrow{\langle v\rangle} e_1'$. By induction hypothesis,

$$(\nu\tilde{E})([\![e_1]\!]|Exp)(\xrightarrow{\tau})^* \xrightarrow{\langle v\rangle}\gtrsim (\nu\tilde{E})([\![e_1']\!]|Exp)\,.$$

Let $Q \triangleq (\nu\tilde{E})([\![e_2 \textbf{ where } x :\in e_1]\!]|Exp)$, then we have

$$\begin{array}{rll}
Q & \sim & (\nu x, r)(\,(\nu\tilde{E})(r \rhd ([\![e_1]\!]|Exp))\,|\,r \rhd (?y)!x.\langle y\rangle\,|\,(\nu\tilde{E})([\![e_2]\!]|Exp)\,) \quad (a)\\
(\xrightarrow{\tau})^* & \gtrsim & (\nu x, r)(\,(\nu\tilde{E})(r \rhd ([\![e_1']\!]|Exp))\,|\,r\rhd !x.\langle v\rangle\,|\,(\nu\tilde{E})([\![e_2]\!]|Exp)\,) \quad (b)\\
 & \gtrsim & (\nu\tilde{E})(\,(\nu x)(!x.\langle v\rangle|[\![e_2]\!])|Exp) \quad (c)\\
 & \gtrsim & (\nu\tilde{E})([\![e_2\{v/x\}]\!]|Exp) \quad (d)
\end{array}$$

where the steps of the derivation are justified as follows: (a) by Lemma 4.3(3); (b) by induction hypothesis, congruence properties of $\gtrsim$ and an elementary properties of the operational semantics; (c) by Lemma 4.4, scope extrusion and congruence properties of $\gtrsim$; (d) by Lemma 5.2.

□

Lemma 5.5. Let e be a closed Orc expression.

1 If $(\nu\tilde{E})([\![e]\!]|Exp) \xrightarrow{\tau} Q$ then there is e' such that $e(\xrightarrow{\tau})^*e'$ and $Q \gtrsim (\nu\tilde{E})([\![e']\!]|Exp)$;

2 If $(\nu\tilde{E})([\![e]\!]|Exp)(\xrightarrow{\tau})^* \xrightarrow{\langle v\rangle} Q$ then there is e' such that $e \xrightarrow{\langle v\rangle} e'$ and $Q \gtrsim (\nu\tilde{E})([\![e']\!]|Exp)$.

Proof. The proof is by induction on the expression e. We only deal with the most interesting case, which arises when $e = (\,e_1$ **where** $x :\in e_2\,)$ and there is a τ-move originating from an interaction between $r \rhd [\![e_1]\!]$ and $r \rhd (?y)!x.\langle y\rangle$ (the other cases are similar or easier). That is, we consider the case

$$(\nu\tilde{E}, r, x)(r \rhd [\![e_1]\!]|r \rhd (?y)!x.\langle y\rangle|[\![e_2]\!]|Exp) \xrightarrow{\tau} (\nu\tilde{E}, r, x)(r \rhd Q'|r\rhd !x.\langle v\rangle|[\![e_2]\!]|Exp) \triangleq Q$$

where $[\![e_1]\!] \xrightarrow{\langle v\rangle} Q'$, hence $(\nu\tilde{E})([\![e_1]\!]|Exp) \xrightarrow{\langle v\rangle} (\nu\tilde{E})(Q'|Exp)$. By induction hypothesis, there is an expression e_1' s.t.

$$e_1 \xrightarrow{\langle v\rangle} e_1' \text{ and } (\nu\tilde{E})(Q'|Exp) \gtrsim (\nu\tilde{E})([\![e_1']\!]|Exp)\,. \quad (1)$$

Hence we have

$$\begin{array}{rcll} Q & \sim & (\nu\, r)(\,(\nu\,\tilde{E})(r \rhd Q'|Exp)\,|\,(\nu\,\tilde{E},x)(r\rhd !x.\langle v\rangle|[\![e_2]\!]|Exp)\,) & (a)\\ & \gtrsim & (\nu\, r)(\,(\nu\,\tilde{E})(r \rhd [\![e'_1]\!]|Exp)\,|\,(\nu\,\tilde{E},x)(r\rhd !x.\langle v\rangle|[\![e_2]\!]|Exp)\,) & (b)\\ & \sim & (\nu\,\tilde{E})(\nu\,\tilde{x})(\,(\nu\,\tilde{r})(r \rhd [\![e'_1]\!]|r\rhd !x.\langle v\rangle)\,|\,[\![e_2]\!]|Exp)\,) & (c)\\ & \gtrsim & (\nu\,\tilde{E})(\nu\,\tilde{x})(\,!x.\langle v\rangle|[\![e_2]\!]|Exp\,) & (d)\\ & \gtrsim & (\,[\![e_2\{^v\!/\!_x\}]\!]|Exp\,) & (e) \end{array}$$

where the various steps are justified as follows: (a) follows from Lemma 4.3(1), (b) follows from inductive hyothesis (1) for Q', plus congruence properties of $\gtrsim$ under $r \rhd [\,\cdot\,]$, (c) follows again from Lemma 4.3(1) and simple rearrangement of restrictions, (d) follows from Lemma 4.4 and finally (e) follows from Lemma 5.2. □

Theorem 5.1 (correctness of translation). Let e, f be closed Orc expressions. Then $e \approx^{\mathsf{Orc}} f$ if and only if for each Orc context $C[\cdot]$, $(\nu\,\tilde{E})([\![C[e]]\!]|Exp) \mathrel{\dot{\approx}} (\nu\,\tilde{E})([\![C[f]]\!]|Exp)$.

Proof. Lemma 5.4 and Lemma 5.5 entail that the translation is fully abstract w.r.t. barbed bisimilarity: $e \mathrel{\dot{\approx}}^{\mathsf{Orc}} f$ iff $(\nu\,\tilde{E})([\![e]\!]|Exp) \mathrel{\dot{\approx}} (\nu\,\tilde{E})([\![f]\!]|Exp)$. The wanted statement follows by definition of barbed equivalence $\approx^{\mathsf{Orc}}$ over Orc expressions. □

The fact that translated Orc terms are emitters suggests that the relation $\mathrel{\dot{\approx}}$ in the above theorem can in fact be promoted to $\approx$; we leave this conjecture for future investigation.

6. Handling session closure: the full CaSPiS

The calculus we have presented in the previous sections offers no primitives for handling session closure. 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 (van der Aalst et al., 2003), or to manage abnormal events, or timeouts.

In this section we present the full syntax and semantics of CaSPiS and show how sessions can be properly handled to guarantee that no session side will be left dangling when its opposite side is closed.

Sessions are units of client-server cooperation and as such their termination must be managed 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., closed in turn. The mechanism of session termination we shall adopt has been informally described in the Introduction. Before presenting this extension formally, we recall that an important aspect of our modelling is that, when shutting down a side, the emitted signal $\dagger(k)$ is (realistically) *asynchronous*. 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. In Section 6.1, when presenting the operational semantics, we will also report some alternative mechanisms that we have considered but discarded when designing CaSPiS primitives. These aspects must be taken into account when defining what "well-programmed" closing means (see Section 6.2).

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

Fig. 6. Syntax of full CaSPiS.

6.1. *Full CaSPiS*

Syntax In what follows, we assume a new countable set $\mathcal{K}$ of *signal names*, ranged by k, disjoint from session and service names. The syntax of full CaSPiS is reported in Fig. 6. The difference w.r.t. Fig. 1 is given by the extended primitives $s_k.P$, $\overline{s}_k.P$ and $r \rhd_k P$ and the new primitives close, $\dagger(k)$, $\blacktriangleright P$ and $k \cdot P$. We consider $r \rhd_k P$ and $\blacktriangleright P$ as run-time syntax. As a matter of notation, when the handler k in $s_k.P$ is vacuous or inessential then we can safely omit it, and the same for $\overline{s}_k.P$ and $r \rhd_k P$; thus, all the processes we have presented using the syntax of the close-free fragment can be thought just as carrying vacuous handlers and remain valid w.r.t. the syntax in Fig. 6.

In what follows, we say that a name *n occurs free in P underneath a context* $\mathbb{C}[\cdot]$ if there is a term Q such that $n \in \mathrm{fn}(Q)$ and $P = \mathbb{C}[Q]$. 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.

While the full technical details are postponed to Section 6.2, here we just mention the main constraints over CaSPiS processes that can guarantee the so-called *graceful termination property*. Informally, the key concept is that of a *balanced* term, roughly, a term with only pairs of session-sides that balance with each other. Termination of one side may lead to unbalanced terms. The graceful termination property guarantees that *any possibly unbalanced term reachable from a balanced term can get balanced in a finite number of reductions*.

Structural congruence. The structural rules listed in Fig. 7 enrich the set of rules already introduced in Fig. 2. The law $\blacktriangleright \dagger(k) \equiv \dagger(k)$ is motivated by the possible presence of subtle race conditions on the order of session closing due to the nesting of sessions (an example is shown later). 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.

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

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

From now on the structural congruence $\equiv$ is defined as the least congruence that includes alpha-equivalence together with all laws in Fig. 7 and Fig. 2. Note that, as usual, structural congruence can be exploited to move to the top level all restrictions that are not in the scope of a dynamic operator.

