Calculi for Service-Oriented Computing⋆

Roberto Bruni

Dipartimento di Informatica, Università di Pisa
bruni@di.unipi.it

Abstract. 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. Nowadays, the increasing popularity of Service-Oriented Computing (SOC) challenges the quest for novel abstractions tailored to the well-disciplined handling of specific issues, like long running interactions, orchestration, and unexpected events. In fact, these features emerge neatly in most SOC applications and need to be studied as first-class aspects, whereas they would be obfuscated if dealt with by sophisticated encoding in traditional process calculi. This paper overviews some of the most recent proposals emerged in the literature, pointing out their main characteristics and presents in more detail one such proposal, called CaSPiS, by providing several examples to give evidence of its flexibility. No prior acquaintance with process calculi is assumed, indeed a gentle introduction to their basics is provided before the more advanced material be presented.

1 Introduction

Service-oriented computing has been one of the latest trend in the IT community, finding in Web Services (WS) technology its major realisation. Services are autonomous computational entities, that are developed separately, loosely coupled, globally available over a widely distributed network in a platform-independent way, and not fully reliable. Service computing consists of assembling services in well-engineered ways to form complex open-ended applications, and this must be done in a highly dynamic way, possibly on demand. To this aim, it is essential to find suitable abstractions to describe services, the so-called *service descriptors*, to be published in public registries. Such registries can be queried by other services and applications to locate those services that best match certain requirements, yielding a brokering architecture. When satisfactory matches are found, then the located services can be dynamically linked and invoked. Therefore, service engineering has to do with the development of methodologies, techniques, formal methods and tools able to guarantee a safe service composition, in the sense of being able to provide some strong guarantees on such dynamic, open-ended applications by applying some static or semi-static analysis.

WS technology has established *de facto* standards for naming schemes and service access (URI, URL), service descriptors (WSDL and BPEL in UDDI registries),

⋆ Research supported by the Project FET-GC II IST-2005-16004 S and by the Italian FIRB Project T . .

communication protocols (SOAP over HTTP, TCP/IP and SMTP) and message format (XML) over the web. Existing infrastructures already enable providers to describe web services in terms of their interfaces, access policies and behaviour, and to combine simpler services into more structured and complex ones. However, some research and solid foundations are still needed to move WS technology from skilled handcrafting to an engineered practice, a step where formal methods must play a fundamental role. For example, it has been shown that the lack of unambiguous semantics of BPEL has led different BPEL engines to exhibit different behaviours under the same circumstances [43].

Research on formal methods for SOC can be roughly separated in two main strands, both equally worth the effort: one dedicated to establishing the missing theoretical foundations of state-of-the-art technologies, so to fix rigorous semantic and logic frameworks for the analysis and verification of SOC and WS systems; another one aimed to rethink the design and development of next generation technology, by understanding the key distinguishing features of SOC, assessing the necessary bits of theory for them in technology agnostic terms, and paving the way to their well-disciplined engineering. In both cases, the mathematical models and tools from the literature that seem to be particularly suited are those coming from concurrency theory, ranging from workflow like models like Petri nets, to Graph Transformation systems and process calculi.

As suggested by the title of this contribution, we shall focus on the use of process calculi for modelling SOC systems. This choice is motivated by the more natural way in which process calculi can accommodate for SOC features such as open-endedness, dynamicity, compositionality, interaction and event handling w.r.t. the other afore mentioned models. Moreover, we shall favour the second strand of research outlined above, trying to distill some key aspects of SOC together with a small set of primitives associated with them and to expose some of the main causes (motivation) and consequences (benefits) of our approach.

Due to the particular nature of this volume, which contains the proceedings of a summer school, and the audience to which this paper is oriented, which for the most we assume to consist of young computer science researchers, we have decided to structure this paper as a tutorial, so that no prior acquaintance with process calculi is assumed on the reader. Being worried that the more expert readers can find some arguments of our survey not dealt with at the sufficient levels of details for their taste, we added, whenever necessary, suitable links to the more advanced papers and texts where the technicalities are exposed in their full glory. Furthermore, we have put some efforts in trying to accompany each calculus by original examples and modelling puzzles in the hope they will provide an enjoyable reading experience by themselves, possibly reusable as course material.