Reduction semantics The reductions must be updated to take into account termination handlers. The only significant change regards the handshake between a service definition and a service invocation, where termination handlers must be annotated in the freshly created session sides.

$$(\textsc{sync})\frac{r \text{ fresh for } \mathbb{C}[\,\cdot,\cdot\,], P, Q}{\mathbb{C}[\, s_{k_1}.P, \bar{s}_{k_2}.Q \,] \xrightarrow{\tau} (\nu\, r)\mathbb{C}[\, r \rhd_{k_2} P, r \rhd_{k_1} Q \,]}$$

Rule (PSync) is left unchanged, while we need to annotate the sessions appearing in rules (SSync), (SRSync) and (PRSync) with suitable termination handlers k and k_1. For the sake of clarity, we show below how the annotated rule (SRSync) looks like (for $\mathbb{C}_{r,k,k'}[\,\cdot\,]$ be a static context of the form $\mathbb{C}[\, r \rhd_k \mathbb{P}[\,\cdot\,], r \rhd_{k'} \mathbb{S}[\,\cdot\,]\,]$).

$$(\textsc{SRsync})\frac{\sigma = \text{match}(F, V)}{\mathbb{C}_{r,k,k'}[\, r_1 \rhd_{k_1} \mathbb{S}_1[\, \langle V\rangle^{\uparrow} P + \sum_i \pi_i P_i \,], (F)Q + \sum_j \pi_j Q_j \,] \xrightarrow{\tau} \mathbb{C}_{r,k,k'}[\, r_1 \rhd_{k_1} \mathbb{S}_1[\, P \,], Q\sigma \,]}$$

Three new rules are needed to handle session cancellation. Two of them regards the generation of notifications to be delivered on the opposite side, which may be due to the execution of the close primitive (Send) or to the termination of an enclosing session (Tend):

$$(\textsc{Send})\frac{}{\mathbb{C}[\, r \rhd_k \mathbb{S}[\,\mathsf{close}\,]\,] \xrightarrow{\tau} \mathbb{C}[\,\dagger(k) \mid \blacktriangleright \mathbb{S}[\,\mathbf{0}\,]\,]} \qquad (\textsc{Tend})\frac{}{\mathbb{C}[\,\blacktriangleright (r \rhd_k P)\,] \xrightarrow{\tau} \mathbb{C}[\,\dagger(k) \mid \blacktriangleright P\,]}$$

The last rule models the handshake between a notification signal and its handler:

$$(\textsc{Tsync})\frac{}{\mathbb{C}[\,\dagger(k) \,|\, k \cdot P\,] \xrightarrow{\tau} \mathbb{C}[\,P\,]}$$

Even for well-defined processes, the session closing primitives do not guarantee *per se* that dangling one-sided sessions arise. However, many situations can be handled satisfactorily just by installing suitable termination handlers of the form $k \cdot \mathbb{C}[\,\mathsf{close}\,]$ in the bodies of client invocations and service definitions. Rather liberal choices of $\mathbb{C}[\,\cdot\,]$are also allowed, that may contain extra actions to be performed when termination handler is activated, e.g., to send signals to other listeners (a sort of compensation, in the language of long-running transactions).

Alternative set of primitives for cancellation. 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* sides of the same session. But this strategy violates the principle that each party is in charge for the closing of its own session side. A second alternative would be to use close as a synchronisation primitive, so that the client-side and service-side sessions are terminated when close is encountered on one side and $\overline{\mathsf{close}}$ 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 as a reasonable solution: each party can exit a session autonomously but it is obliged to inform the other party.

Examples. 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:

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

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 $\dagger(k_1) \mid \blacktriangleright (P \mid \dagger(k_2) \mid \blacktriangleright Q)$, 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 $\dagger(k)$.

As a more elaborated example, consider the process *News* defined as follows:

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

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`. To be more concrete, we can assume `ANSA`, `BBC` and `CNN` be services that, upon invocation, return a possibly infinite sequence of values representing pieces of news (disregarding the identity of these news, these services might resemble $!ANSA.!(\nu n)\langle n\rangle$).

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 top-most session will then cause the termination of all (not yet terminated) nested news clients.

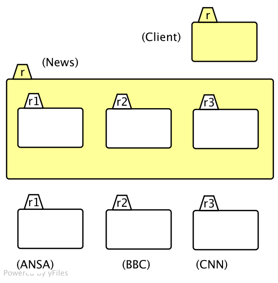
For example, after invoking `collect`, the client below receives all the news produced by `ANSA`, `BBC` and `CNN` (in some interleaved order):

$$HeavyReader \ \triangleq\ (\nu k')\overline{\mathtt{collect}}_{k'}.(!(?y)\langle y\rangle^{\uparrow} \mid k'\cdot\mathsf{close})$$

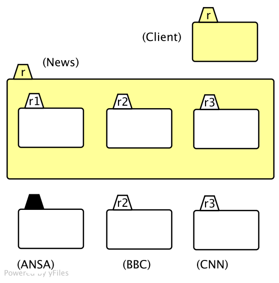
Instead the client below receives only the first news and then abandons the session:

$$EasyReader \ \triangleq\ (\nu k')\overline{\mathtt{collect}}_{k'}.((?y)\langle y\rangle^{\uparrow}\mathsf{close} \mid k'\cdot\mathsf{close})$$

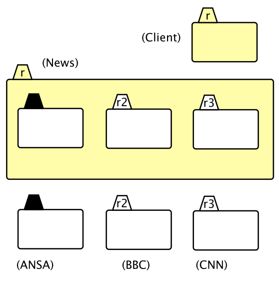
Figure 8 shows a possible propagation of termination. Initially all services have been invoked



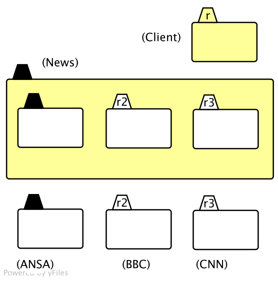
(a) All sides are active.



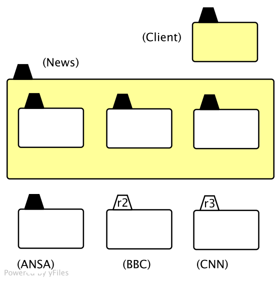
(b) `ANSA`-side terminates.



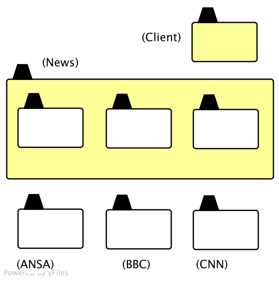
(c) `ANSA`-client terminates.



(d) `News`-side terminates.



(e) Client-side terminates.



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

Fig. 8. Propagation of side-termination via listeners.

and the corresponding sessions have been triggered (Figure 8(a)). Suppose the session running on `ANSA`-side shuts down (Figure 8(b)), then its partner session-side running as a child of the main session of the news collector is informed and terminated (Figure 8(c)). Moreover, the listener causes the termination of the parent session (Figure 8(d)). Termination of the main session causes the termination of the remaining two children sections and of the main client-side session (Figure 8(e)). Finally, the sessions running on `BBC`-side and `CNN`-side are also terminated (Figure 8(f)).

6.2. *Programming graceful termination*

Some 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. In fact their combination would lead to sessions of the form $(\nu\, r, k_1, k_2)(\; r \rhd_{k_2} (P_1|k_1 \cdot \mathsf{close}\,) \;|\; r \rhd_{k_1} (P_2|k_2 \cdot \mathsf{close}\,)\;)$.

In this section, we show much more general constraints to make sure that termination handlers of the form $k \cdot \mathbb{C}[\,\mathsf{close}\,]$, for suitable static contexts $\mathbb{C}[\,\cdot\,]$ and signals k, are installed in the bodies of client invocations and service definitions. We will be rather liberal with the choice of $\mathbb{C}[\,\cdot\,]$, 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 previously presented process *News* 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.

Well-formedness is not sufficient to guarantee graceful session closing. We have to restrict CaSPiS syntax appropriately. The first step is to ensure that session termination is properly programmed inside all service definitions and invocations.

Definition 6.1 (graceful property). We say P is *graceful* if there is $Q \equiv P$ such that 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 $s_k.(\mathbb{S}[\, k \cdot \mathbb{S}'[\, \mathsf{close}\ ]\,])$ or $\overline{s}_k.(\mathbb{S}[\, k \cdot \mathbb{S}'[\, \mathsf{close}\ ]\,])$, with $\mathbb{S}[\,\cdot\,], \mathbb{S}'[\,\cdot\,]$ static and session immune; (b) in Q there is at most one occurrence of the listener $k\cdot$ and one occurrence of $\rhd_k$.

Lemma 6.1. Let P be graceful and suppose $P \xrightarrow{\tau} Q$. Then Q is graceful.

The proof of Lemma 6.1 involves some case analysis based on the characterisation of active and passive components of any reduction $P \xrightarrow{\tau} Q$. The proof of Lemma 6.1 is then an easy inspection of the cases 1–8 that are exposed by Lemma 6.2 below to summarise the possible shapes of the source and target well-formed processes of a reduction. (Note that, by definition, the graceful property is preserved by structural congruence.)

Lemma 6.2 (context lemma for reductions). Suppose P is well-formed and $P \xrightarrow{\tau} Q$. Then there is a 1- or 2-hole static context, $\mathbb{C}[\,\cdot\,]$ or $\mathbb{C}[\,\cdot,\cdot\,]$, such that one of the following cases is true (for some static and session-immune contexts $\mathbb{S}_0[\,\cdot\,], \mathbb{S}_1[\,\cdot\,], \mathbb{S}_2[\,\cdot\,]$, some names r, k, k', k'', s, some prefixes V_i's, F_j's, and some processes P_i's, R_j's, R, P', σ, such that match$(V_l, F_h) = \sigma$).

1 (Sync)
$$\begin{array}{lcl} P & \equiv & \mathbb{C}[\, s_k.P',\ \overline{s}_{k'}.R\,] \\ Q & \equiv & (\nu\, r)\mathbb{C}[\, r \rhd_{k'} P',\ r \rhd_k R\,] \quad r \text{ fresh for } P', \mathbb{C}[\,\cdot\,], R \end{array}$$

2 (Ssync)
$$\begin{array}{lcl} P & \equiv & \mathbb{C}[\, r \rhd_k (P' | \sum_i \langle V_i \rangle P_i),\ r \rhd_{k'} \mathbb{S}_0[\, \sum_j (F_j) R_j\,]\,] \\ Q & \equiv & \mathbb{C}[\, r \rhd_k (P' | P_l),\ r \rhd_{k'} \mathbb{S}_0[\, R_h \sigma\,]\,] \end{array}$$

3 (SRsync)
$$\begin{array}{lcl} P & \equiv & \mathbb{C}[\, r' \rhd_k (r \rhd_{k''} \mathbb{S}_1[\, \sum_i \langle V_i \rangle^{\uparrow} P_i\,] | P'),\ r \rhd_{k'} \mathbb{S}_2[\, \sum_j (F_j) R_j\,]\,] \\ Q & \equiv & \mathbb{C}[\, r' \rhd_k (r \rhd_{k''} \mathbb{S}_1[\, P_l\,] | P'),\ r \rhd_{k'} \mathbb{S}_2[\, R_h \sigma\,]\,] \end{array}$$

4 (Psync)
$$\begin{array}{lcl} P & \equiv & \mathbb{C}[\, (\sum_i \langle V_i \rangle P_i | P') > \mathbb{S}_1[\, \sum_j (F_j) R_j\,]\,] \\ Q & \equiv & \mathbb{C}[\, R_h \sigma \,|\, ((P_l | P') > \mathbb{S}_1[\, \sum_j (F_j) R_j\,])\,] \end{array}$$

5 (PRsync)
$$\begin{array}{lcl} P & \equiv & \mathbb{C}[\, ((r \rhd_k \mathbb{S}_1[\, \sum_i \langle V_i \rangle^{\uparrow} P_i\,]) \,|\, P') > \mathbb{S}_2[\, \sum_j (F_j) R_j\,]\,] \\ Q & \equiv & \mathbb{C}[\, R_h \sigma \,|\, ((r \rhd_k \mathbb{S}_1[\, P_l\,] | P') > \mathbb{S}_2[\, \sum_j (F_j) R_j\,])\,] \end{array}$$

6 (Send) $P \equiv \mathbb{C}[\, r \rhd_k \mathbb{S}_0[\, \mathsf{close}\ ]\,]$ and $Q \equiv \mathbb{C}[\, \blacktriangleright \mathbb{S}_0[\, \mathbf{0}\,] | \dagger(k)\,]$
7 (Tend) $P \equiv \mathbb{C}[\, \blacktriangleright (r \rhd_k R)\,]$ and $Q \equiv \mathbb{C}[\, \blacktriangleright R | \dagger(k)\,]$
8 (Tsync) $P \equiv \mathbb{C}[\, \dagger(k) | k \cdot R\,]$ and $Q \equiv \mathbb{C}[\, R\,]$

Proof. The proof follows by noting that all reduction rules for CaSPiS are axioms (all their derivations have length one), that guarded summation carry uniform prefixes in well-formed processes and that in static, session- and pipeline-immune contexts $\mathbb{P}[\,\cdot\,]$ appearing in the rules the hole can appear only underneath parallel composition and restrictions (see Lemma 2.1). $\square$

The graceful property is not sufficient to guarantee the main result 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 6.2 (*r*-balancing). Let P be a process. We say P is

— *r-balanced* if either $r \notin \mathrm{fn}(P)$ or, for some static $\mathbb{C}[\,\cdot\,]$ not mentioning r, and some k and k' (with $k \neq k'$), $P \equiv \mathbb{C}[\, r \rhd_k A, r \rhd_{k'} B\,]$ where one of the following holds for some session-immune $\mathbb{S}'[\,\cdot\,]$ and $\mathbb{S}''[\,\cdot\,]$

(a) $A \equiv \mathbb{S}'[\,\mathsf{close}\,]$

(b) $A \equiv \mathbb{S}'[\,\dagger(k')|k' \cdot \mathbb{S}''[\,\mathsf{close}\,]\,]$

(c) $A \equiv \mathbb{S}'[\,k' \cdot \mathbb{S}''[\,\mathsf{close}\,]\,]$

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 $\mathbb{C}[\,\cdot\,]$ not mentioning r, $P \equiv \mathbb{C}[\, r \rhd_k A\,]$ with either of *(a)* or *(b)* above holding for A, or $P \equiv \mathbb{C}[\,\blacktriangleright\, r \rhd_k A\,]$.

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

Definition 6.3 ((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.

Remark 6.1. However intuitive the underlying concept, the definition of (quasi-)balanced process involves a non straightforward combination of contexts and structural properties of processes. A type system that would single out balanced terms might seemingly represent a natural alternative to this presentation. This possibility has indeed been pursued by the authors at the early stages of development of the calculus, but then abandoned in favour of the present formulation. In fact, balancing conditions naturally arise as "global" term properties that are awkward to check in a compositional fashion, hence via a type system. We note that a conceptually similar notion of balancing is also found in (Acciai and Boreale, 2008a), which employs type systems to characterize responsiveness in the pi-calculus. The interested reader is referred to this paper for futher discussion on the issue of balancing in connection with type systems.

Theorem 6.1 (graceful termination). Let P be balanced. Whenever $P \xrightarrow{\tau}{}^{*} P'$ there exists a balanced process Q such that $P' \xrightarrow{\tau}{}^{*} Q$.

Proof. 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 (Lemma A.3 in the Appendix).

— Second, we prove that for any quasi-balanced process R there is a balanced process R'' such that $R \xrightarrow{\tau}^{*} R''$ (Lemma A.4 in the Appendix).

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'. $\square$

We can now check that session handling for process *News* defined in Section 6 is entirely satisfactory. In fact it is immediate to verify that it 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.

Remark 6.2. Graceful termination guarantees that whenever a session side is dangling then it can be closed, but in general it does not prevent dangling signals $\dagger(k)$ addressed to closed session sides. However, simple garbage collecting rules can be set up for removing dangling signals and disposing inactive processes. For example, the following rules can be safely applied, because the abstract semantics considered for CaSPiS is insensitive to them:

$$\frac{P \in \left\{ \sum_i \pi_i P_i,\ u_k.Q,\ \overline{u}_k.Q,\ \mathsf{close},\ k \cdot Q \right\}}{\mathbb{C}[\, \blacktriangleright P \,] \xrightarrow{\tau} \mathbb{C}[\, \mathbf{0} \,]} \qquad \frac{S \equiv \dagger(k) \,|\, \ldots \,|\, \dagger(k)}{\mathbb{C}[\, (\nu k) S \,] \xrightarrow{\tau} \mathbb{C}[\, \mathbf{0} \,]} \qquad \frac{}{\mathbb{C}[\, \mathbf{0} > P \,] \xrightarrow{\tau} \mathbb{C}[\, \mathbf{0} \,]}$$

The first rule cancels a closed session side when its process cannot perform any interaction. The second rule deletes all $\dagger(k)$ signals with no listener. The third rule cancels a pipeline whose left element is the null process.

6.3. *Encoding Orc, gracefully*

Using graceful processes we can improve the encoding of the asymmetric parallel operator e_2 **where** $x :\in e_1$ of Orc seen in Section 5.4, by closing the branch e_1 after the first value has been issued to e_2 via x. (The encoding in Fig. 5 just guarantees that the useless branch will no longer interfere with the computation, but it may still execute some internal steps).

The main idea is to modify the encoding shown in Fig. 5 to guarantee the graceful termination property. In particular, we need to change the way in which the manager of x is installed in the encoding of asymmetric parallel composition. Then, remind that $[\![e_1]\!]$ could have opened many other session sides before cancellation occurs and hence find suitable policies for invoking sites, expression definitions and local services.