1.1 What you will not find here

The level of abstraction at which we intend to model SOC systems disregards the technologies and the implementation details, hence we are to some extent disconnected from current WS standards. More precisely, we disregards those aspects related to the so-called semantic web, like ontologies for classifying services, XML coding and standardisation issues. In fact these aspects can be superimposed later, on the concrete realisation of our techniques.

For analogous reasons, we are not concerned with the exact ways in which services and their descriptors are made public available, queried and located, even if some of these issues can be reasonably encoded in the same formalisms we shall present. Instead we handle service publication, discovery and linking in terms of name-handling á la pi-calculus, i.e. the scope of certain service names can be restricted, new services can be dynamically deployed and updated, their names can be communicated and extruded to enlarge their scope, etc.

Moreover, we abstract away from non-functional aspects (like Quality of Service and Service Level Agreement) and quantitative analysis, which also constitute themselves an active area of research. On the other hand, some preliminary ideas in this field have already led to process calculi extensions that are compatible with the proposals discussed here and we give relevant pointers to the related literature.

We also deliberately omit the exact formulation of many useful theorems (and any proof sketch) from the literature, which we try to replace by more intuitive descriptions of their underlying properties and consequences, at the informal level.

1.2 Aspects of interest

The common trait of all issues we aim to encompass here is the handy, disciplined composition of services. This includes: the possibility to extract service descriptors that carry some behavioural information rather than mere syntactic information as those found in WSDL documents; the possibility to carry long-running conversation between the service caller and its callee, which are far more general than limited one-way and request-response patterns of WS; suitable techniques for checking the behavioural conformance of the service to be invoked w.r.t. the application requirements; the way in which service invocations and their outcomes can be orchestrated; the way in which the system can foresee at the design time the actions to undergo in case some unexpected event will happen during a conversation, like a peer abandoning a business transaction. More precisely, we briefly discuss below different alternatives proposed in the literature on the above topics, and outline our preferred design choices.

Orchestration and choreography: The terms *orchestration* and *choreography* were coined to describe two different flavours of service compositions: orchestration is about describing and executing a single view point model, while choreography is about specifying and guiding a global model. Though the difference between the two terms can be sometimes abused or blurred, substantially orchestration is usually associated with an executable flavour, for which a centralised orchestration engine is responsible (although distributed engines can be also considered), as opposed to the fully distributed vision of choreography, usually associated with some sort of protocol narration. Roughly, from a formal modelling viewpoint, orchestration is mainly concerned with governing the control and data flow between services, while choreography is concerned with interaction protocols between single and composite autonomous services. Our presentation shall privilege orchestration, but out approach is compatible with the choreography perspective, as the type systems defined to check the conformance of services w.r.t. the requirements of the invoker share some similarities with the use of so-called *contracts* to express choreographies.

Interaction: Process calculi can exploit different forms of interactions, ranging from shared data-space, to event-based (subscribe-notify), and message passing. We shall rely on synchronous message passing, that is best suited for the level of abstraction of this tutorial.

Sessions and correlation sets: When long-running conversation with services are established, different instances of the same service can be running concurrently to serve different requests. Therefore it is important to route interaction between the correct pairs, avoiding any interference. Web service standards exploit the idea of *correlation sets*, i.e., pre-defined subsets of the invocation parameters that are used each time to choose the corresponding service instance (e.g., requests are routed according to usernames). Though correlation sets offer a good expressiveness, we argue that they might complicate static analysis, because all interactions rely on data values. For example, applications can interfere with each other if they know (or use by chance) the right values. A different school of thought advocates the notion of a *session* as a more convenient abstraction mechanism for enclosing arbitrarily complex interactions between peers. Session keys are data-independent and can be created implicitly when the service is first invoked. This way, type systems can be more easily developed to check properties like the presence of exactly one peer. Session can come in different flavours: nested, interleaved, with delegation, used recursively, dyadic, multy-party, mergeable, closeable, permeable, etc. We shall focus on primitives for a well-disciplined use of nested, dyadic and closeable sessions.