The "graceful" encoding is shown in Fig. 9, where we exploit the following macros to improve readability

$$\mathbf{serv}(s, P) \triangleq (\nu k) s_k.(P \,|\, k \cdot \mathsf{close}\,) \qquad \mathbf{get}(s) \triangleq (\nu k)\overline{s}_k.((?x_r)\langle x_r \rangle^{\uparrow}\mathsf{close} \,|\, k \cdot \mathsf{close}\,)$$

and let:

$$
\begin{array}{rcl}
[\![E(x) \triangleq e]\!] & \triangleq & !\mathbf{serv}(E, (?x)[\![e]\!]) \\
[\![\mathbf{0}]\!] & \triangleq & \mathbf{0} \\
[\![\langle p \rangle]\!] & \triangleq & [\![p]\!]_{\mathrm{v}} \\
[\![?^s_v]\!] & \triangleq & \left(\sum\limits_{w \in resp(s,v)} \langle w \rangle \right) > (?x_w)\langle x_w \rangle \\
[\![E(p)]\!] & \triangleq & (\nu\, k)\overline{E}_k.(\langle p \rangle!(?x_r)\langle x_r \rangle^{\uparrow} \,|\, k \cdot \mathsf{close}\,) \\
[\![s(p)]\!] & \triangleq & [\![p]\!]_{\mathrm{v}} > \sum\limits_{[v] \in \mathcal{V}/\asymp_s} F_{s,[v]}\, [\![?^s_v]\!] \\
[\![x(p)]\!] & \triangleq & [\![x]\!]_{\mathrm{v}} > (?s)[\![s(p)]\!] \\
[\![e_1 \,|\, e_2]\!] & \triangleq & [\![e_1]\!] \,|\, [\![e_2]\!] \\
[\![e_1 > x > e_2]\!] & \triangleq & [\![e_1]\!] > (?x_v)(\nu\, x)([\![e_2]\!] \,|\, !\mathbf{serv}(x, \langle x_v \rangle)) \\
[\![e_2 \ \mathbf{where}\ x :\in e_1]\!] & \triangleq & (\nu\, wh, x)(\ \mathbf{serv}(wh, [\![e_1]\!]) \ |\ (\mathbf{get}(wh) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \ |\ [\![e_2]\!]\ )
\end{array}
$$

Fig. 9. Encoding Orc into CaSPiS, gracefully

$$[\![s]\!]_{\mathrm{v}} \triangleq \langle s \rangle \qquad [\![v]\!]_{\mathrm{v}} \triangleq \langle v \rangle \qquad [\![x]\!]_{\mathrm{v}} \triangleq \mathbf{get}(x)$$

Note that the only differences w.r.t. Fig. 5 are concerned with the encoding of $E(x) \triangleq e$, $E(p)$, $e_1 > x > e_2$ and e_2 **where** $x :\in e_1$.

The macro $\mathbf{serv}(s, P)$ defines a service s with body P together with a proper fresh listener that can be used to close the session when the other side terminates. Note that we always use the macro with a body P that does not contain close, i.e. the service side cannot terminate the session autonomously.

The macro $\mathbf{get}(s)$ defines a client for service s that takes one value, returns it one level up and then closes its session side. In this case the listener is superfluous, because the client side will always close first, but inserted to satisfy the constraints required by the graceful property.

We exemplify below how the graceful encoding of asymmetric parallel works in practice. To improve readability, we omit all restrictions and write P_1 and P_2 in place of $[\![e_1]\!]$ and $[\![e_2]\!]$, respectively. First, by (Sync), a session r is established between the server side of wh (where P_1 runs embedded) and the client side, where the manager of the where is waiting for one value to instantiate x and terminate the session r.

$$
\begin{array}{l}
[\![e_2 \ \mathbf{where}\ x :\in e_1]\!] \\
\xrightarrow{\tau}\ r \rhd_{k'}\ (P_1 \,|\, k \cdot \mathsf{close}\,) \,|\, ((r \rhd_k\ ((?x_r)\langle x_r \rangle^{\uparrow}\mathsf{close} \,|\, k' \cdot \mathsf{close}\,)) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\ P_2
\end{array}
$$

Now suppose $P_1(\xrightarrow{\tau})^* \xrightarrow{\langle 5 \rangle} Q_1$, then by (Ssync), (PRsync), (Send):

$$
\begin{array}{l}
r \rhd_{k'}\ (P_1 \,|\, k \cdot \mathsf{close}\,) \,|\ ((r \rhd_k\ ((?x_r)\langle x_r \rangle^{\uparrow}\mathsf{close} \,|\, k' \cdot \mathsf{close}\,)) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\ P_2 \\
(\xrightarrow{\tau})^*\ r \rhd_{k'}\ (Q_1 \,|\, k \cdot \mathsf{close}\,) \,|\ ((r \rhd_k\ (\langle 5 \rangle^{\uparrow}\mathsf{close} \,|\, k' \cdot \mathsf{close}\,)) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\ P_2 \\
\xrightarrow{\tau}\ r \rhd_{k'}\ (Q_1 \,|\, k \cdot \mathsf{close}\,) \,|\ ((r \rhd_k\ (\mathsf{close} \,|\, k' \cdot \mathsf{close}\,)) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\ !\mathbf{serv}(x, \langle 5 \rangle) \,|\ P_2 \\
\xrightarrow{\tau}\ r \rhd_{k'}\ (Q_1 \,|\, k \cdot \mathsf{close}\,) \,|\ ((\dagger(k) \,|\ \blacktriangleright k' \cdot \mathsf{close}\,) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\ !\mathbf{serv}(x, \langle 5 \rangle) \,|\ P_2\ .
\end{array}
$$

Finally, by structural congruence, (Tsync) and (Send):

$$\begin{array}{l}
r \rhd_{k'} \ (Q_1 \,|\, k \cdot \mathsf{close}\,) \,|\, ((\dagger(k) \,|\, \blacktriangleright k' \cdot \mathsf{close}\,) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\, !\mathbf{serv}(x, \langle 5 \rangle) \,|\, P_2 \\
\equiv \ r \rhd_{k'} \ (Q_1 \,|\, \dagger(k) \,|\, k \cdot \mathsf{close}\,) \,|\, ((\blacktriangleright k' \cdot \mathsf{close}\,) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\, !\mathbf{serv}(x, \langle 5 \rangle) \,|\, P_2 \\
\xrightarrow{\tau} \ r \rhd_{k'} \ (Q_1 \,|\, \mathsf{close}\,) \,|\, ((\blacktriangleright k' \cdot \mathsf{close}\,) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\, !\mathbf{serv}(x, \langle 5 \rangle) \,|\, P_2 \\
\xrightarrow{\tau} \blacktriangleright Q_1 \,|\, \dagger(k') \,|\, ((\blacktriangleright k' \cdot \mathsf{close}\,) > (?x_v)!\mathbf{serv}(x, \langle x_v \rangle)) \,|\, !\mathbf{serv}(x, \langle 5 \rangle) \,|\, P_2 \ .
\end{array}$$

If needed, structural congruence and rule (Tend) will allow in turn to close any session side within $\blacktriangleright Q_1$, hence in the end we get $!\mathbf{serv}(x, \langle 5 \rangle) \,|\, P_2$, up to the presence of inactive processes (remind that, for the sake of readability, we are not displaying top-most restrictions for k, k').

Of course, the above term contains many inactive processes that cannot do any harm. However, they could be safely removed by resorting to the garbage collection rules discussed in Remark 6.2.

Please notice that we needed all notable features of CaSPiS (session-based communication, pipelining, pattern matching) in order to encode Orc. In particular, sessions are necessary to link (the encoding of) Orc variables to their possible sources of published values. Indeed, without them expressiveness would be very limited. More generally, sessions (arising from service invocations) are essential in the encoding of communications. They have been used also in the encoding of π-calculus that we sketched in Section 3, because pipeline-based communication (the only possible alternative) turned out to be too rigid: it is not bidirectional and it is driven by the structure of terms.

7. Related work

CaSPiS has been developed inside the Sensoria project (Sensoria Project, 2010; Wirsing and Hölzl, 2011), as part of a larger research effort aimed to develop *core calculi for SOC* at three different levels of abstraction:

Service middleware level: this level is close to current networking technologies to be directly implementable, but sufficiently expressive to support service oriented applications.

Service description level: this level facilitates more abstract formalisation of basic concepts such as service definition, invocation, instantiation, and communication.

Service composition level: this level provides mechanisms for the modelling and analysis of qualitative and quantitative aspects of multiparty service compositions.

At the middleware level we find, e.g., the *signal calculus* (SC) (Ferrari et al., 2006): it is based on a flexible and dynamically reconfigurable network of components communicating via the publish-subscribe message delivery paradigm. Sessions and message correlations are supported through a type system (Ferrari et al., 2007). This calculus is implemented as a Java library and is equipped with a high-level graphical programming environment.

At the service composition level we find, e.g., λreq (Bartoletti et al., 2007) and *concurrent constraint pi-calculus* (cc-pi) (Buscemi and Montanari, 2007). The former has been exploited to support the development of techniques for the analysis of service compositions (such as static analysis of the access to protected resources) within the so-called "call-by-contract" paradigm, while the latter integrates name handling features with constraint semirings to deal more effectively with quantitative aspects of negotiations (such as the so-called service level agreement).

CaSPiS lies at the service description level, where several other interesting proposals are also present. Indeed, in the service-oriented computing literature, two main approaches are considered to maintain the link between the entities incolved in the service invocation.

Correlations: The link between partners (caller and callee) is determined by correlation values. Those values are explicitly included in the exchanged messages and only messages containing the 'right' correlation values are processed by the partner (OASIS, 2007).

Sessions: A private channel is implicitly instantiated when calling a service: it binds caller and callee and is used for their future communication (Honda et al., 1998).

Correlation information is the basis of the Web Services interaction mechanisms. Suitable tokens, exchanged during a conversation, are matched to identify the specific instance of a business process in a runtime environment where several instances of it are running. Correlation is, thus, a flexible and user programmable mechanism for managing the relationships among collaborating partners. Different techniques have been identified to direct messages to the right partners. In the so called, *stateless correlation* partners are selected via pattern matching and, in case of multiple matchings, input processes receive the message that requires less substitutions. In the so called *stateful correlation*, every process has a corresponding state consisting of valued variables. Given a set of variables, explicitly indicated as those driving the correlation, a message is directed to a process that contains in its correlation variables the same correlation values included in the message. Stateless correlation follows more tightly the tradition of process calculi, adding a sophisticated pattern-matching mechanism on top of a π-calculus like communication. Stateful correlation, instead, is more tightly related to actual service-oriented programming language such as WS-BPEL (OASIS, 2007) and heavily relies on the notion of state. The modeling of these two forms of correlation-based service invocation and conversation generated the two process calculi COWS (Lapadula et al., 2007) and SOCK (Busi et al., 2006).

Sessions are instead a more abstract alternative to correlation variables and permit more abstract reasoning about process behaviours. Sessions are established by creating private names that are carefully managed by interacting partners in correspondence of services invocations.. The group of formalism with explicit session names comprises the so-called SCC-family of calculi (Boreale et al., 2006), spawned by a first proposal of a basic calculus with nested session, the *Service Centred Calculus* (SCC), later enriched and refined with different mechanisms for inter-session communication, like *data streaming* (Lanese et al., 2007), *context-sensitive message passing* (Caires et al., 2008), *locations* and *dynamic multiparty sessions* (Bruni et al., 2008), and *pipelines* (Boreale et al., 2008).

CaSPiS evolved from SCC because the original proposal turned out to be unsatisfactory in some respects. In particular, SCC has no dedicated mechanism for orchestrating values arising from different activities, and it has a rudimental 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 the above mentioned evolutions of SCC. The calculus proposed in (Lanese et al., 2007) is *stream*-oriented, in that values produced by sessions are stored into dedicated queues, accessible by their names. The calculus proposed in (Caires et al., 2008) 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 (Bruni et al., 2008) that allows for the dynamic joining of multiparty sessions. The second problem has been addressed in CaSPiS by introducing the property of graceful termination, which is not present in any of the other calculi.

While all the above calculi are closer to the orchestration perspective, the *global calculus* (Carbone et al., 2007) is closer to the choreography perspective and allows for static multiparty sessions, where session identifiers are modelled just as pi-calculus channel names (freshly created and distributed to participants during the initialisation phase of the service protocol). Also (Bonelli and Compagnoni, 2008) considers multiparty sessions, but they are required to include one master endpoint and one or more slave endpoints, and direct communication is allowed only between the master and any slave.

To deal with dynamic escape from ongoing conversations, a recent thread of research (Carbone et al., 2008; Carbone et al., 2009; Capecchi et al., 2010) proposes a notion of structured exceptions for communicating processes based on session types: upon a special communication action the conversation is interrupted and all peers move to a different stage. This mechanism has been named *interactional exceptions* because the interruption of the conversation must be coordinated among peers. The handling of exceptions is *structured*, in the sense that it is based on possibly nested try-catch blocks. The main differences between CaSPiS and the languages introduced to study structured interactional exceptions (SIE, for short) are the following: (i) CaSPiS has binary sessions with nesting, SIE considers unstructured multiparty sessions; (ii) in CaSPiS the escape from a session is asynchronous, in SIE it is coordinated among peers; (iii) in CaSPiS signal handlers are associated with the enclosing session, in SIE try-catch blocks are orthogonal to the ongoing sessions.

Developing safe client-service interactions requires some notion of compliance between conversation protocols. The notion of behavioural types and session types come at hand to describe the abstract sequence of interactions each session side is responsible for, allowing compliance check with varying levels of guarantees (e.g., type safety, deadlock freedom, client progress). 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. (Gay and Hole, 1999; Honda et al., 1998), or in the finite-state contract languages of (Bravetti and Zavattaro, 2007; Carpineti et al., 2006; Castagna et al., 2008).

Behavioural type systems can also play a crucial measure for evaluating the various proposals, because they offer a mean to establish the compatibility of peers (Honda, 1993; Honda et al., 1998; Carbone et al., 2007; Dezani-Ciancaglini et al., 2007; Gay and Hole, 1999; Bonelli and Compagnoni, 2008; Honda et al., 2008; Lanese et al., 2007; Acciai and Boreale, 2008b; Mezzina, 2008; Bruni and Mezzina, 2008). In this sense, it is interesting to relate behavioural types and the language independent approach based on contracts (Bravetti and Zavattaro, 2008; Castagna et al., 2008) along the ideas put forward in (Laneve and Padovani, 2008). More generally, there are some interesting analogies between the way behavioural types resemble orchestration mechanisms and contracts resemble choreography descriptions.

Two contributions exploring the problem of compliance in the setting of CaSPiS are (Acciai and Boreale, 2008b; Bruni and Mezzina, 2008), whose type systems make evident the benefits of the concept of session (see Section 7.1). A type inference algorithm is proposed in (Mezzina, 2008).

Besides the foundational aspects, also prototype implementations of CaSPiS or variants thereof have been considered. In (Bettini et al., 2008) a Java framework that permits implementing service oriented applications based on CaSPiS paradigm has been proposed. The framework, named `JCaSPiS`, provides a set of classes that implements primitives for publishing and invoking services, for defining protocols used for controlling the service interactions, and mechanisms for handling unexpected behaviours (session closures). In (Bruni et al., 2009) a service oriented abstract machine, named SOAM, has been used to develop a runtime environment for CaSPiS specifications. Moreover, by exploiting the SOS semantics of the abstract machine, correctness of the proposed implementations has been also proved.

7.1. *CaSPiS variants*

Some variants of CaSPiS that introduce suitable restrictions to favour analysis and verification of processes have been recently considered.

In (Bruni and Mezzina, 2008), it is assumed that: (1) service definitions can only be present at the top level and cannot be dynamically deployed, (2) label-guarded sums and label-choice are considered instead of guarded sums and pattern-matching, (3) the pipeline is restricted to the form $P > (?\vec{x})Q$, i.e. to Orc sequencing, (4) conditional statements are introduced, (5) session sides are polarised, (6) services are persistent and can be invoked recursively, but general replication is not allowed. Under these requirements, a type system is developed that guarantees that all session protocols are deadlock free, in the sense that well-typed processes either reach a normal form or diverge and keep opening new nested sessions. In (Acciai and Boreale, 2008b), under similar restrictions, it is shown that session names can be disregarded and a type system is provided that guarantees client-progress property (i.e., client-side protocols will not deadlock). The above results have then been extended in (Mezzina, 2009) by introducing general recursion at the level of session protocols and using the type system to prevent communication errors.

In (Kolundzija, 2009) a security-oriented extension of the work in (Bruni and Mezzina, 2008) is presented, where security levels can be assigned to service definitions, clients and data. In order to invoke a service, a client must be endowed with an appropriate clearance, and once the service and client agree on the security level, the data exchanged in the initiated session will not exceed this level. The main result is a type system that guarantees these security properties.

Besides qualitative aspects, in SOC it is also important to consider phenomena related to performance and dependability to deal with issues related to Quality of Service. They are particularly relevant for services running over congestioned networks, where unpredictable delays and failures are more likely. In (De Nicola et al., 2009) a Markovian extension of CaSPiS, called MarCaSPiS, has been studied, where: output activities are enriched with rates (characterising random variables with exponential distributions) and input activities are equipped with weights (characterising the relative selection probability). Then continuous time Markov chains can be obtained from MarCaSPiS specifications to perform quantitative analysis.

Some recent work is also concerned with graphical encoding and concurrent semantics for SOC calculi (Bruni et al., 2010a; Bruni et al., 2010b). Models based on hierarchical graphs are used that best reflect the nesting of sessions and the possibility to operate on the nested session sides.

8. Conclusions

We have presented CaSPiS, a core calculus for service-oriented applications. One of the key notions of CaSPiS is that of session, which allows for the definition of arbitrarily structured interaction modalities between clients and services. We have provided CaSPiS with a fully formalized operational semantics in the SOS style and with a simple discipline to program graceful session termination. Several examples witness the expressiveness of our proposal. A recent prototype implementation is also available (Bettini et al., 2008).

The design of CaSPiS has been influenced by the π-calculus (Milner et al., 1992) and by Orc (Misra and Cook, 2007). 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). 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 (Boreale et al., 2006)). As our examples show, sessions and pipelines are handy constructs that is worth having as first-class objects. The advantages of having them become even more evident when dealing with session termination, which has no (obvious) counterpart in the π-calculus.

Regarding future work, we would like to move along two directions one more theoretical to study the impact of adding a mechanism of *delegation* to the calculus, another more practical that considers the possibility of developing full fledged programming languages based on foundational calculi.

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.

Foundational calculi can also have an important rôle in designing new programming languages. These would benefit pragmatically because their primitives would be based on a rigorous semantics and could take advantage of it to exploit tools and techniques for the formal verification of important properties of services like deadlock freedom, client progress, eventual servicing, protocol conformance. What we expect is new, richer and more usable service oriented programming languages for writing programs that could be the basis (possibly after stripping off some information) for obtaining a sort of "program skeleton" to be checked and validated with the tools developed for the foundational calculi. Moreover, working in the opposite direction, programmers could rely on the calculi to model system components and their interactions and then use the specification to obtain a model for driving the development process. Analysis of important properties could then be performed at the model level to acquire guarantees of correct behaviours of the derived implementation.