Compensations and session handlers: Each service has full autonomy in denying a request or abandoning a pending interaction. It is then important to rely on standard mechanisms for programming such decisions and to handle their consequences in a *safe* way. For example, the classical travel agency scenario may involve a complex interaction between the customer and the travel agent, to let the service learn the customer preferences, let the customer select one among available packages, confirm or cancel the choice, and the service may need to invoke third-party services to get, say, up-to-date flight or hotel information. By *safe*, we mean that, in principle, the involved parties should always be able either to complete the interaction or to recover from errors that prevent its completion, like when a time-out expires or when one of the third-party services unexpectedly abandon the conversation because its server is overloaded. In the area of transactions, compensation mechanisms have to do with the programming of suitable counter-actions that are installed after a certain activity has been executed to compensate for its effects in case the rest of the interaction cannot be completed successfully. Of course, it is often the case that the previous actions cannot be simply undone (e.g., a sent message cannot disappear, booking cancellation can require some fees) hence full recovery is simply not possible. In the case of sessions, we shall consider a simple built-in mechanism for the graceful closure of nested sessions upon the abandon of a peer.

The outcome of the above consideration language was a new calculus, called CaSPiS (*Calculus of Sessions and Pipelines*) [8], which is the main objective of this tutorial.

1.3 Related work

CaSPiS has been developed inside the S project [57], as part of a larger research effort aimed to develop *core calculi for SOC* at three different levels of abstraction: (i) the service middleware level (close to current networking technologies to be directly implementable, but sufficiently expressive to support service oriented applications), (ii) the service description level (favouring more abstract formalisation of basic concepts such as service definition, invocation, instantiation, and communication), and (iii) the service composition level (with 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) [28]: it is based on a flexible and dynamical reconfigurable network of components communicating via the publish-subscribe message delivery paradigm. Sessions and message correlation are supported through a type system [29]. This calculus revealed easily implementable (in terms of a Java library) as well as expressive enough to support a high-level graphical programming environment.

At the service composition level we find, e.g., λreq [3] and *concurrent constraint pi-calculus* (cc-pi) [16]. The former has been exploited to support the development of techniques for the analysis of service compositions (such as statical 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, which can roughly be divided in two families: correlation-based and session-based.

The first group comprises COWS [42] (based on message-passing and stateless components) and SOCK [17] (based on shared data spaces and stateful components). The former can be seen as an extension of the pi-calculus with correlation-based communication mechanism and primitives for activity cancellation and preservation, while the latter is closer to WS standards like BPEL and it includes an explicit modelling of processes obtained as service instantiations, process memory, etc..

The second group comprises the so-called SCC-family of calculi [7, 39, 20, 8, 12], 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* [39], *context-sensitive message passing* [20], *locations* and *dynamic multiparty sessions* [12], and *pipelines* [8].

While the above calculi are closer to the orchestration perspective, the *global calculus* [21] 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). In [6] multiparty sessions are considered, 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.

It is important to remark that communication mechanisms are somehow orthogonal to sessions. In fact, while CCS-like communication [46] is the obvious choice when only two-party sessions are considered, in the presence of multiparty sessions a more natural and more sophisticated alternative would be some variant of multicast (like broadcast [27] or CSP-like interaction [33], or even some combination of different policies [11]).

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 [34, 35, 21, 26, 30, 6, 36, 39, 1, 44, 15]. In this sense, it is interesting to relate behavioural types and the language independent approach based on contracts [9, 22] along the ideas in [40]. More generally, there are some interesting analogies between the way in which behavioural types resemble orchestration mechanisms and contracts resemble choreography descriptions.

1.4 Structure of the paper

Section 2 gives some background on the basics mathematical ingredients of process calculi, like labelled transition systems, operational semantics, structural congruence, reduction systems, bisimilarity equivalences. We illustrate such concepts by simple and detailed presentation of the main sources of inspiration for CaSPiS. Step by step, we go from the basic interaction primitives of CCS, to the more advanced name handling features of the pi-calculus, to the use of explicit sessions and to the orchestration primitives of Orc. Section 3 introduces the main principles of CaSPiS, its syntax and reduction semantics and some modelling examples. Section 4 relates CaSPiS with other well-known formalisms by presenting several intuitive encoding. Some concluding remarks are in Section 5

2 Setting the context on interactive and orchestrated systems

2.1 CCS, labelled transition systems and SOS rules

An elementary *action* of a system represents the atomic (i.e., that cannot be interrupted at the given level of granularity) abstract step of a computation that is performed by a system to move from one state to the other.

Ordinary computational models like Turing machines, register machines, several kinds of automata, the lambda-calculus and many imperative programming languages all rely on basic activities like reading from or writing on some kind of (passive) storage device or invoking a procedure with actual parameters.