Moreover, 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 (Bruni et al., 2004; Laneve and Zavattaro, 2005), and its relationship with the cCSP and the Sagas-calculi of (Bruni et al., 2005).

Acknowledgements: First and foremost, we would like to thank Ivan Lanese and Davide Sangiorgi, the editors of this special issue, for inviting us to submit this contribution. We also thank the anonymous referees for their helpful revisions and comments on the preliminary version of this paper. Last but not least, we would like to thank all researchers and reviewers of the Sensoria project with whom we had many stimulating discussions that significantly contributed to the shaping of CaSPiS.

References

Acciai, L. and Boreale, M. (2008a). Responsiveness in process calculi. *Theoret. Comput. Sci.*, 409(1):59–93.

Acciai, L. and Boreale, M. (2008b). 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.*, pages 642–658. Springer.

Acciai, L. and Boreale, M. (2008c). Xpi: A typed process calculus for xml messaging. *Sci. Comput. Program.*, 71(2):110–143.

Bartoletti, M., Degano, P., Ferrari, G., and Zunino, R. (2007). Types and effects for Resource Usage Analysis. In Seidl, H., editor, *Proceedings of FOSSACS'07*, volume 4423 of *Lect. Notes in Comput. Sci.*, pages 32–47. Springer.

Bettini, L., De Nicola, R., and Loreti, M. (2008). Implementing Session Centered Calculi. In Lea, D. and Zavattaro, G., editors, *Proceedings of COORDINATION'08*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 17–32. Springer.

Bodei, C., Brodo, L., and Bruni, R. (2009). Static detection of logic flaws in service applications. In Degano, P. and Viganò, L., editors, *Proceedings of ARSPA-WITS'09*, volume 5511 of *Lect. Notes in Comput. Sci.*, pages 70–87. Springer.

Bonelli, E. and Compagnoni, A. (2008). Multisession session types for a distributed calculus. In Barthe, G. and Fournet, C., editors, *Proceedings of TGC'07*, volume 4912 of *Lect. Notes in Comput. Sci.*, pages 240–256. Springer.

Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., and Zavattaro, G. (2006). SCC: a service centered calculus. In Bravetti, M., Núñez, M., and Zavattaro, G., editors, *Proceedings of WS-FM'06*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 38–57. Springer.

Boreale, M., Bruni, R., De Nicola, R., and Loreti, M. (2008). Sessions and pipelines for structured service programming. In Barthe, G. and de Boer, F. S., editors, *Proceedings of FMOODS'08*, volume 5051 of *Lect. Notes in Comput. Sci.*, pages 19–38. Springer.

Boreale, M. and Sangiorgi, D. (1998). A fully abstract semantics for causality in the pi-calculus. *Acta Inf.*, 35(5):353–400.

Bravetti, M. and Zavattaro, G. (2007). A theory for strong service compliance. In Murphy, A. L. and Vitek, J., editors, *Proceedings of COORDINATION'07*, volume 4467 of *Lect. Notes in Comput. Sci.*, pages 96–112. Springer.

Bravetti, M. and Zavattaro, G. (2008). A Foundational Theory of Contracts for Multi-party Service Composition. *Fundamenta Informaticae*, 89(4):451–478.

Bruni, R. (2009). Calculi for service-oriented computing. In Bernardo, M., Padovani, L., and Zavattaro, G., editors, *Proceedings of SFM-WS'09*, volume 5569, pages 1–41. Springer.

Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H., and Montanari, U. (2005). Comparing two approaches to compensable flow composition. In Abadi, M. and de Alfaro, L., editors, *Proceedings of CONCUR'05*, volume 3653 of *Lect. Notes in Comput. Sci.*, pages 383–397. Springer.

Bruni, R., De Nicola, R., Loreti, M., and Mezzina, L. (2009). Provably correct implementations of services. In Kaklamanis, C. and Nielson, F., editors, *Proceedings of TGC'08*, volume 5474 of *Lect. Notes in Comput. Sci.*, pages 69–86. Springer.

Bruni, R., Gadducci, F., and Lluch-Lafuente, A. (2010a). An algebra of hierarchical graphs and its application to structural encoding. *Scientific Annals of Computer Science*, 20:53–96.

Bruni, R., Lanese, I., Melgratti, H., and Tuosto, E. (2008). Multiparty sessions in SOC. In Lea, D. and Zavattaro, G., editors, *Proceedings of COORDINATION'08*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 67–82. Springer.

Bruni, R., Liu, Z., and Zhao, L. (2010b). Graph representation of sessions and pipelines for structured service programming. In *Proceedings of FACS'10*, Lect. Notes in Comput. Sci. Springer. To appear.

Bruni, R., Melgratti, H., and Montanari, U. (2004). Nested commits for mobile calculi: extending Join. In Lévy, J.-J., Mayr, E., and Mitchell, J., editors, *Proceedings of IFIP-TCS'04*, pages 569–582. Kluwer Academic Publishers.

Bruni, R. and Mezzina, L. (2008). Types and deadlock freedom in a calculus of services, sessions and pipelines. In Rosu, G. and Meseguer, J., editors, *Proceedings of AMAST'08*, volume 5140 of *Lect. Notes in Comput. Sci.*, pages 100–115. Springer.

Buscemi, M. and Montanari, U. (2007). CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In De Nicola, R., editor, *Proceedings of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 18–32. Springer.

Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., and Zavattaro, G. (2006). Choreography and orchestration conformance for system design. In Ciancarini, P. and Wiklicky, H., editors, *Proceedings of COORDINATION'06*, volume 4038 of *Lect. Notes in Comput. Sci.*, pages 63–81. Springer.

Caires, L., Vieira, H. T., and Seco, J. C. (2008). The conversation calculus: A model of service oriented computation. In Drossopoulou, S., editor, *Proceedings of ESOP'08*, volume 4960 of *Lect. Notes in Comput. Sci.*, pages 269–283. Springer.

Capecchi, S., Giachino, E., and Yoshida, N. (2010). Global escape in multiparty sessions. In Lodaya, K. and Mahajan, M., editors, *Proceedings of FSTTCS'10*, volume 8 of *LIPIcs*, pages 338–351. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Carbone, M., Honda, K., and Yoshida, N. (2007). Structured communication-centred programming for web services. In De Nicola, R., editor, *Proceedings of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 2–17. Springer.

Carbone, M., Honda, K., and Yoshida, N. (2008). Structured interactional exceptions in session types. In van Breugel, F. and Chechik, M., editors, *Proceedings of CONCUR'08*, volume 5201 of *Lect. Notes in Comput. Sci.*, pages 402–417. Springer.

Carbone, M., Yoshida, N., and Honda, K. (2009). Asynchronous session types: Exceptions and multiparty interactions. In Bernardo, M., Padovani, L., and Zavattaro, G., editors, *Proceedings of SFM-WS'09*, volume 5569 of *Lect. Notes in Comput. Sci.*, pages 187–212. Springer.

Carpineti, S., Castagna, G., Laneve, C., and Padovani, L. (2006). A formal account of contracts for web services. In Bravetti, M., Núñez, M., and Zavattaro, G., editors, *Proceedings of WS-FM'06*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 148–162. Springer.

Castagna, G., Gesbert, N., and Padovani, L. (2008). A theory of contracts for web services. In Necula, G. C. and Wadler, P., editors, *Proceedings of POPL'08*, pages 261–272, New York, NY, USA. ACM.

Cook, W. R., Patwardhan, S., and Misra, J. (2006). Workflow patterns in Orc. In Ciancarini, P. and Wiklicky, H., editors, *Proceedings of COORDINATION'06*, volume 4038 of *Lect. Notes in Comput. Sci.*, pages 82–96. Springer.

De Nicola, R., Latella, D., Loreti, M., and Massink, M. (2009). MarCaSPiS: a markovian extension of a calculus for services. *Electronic Notes in Theoretical Computer Science*, 229(4):11–26.

Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., and Drossopoulou, S. (2007). A distributed object-oriented language with session types. In De Nicola, R. and Sangiorgi, D., editors, *Proceedings of TGC'05*, volume 3705 of *Lect. Notes in Comput. Sci.*, pages 299–318. Springer.

Ferrari, G., Guanciale, R., and Strollo, D. (2006). JSCL: A Middleware for Service Coordination. In Najm, E., Pradat-Peyre, J.-F., and Donzeau-Gouge, V., editors, *Proceedings of FORTE'06*, volume 4229 of *Lect. Notes in Comput. Sci.*, pages 46–60. Springer.

Ferrari, G., Guanciale, R., Strollo, D., and Tuosto, E. (2007). Coordination via types in an event-based framework. In Derrick, J. and Vain, J., editors, *Proceedings of FORTE'07*, volume 4574 of *Lect. Notes in Comput. Sci.*, pages 66–80.

Gay, S. J. and Hole, M. J. (1999). Types and subtypes for client-server interactions. In Swierstra, S. D., editor, *Proceedings of ESOP'99*, volume 1576 of *Lect. Notes in Comput. Sci.*, pages 74–90. Springer.

Honda, K. (1993). Types for dyadic interaction. In Best, E., editor, *Proceedings of CONCUR'93*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 509–523. Springer.

Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In Hankin, C., editor, *Proceedings of ESOP'98*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 122–138. Springer.

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty asynchronous session types. In Necula, G. C. and Wadler, P., editors, *Proceedings of POPL'08*, pages 273–284. ACM.

Kitchin, D., Cook, W. R., and Misra, J. (2006). A language for task orchestration and its semantic properties. In Baier, C. and Hermanns, H., editors, *Proceedings of CONCUR'06*, volume 4137 of *Lect. Notes in Comput. Sci.*, pages 477–491. Springer.

Kolundzija, M. (2009). Security types for sessions and pipelines. In Bruni, R. and Wolf, K., editors, *Proceedings of WS-FM'08*, volume 5387 of *Lect. Notes in Comput. Sci.*, pages 176–190. Springer.

Lanese, I., Vasconcelos, V., Martins, F., and Ravara, A. (2007). Disciplining orchestration and conversation in service-oriented computing. In *Proceedings of SEFM'07*, pages 305–314. IEEE Computer Society Press.

Laneve, C. and Padovani, L. (2008). The pairing of contracts and session types. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lect. Notes in Comput. Sci.*, pages 681–700. Springer.

Laneve, C. and Zavattaro, G. (2005). Foundations of web transactions. In Sassone, V., editor, *Proceedings of FOSSACS'05*, volume 3441 of *Lect. Notes in Comput. Sci.*, pages 282–298. Springer.

Lapadula, A., Pugliese, R., and Tiezzi, F. (2007). A calculus for orchestration of web services. In De Nicola, R., editor, *Proceedings of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 33–47. Springer.

Mezzina, L. (2008). How to infer finite session types in a calculus of services and sessions. In Lea, D. and Zavattaro, G., editors, *Proceedings of COORDINATION'08*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 216–231. Springer.

Mezzina, L. (2009). *Typing Services*. PhD in Computer Science and Engineering, IMT Institute for Advanced Studies, Lucca.

Milner, R., Parrow, J., and Walker, J. (1992). A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77.

Misra, J. and Cook, W. R. (2007). Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 6(1):83–110.

OASIS (2007). *Web Services Business Process Execution Language Version 2.0, Working Draft.* `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`.

Sensoria Project (2010). Software Engineering for Service-Oriented Overlay Computers. Public Web Site. `http://sensoria.fast.de/`.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51.

Wehrman, I., Kitchin, D., Cook, W. R., and Misra, J. (2008). A timed semantics of Orc. *Theoretical Computer Science*, 402(2-3):234–248.

Wirsing, M. and Hölzl, M., editors (2011). *Rigorous Software Engineering for Service-Oriented Systems. Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *Lect. Notes in Comput. Sci.* Springer.

Appendix A. Proof steps for Theorem 6.1

This technical section is devoted to the proof of the two lemmas mentioned in the proof of Theorem 6.1, namely Lemma A.3 and A.4 stated below.

To this aim, it is convenient to introduce some additional terminology. Recall that $\mathbb{S}[\,\cdot\,], \mathbb{S}'[\,\cdot\,]$ range over session-immune contexts. We say a term is

— (r,k)-*closing* if it is of the form $r \rhd_k \mathbb{S}[\,\mathsf{close}\,]$ or $r \rhd_k \mathbb{S}[\,\dagger(k')|k' \cdot \mathbb{S}'[\,\mathsf{close}\,]\,]$, for suitable $\mathbb{S}[\,\cdot\,], \mathbb{S}'[\,\cdot\,], k'$;

— (r,k)-*closing after* k' if it is of the form $r \rhd_k \mathbb{S}[\,k' \cdot \mathbb{S}'[\,\mathsf{close}\,]\,]$, for some $\mathbb{S}[\,\cdot\,], \mathbb{S}'[\,\cdot\,]$.

We write "R is (r,k)-closing (after k')" if R is either (r,k)-closing or (r,k)-closing after k'. Then we define:

— an *r-balanced pair* to be a pair of terms (R, R') such that, for some distinct k and k', R is (r,k)-closing (after k') and R' is (r,k')-closing (after k);

— an *r-self-balanced term* to be either an (r,k)-closing term or a term of the form $\blacktriangleright R$, with R (r,k)-closing (after k'), for some r, k and k';

— an *r-side* is a term of the form $r \rhd_k R$.

With the above terminology, it is technically convenient to re-write the definitions of r-balanced and quasi r-balanced process in the following, equivalent manner. A process P is *r-balanced* if either $r \notin \mathrm{fn}(P)$ or $P \equiv Q$ and for some Q containing two terms that form an r-balanced pair. A process P is *quasi r-balanced* if either P is r-balanced or $P \equiv Q$ for some Q containing a single, r-self-balanced term.

Our next task is showing that r-balancing is preserved through reductions: this is the content of Lemma A.2, whose proof necessitates of the following lemma about contexts. In the following, we say a context $\mathbb{D}[\,\cdot\,]$ is *quasi-static* if its hole falls only in the scope of static operators *or* the $\blacktriangleright$ operator. In what follows, $\mathbb{D}[\,\cdot\,], \mathbb{D}'[\,\cdot\,], \ldots$ will range over 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.

Lemma A.1. It holds that

1 For any quasi-static $\mathbb{D}[\,\cdot,\cdot\,]$ and $\mathbb{D}'[\,\cdot\,]$, we have that $\mathbb{D}[\,\dagger(k), \mathbb{D}'[\,P\,]\,] \equiv \mathbb{D}[\,\mathbf{0}, \mathbb{D}'[\,\dagger(k)|P\,]\,]$.

2 For any static $\mathbb{C}[\,\cdot\,]$, we have that $\blacktriangleright \mathbb{C}[\,P\,] \equiv \mathbb{D}[\,\blacktriangleright P\,]$, where $\mathbb{D}[\,\cdot\,]$ is the quasi-static context obtained from $\mathbb{C}[\,\cdot\,]$ by replacing each "$r\rhd_k$" with "$\blacktriangleright\, r\rhd_k$", for each r and k.

Proof. An easy induction on $\mathbb{D}[\,\cdot,\cdot\,], \mathbb{D}'[\,\cdot\,]$ and $\mathbb{C}[\,\cdot\,]$ that exploits the structural rules for $\dagger(k)$ and $\blacktriangleright$. $\square$

Lemma A.2. Let P be graceful and quasi r-balanced. Suppose $P \xrightarrow{\tau} Q$. Then Q is quasi r-balanced.

Proof. The reduction $P \xrightarrow{\tau} Q$ must fall in one of the cases 1–8 of Lemma 6.2. The proof proceeds by analyzing each of these cases separately. In fact, we just look at the only two non obvious ones, which are 6 and 8.

(6) Let $A \triangleq \mathbb{C}[\, r' \rhd_k \mathbb{S}_0[\,\mathsf{close}\,]\,]$ and $B \triangleq \mathbb{C}[\,\blacktriangleright \mathbb{S}_0[\,\mathbf{0}\,]|\dagger(k)\,]$. We distinguish the case $r' \neq r$ from $r' = r$.
Assume $r' \neq r$. Then, B is obtained from A by replacing $r' \rhd_k \mathbb{S}_0[\,\mathsf{close}\,]$ with $\blacktriangleright \mathbb{S}_0[\,\mathbf{0}\,]|\dagger(k)$ inside $\mathbb{C}[\,\cdot\,]$. This operation does not affect quasi r-balancing. To see this, let us consider here the case where in A we find a signal $\dagger(k')$ in $\mathbb{C}[\,\mathsf{close}\,]$ and an (r, k'')-closing after k' term, $r \rhd_{k''} R$, in $\mathbb{C}[\,\mathsf{close}\,]$: so that, up to structural congruence, in A there is an r-self-balanced term $r \rhd_{k''} (R|\dagger(k'))$ (the other cases are obvious). Applying Lemma A.1(1), in B we can still move $\dagger(k')$ from within $\blacktriangleright \mathbb{C}[\,\mathbf{0}\,]$ and place it inside $r \rhd_{k''} (\cdots)$. Doing so we get, up to structural congruence, a term $r \rhd_{k''} (R|\dagger(k'))$, which is self-balanced. Hence B is quasi r-balanced.
Assume now $r' = r$. If no r-side occurs in B, or if just a single, self-balanced r-side occurs in B, there is nothing to prove. So assume a non self-balanced $R \triangleq r \rhd_{k'} Q_0$ $(k' \neq k)$ occurs in B. Since R cannot occur inside $\mathbb{S}_0[\,\mathbf{0}\,]$, it must occur in $\mathbb{C}[\,\cdot\,]$, hence also in A. Hence, choosing a suitable quasi-static $\mathbb{D}[\,\cdot,\cdot\,]$ we can write

$$\begin{array}{rcl} A & = & \mathbb{D}[\, r \rhd_{k'} Q_0, r \rhd_k \mathbb{S}_0[\,\mathsf{close}\,]\,] \\ B & = & \mathbb{D}[\, r \rhd_{k'} Q_0, \mathbb{S}_0[\,\mathbf{0}\,]\,] \end{array}$$

In A, R must necessarily form an r-balanced pair with $r \rhd_k \mathbb{S}_0[\,\mathsf{close}\,]$. Hence, R must be (r, k')-closing after k, which implies $Q_0 \equiv \mathbb{S}_1[\, k \cdot \mathbb{S}_2[\,\mathsf{close}\,]\,]$ for static, session immune $\mathbb{S}_1[\,\cdot\,], \mathbb{S}_2[\,\cdot\,]$. Using Lemma A.1(1), we then have that $B \equiv \mathbb{D}[\, r \rhd_{k'} \mathbb{S}_1[\,\dagger(k)|k \cdot \mathbb{S}_2[\,\mathsf{close}\,]\,], \mathbf{0}\,]$, where there is a single r-side which is (r, k')-closing, hence r-self-balanced. So B and hence Q are quasi r-balanced.