Milner's *Calculus of Communicating Systems* [46] (CCS) introduced a model whose basic activities rely on some sort of handshake between two active, autonomous *processes*. Hence, in the case of concurrent systems, actions represent activities such as sending a message and receiving a message, exposing some alternatives and picking one alternative, producing a resource and consuming a resource, etc. On the one hand, when studying one process in separation from the others it is important to observe the kind of handshake it is willing to perform with other processes. On the other hand, when an handshake is performed between two entities, it constitutes a special *silent* action that has no further interaction capability.

To convince yourself about the ease of CCS in modelling concurrent systems and communication protocols, try writing down the solution to the puzzle below, adapted from [54], using first your favourite formalism, and then, after having learnt CCS basics, using CCS processes for modelling the various interacting entities (the light, the special room, and the strategies followed by humans). We shall show later some bits of the solution for the case where the light is initially on.

Exercise 1. 50 young, bright computer scientists are kept in Bertinoro until all exams will be completed, each locked in her/his own room. Their chance to be released is as follows: from time to time, one of them will be carried in a special room (in no particular order, possibly multiple times consecutively, but with a fair schedule to avoid infinite wait) and then back to her/his room. The special room is completely empty except for a switch that can turn the light either on or off (the light is not visible from outside and cannot be broken). At any time, if one of them truthfully asserts that all of them have already entered the special room at least once, then they all pass the exam and are released, but if she/he is wrong, then the chance ends and they will never pass the exam. Before the challenge starts, they have the possibility to discuss together some "protocol" to follow. Can you find a winning strategy when the initial state of the light in the room is known? And if it is not?

In CCS, we assume given a set A of activities, ranged by a, and let $\overline{A} \triangleq \{ \overline{a} \mid a \in A \}$ be the set of co-activities (disjoint from A), with $\overline{\overline{a}} = a$. The set of CCS *labels* is $\mathcal{L} \triangleq A \cup \overline{A}$, ranged by λ, and the set of CCS *actions* is $\mathrm{Act} \triangleq \mathcal{L} \cup \{ \tau \}$, ranged by α, where τ is the special *silent action*. Then, a CCS *processes* P is composed via a number of primitives, that we sketch below in an incremental way. Though the syntax may slightly vary in the literature, we let CCS processes generated by the grammar:

$$P \quad ::= \quad \textstyle\sum_{i \in I} \alpha_i.P_i \quad | \quad P[\phi] \quad | \quad P_1 | P_2 \quad | \quad (\nu a)P \quad | \quad X \quad | \quad \mathrm{rec}\, X.\, P$$

The meaning of each such process is given by a suitable *Labelled Transition System* (LTS) defined by structural induction on the syntax of the process, following Plotkin's *Structural Operational Semantics* (SOS) scheme [51–53].

We recall that an LTS $T = (S, L, \rightarrow)$ consists of: a set S of *states*, a set L of *labels*, and a *transition relation* $\rightarrow \subseteq S \times L \times S$. Sometimes a distinguished initial state $s_0 \in S$ is also considered. As usual, we shall write $s \xrightarrow{\lambda} s'$ instead of $(s, \lambda, s') \in \rightarrow$, with the meaning that there is a transition leading from state s to state s' and exposing label λ. The label gives some abstract information about the nature of the evolution. For a given label λ we denote by $\xrightarrow{\lambda}$ the binary relation $\{ (s, s') \mid s \xrightarrow{\lambda} s' \} \subseteq S \times S$.

Formally, in the case of CCS, the states of the LTS are CCS processes, the set of labels is Act, and the transition relation is the least one satisfying all SOS inference rules. When a particular process P is considered, then the initial state s_0 of its LTS is P itself and the LTS can be restricted just to the states reachable from P (after any number of transitions). The elegance of SOS relies on the fact that few inference rules define the LTS of any process that can ever be specified. Moreover, SOS rules allow for proofs by structural induction, where the interaction of complex systems is defined in terms of (the behaviour of) their components and proofs by rule induction, where a property can be proved to hold true for the whole LTS if whenever it holds for the premises of each rule, it holds also for the conclusions.