(8) Let $A \triangleq \mathbb{C}[\,\dagger(k)|k \cdot R\,]$ and $B \triangleq \mathbb{C}[\,R\,]$. If in A there is no term which is (r, k')-closing after k, for some $k' \neq k$, then the consumption of the listener $k\cdot$ and of signal $\dagger(k)$ does not affect quasi r-balancing, so B is quasi r-balanced, hence Q is.
Assume the contrary, that is, in A there is a term which is (r, k')-closing after k, with $k' \neq k$. By definition of "(r, k')-closing after k", this subterm of A must of the form $R_1 = r \rhd_{k'} \mathbb{S}_0[\, k \cdot \mathbb{S}_1[\,\mathsf{close}\,]\,]$, with $\mathbb{S}_0[\,\cdot\,], \mathbb{S}_1[\,\cdot\,]$ static, session-immune. By uniqueness of the listener $k\cdot$ in A (by graceful-ness), it must be $R = \mathbb{S}_1[\,\mathsf{close}\,]$. Then B is obtained from A by replacing the subterm $R_1 = r \rhd_{k'} \mathbb{S}_0[\, k \cdot \mathbb{S}_1[\,\mathsf{close}\,]\,]$ by $R_2 = r \rhd_{k'} \mathbb{S}_0[\,\mathbb{S}_1[\,\mathsf{close}\,]\,]$, which is (r, k')-closing. Hence quasi r-balancing is unaffected, so B, hence Q, are quasi r-balanced.

$\square$

A fundamental property of quasi-balancing is that it is preserved through reductions.

Lemma A.3. Let P be quasi-balanced and $P \xrightarrow{\tau} Q$. Then Q is quasi-balanced.

Proof. By Lemma 6.1, we already know that Q is graceful. By definition, it must be $P \equiv$

$(\nu\,\tilde{r})P_0$, where P_0 has no top bound sessions, $\tilde{r} \subseteq \mathrm{fn}(P_0)$ and for each $r' \in \mathrm{fn}(P_0)$, P_0 is quasi r'-balanced.

Now, $P \xrightarrow{\tau} Q$ implies that for some Q_0 we have that $P_0 \xrightarrow{\tau} Q_0$ and $Q \equiv (\nu\,\tilde{r})Q_0$. Since $\mathrm{fn}(Q_0) \subseteq \mathrm{fn}(P_0)$, by Lemma A.2 Q_0 is quasi r'-balanced for each $r' \in \mathrm{fn}(Q_0)$. There are now two cases.

— Q_0 has no top bound session. We can conclude, by structural congruence, that Q is quasi-balanced.

— Q_0 has (one) top bound session. Recalling that, in well-formed terms, session names cannot appear within the scope of dynamic operators, the only possibility according to Lemma 6.2 is that the reduction $P_0 \xrightarrow{\tau} Q_0$ falls in case 3. Then $Q_0 \equiv (\nu\,r_0)Q_0'$ with Q_0' r_0-balanced (by graceful-ness) and having no top bound sessions. In the end, $Q \equiv (\nu\,\tilde{r}r_0)Q_0'$, where for each $r' \in \tilde{r}r_0$, Q_0' is quasi r'-balanced. Hence, by structural congruence, Q is quasi-balanced.

□

The following lemma states the second fundamental property of quasi-balancing.

Lemma A.4. Let P be quasi-balanced. Then there is Q' balanced such that $P \xrightarrow{\tau}{}^{*} Q$.

Proof. By definition, $P \equiv (\nu\,\tilde{r}')P_0$, where P_0 has no top bound sessions, $\tilde{r} \subseteq \mathrm{fn}(P_0)$ and for each $r \in \mathrm{fn}(P_0)$, P_0 is quasi r'-balanced.

We now show that there is a balanced Q_0 such that $P_0 \xrightarrow{\tau}{}^{*} Q_0$. This will imply the wanted result, as we then have that $P \xrightarrow{\tau}{}^{*} Q \triangleq (\nu\,\tilde{r})Q_0$ with Q balanced as well.

For the actual proof, we introduce the following set and quantities. We let $\mathrm{sn}(P)$ be the set of free session name that occur in P.

$$\begin{array}{rcl}
\mathrm{unb}(P_0) & \triangleq & |\{r' \in \mathrm{sn}(P_0) : \ P_0 \text{ is not } r'\text{-balanced }\}| \\
\mathrm{pre}_{P_0}(r) & \triangleq & \{r' \,:\, r' \prec^{+}_{P_0} r\} \\
\mathrm{preunb}(P_0) & \triangleq & \min_{r \in \mathrm{unb}(P_0)} |\mathrm{pre}_{P_0}(r)|\,.
\end{array}$$

(In the last definition above, we stipulate that $\min \emptyset \triangleq 0$). We proceed by lexicographic induction on $(\mathrm{unb}(P_0), \mathrm{preunb}(P_0))$. If $\mathrm{unb}(P_0) = 0$, then also $\mathrm{preunb}(P_0) = 0$, and P_0 is balanced, hence we can take $Q_0 = P_0$. Assume $\mathrm{unb}(P_0) > 0$. Take $r \in \mathrm{sn}(P_0)$ that minimizes $|\mathrm{pre}_{P_0}(r)|$ among all the non-balanced session names, in other words take an r s.t. $|\mathrm{pre}_{P_0}(r)| = \mathrm{preunb}(P_0)$. Consider the set $\mathrm{pre}_{P_0}(r)$, there are two possibilities:

— In P_0, no $r' \in \mathrm{pre}_{P_0}(r)$ occurs underneath a dead session operator $\blacktriangleright$. Now, P_0 is by hypothesis r-self-balanced. This means $P_0 \equiv \mathbb{C}[\,R\,]$ for a suitable static $\mathbb{C}.[\ ]$ and R such that either R is (r,k)-closing or $R \Rightarrow\!\!\blacktriangleright R'$ for R' (r,k)-closing (after k'), for some k and k'. In any case, R, hence P_0, can reduce in 1 or 2 steps, to a term where r has been removed (use either rules Tsync, Send or rule Tend). By inspection, in this term the balancing of any session name $r' \neq r$ is unaffected. To sum up, we have found a sequence $P_0 \xrightarrow{\tau}{}^{i} P_0'$ $(i \leq 2)$ with $\mathrm{unb}(P_0') = \mathrm{unb}(P_0) - 1$. Moreover P_0' is still quasi-balanced (Lemma A.3), and has no top bound sessions, because we have applied only rules that do not generate new sessions. So we can apply the induction hypothesis and conclude.

— In P_0, there is $r' \in \mathrm{pre}_{P_0}(r)$ that occurs underneath a dead session operator $\blacktriangleright$. Choose one such r' with the property that no $r'' \in \mathrm{pre}_{P_0}(r')$ occurs underneath $\blacktriangleright$ in P_0: this r' must exist by acyclicity of $\prec_{P_0}$. Moreover, being $\mathrm{pre}_{P_0}(r')$ strictly included in $\mathrm{pre}_{P_0}(r)$ (again by acyclicity), we deduce that P_0 is r'-balanced. To sum up, we can write, for suitable contexts and terms

$$P_0 \equiv \mathbb{C}[\, \blacktriangleright \mathbb{S}[\, r' \rhd_k R\,]\,,\, R'\,] \;\equiv\; \mathbb{C}[\, \mathbb{S}[\, \blacktriangleright r' \rhd_k R\,]\,,\, R'\,]$$

where: $\mathbb{C}[\,\cdot,\cdot\,]$ is static, $\mathbb{S}[\,\cdot\,]$ is static session-immune and R' contains an r-side forming with $r' \rhd_k R$ a mutually balanced pair (in the second equivalence above, we have used Lemma A.1(2) to push $\blacktriangleright$ in front of $r''\rhd$). Concerning R', since r' cannot be in the scope of any r'' that is under $\blacktriangleright$, there are two cases: either (a) $R' \;\equiv\; r' \rhd_{k'} R''$ or (b) $R' \;\equiv \blacktriangleright \mathbb{S}'[\, r' \rhd_{k'} R''\,] \;\equiv\; \mathbb{S}'[\, \blacktriangleright\, r' \rhd_{k'} R''\,]$ for some static, session-immune $\mathbb{S}'[\,\cdot\,]$ (in last equivalence, we have again used implicitly Lemma A.1(2)). In either case, we can reduce P_0 in few steps into to a term P_0' where both occurrences of r' have been deleted: first apply Tend to $\mathbb{S}[\, \blacktriangleright\, r' \rhd_k R\,]$, then either (a) use the resulting $\dagger(k)$ (rules Tsync, Send), or (b) apply again Tend to eliminate the remaining r-side. In all cases, these reductions do not affect the balancing conditions of any $r'' \neq r'$. Hence we have $\mathrm{unb}(P_0') = \mathrm{unb}(P_0)$ but $\mathrm{pre}_{P_0'}(r) = \mathrm{pre}_{P_0}(r) - 1$. Moreover P_0' is still quasi-balanced (Lemma A.3), and has no top bound sessions, because we have applied only rules that do not generate new sessions. So we can apply the induction hypothesis and conclude.

□