The simplest process is the *inactive* process, written **0** and called *nil*: it is not capable of performing any action. Trailing **0**s are often omitted. No inference rule is needed for **0**. *Action prefix*, written $\alpha.P$, prefixes a process P by an action α: the process $\alpha.P$ can perform α and then behave as P. The inference rule for action prefix is the axiom:

$$(\quad)\ \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

Non-deterministic choice, written $P_1 + P_2$, composes two processes in mutual exclusion: process $P_1 + P_2$ can behave as either P_1 or P_2. The inference rules for choice are:

$$(\quad)\ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'} \qquad (\quad)\ \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 + P_2 \xrightarrow{\alpha} P_2'}$$

Sometimes *guarded summation* $\sum_{i\in I} \alpha_i.P_i$ is preferred to choice, prefix (single sum) and nil (empty sum). The corresponding inference rule is:

$$(\quad)\ \frac{j \in I}{\sum_{i\in I} \alpha_i.P_i \xrightarrow{\alpha_j} P_j}$$

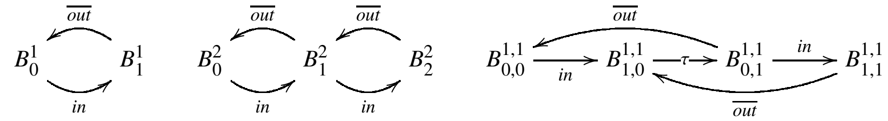
Renaming, written $P[\phi]$, renames any action α performed by P to $\phi(\alpha)$, where $\phi : \mathrm{Act} \to \mathrm{Act}$ is any renaming function such that $\phi(\overline{\lambda}) = \overline{\phi(\lambda)}$ and $\phi(\tau) = \tau$. The corresponding inference rule is:

$$(\quad)\ \frac{P \xrightarrow{a} P'}{P[\phi] \xrightarrow{\phi(a)} P'[\phi]}$$

Parallel composition, written $P_1 \,|\, P_2$, composes two processes in parallel: P_1 and P_2 evolve autonomously by interleaving their actions, but with the possibility to handshake on complementary actions, in which case $P_1 \,|\, P_2$ performs a τ action. The corresponding inference rules are:

$$(\quad)\ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \,|\, P_2 \xrightarrow{\alpha} P_1' \,|\, P_2} \qquad (\quad)\ \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 \,|\, P_2 \xrightarrow{\alpha} P_1 \,|\, P_2'} \qquad (\quad)\ \frac{P_1 \xrightarrow{\lambda} P_1' \quad P_2 \xrightarrow{\overline{\lambda}} P_2'}{P_1 | P_2 \xrightarrow{\tau} P_1' \,|\, P_2'}$$

Restriction, usually written $P\backslash a$, but here written in pi-calculus style as $(\nu a)P$, restricts the scope of activity a to process P: the process $(\nu a)P$ is allowed to perform neither action a nor $\overline{a}$; however, if P comprises two parallel processes P_1 and P_2 that can perform a and $\overline{a}$, respectively, then they can still handshake on a "under" the restriction. As usual, we abbreviate $(\nu a_1)(\nu a_2)P$ by $(\nu a_1, a_2)P$ (and similarly for three or more consecutive restrictions). The corresponding inference rule is:



(a) LTS for B^1_0 (b) LTS for B^2_0 (c) LTS for $B^{1,1}_{0,0}$

Fig. 1. Labelled transition systems associated with some simple buffers

$$(\quad)\ \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \{\, a, \overline{a} \,\}}{(\nu a)P \xrightarrow{\alpha} (\nu a)P'}$$

It should be obvious that the above operators can define only finite behaviours. There are several ways to introduce some form of interation and recursion. *Replication*, written $!P$ or also $*P$ accounts for making an unlimited number of copies of P available. Sometimes it is restricted to some guarded form, like $!\alpha.P$ or $!\sum_{i\in I} \alpha_i.P_i$. The usual inference rule for replication is:

$$(\quad)\ \frac{P\,|\,!P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

However, ( ) has a couple of drawbacks: 1) it makes the transition relation not *image-finite*, i.e. there are processes P that can reach infinitely many syntactically different processes by performing the same action α, 2) it disallows proofs by structural induction, which is maybe a minor issue. If needed, rule ( ) can be safely replaced by the following two rules, that account for the possibility of one copy of P to evolve alone, or for two copies of P to handshake:

$$(\quad 1)\ \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'\,|\,!P} \qquad (\quad 2)\ \frac{P \xrightarrow{\lambda} P_1 \quad P \xrightarrow{\overline{\lambda}} P_2}{!P \xrightarrow{\alpha} P_1\,|\,P_2\,|\,!P}$$

A more flexible alternative to replication is given by the *recursion operator* $\mathrm{rec}\,X.\,P$, where X can appear as a constant in P. The corresponding rule is:

$$(\quad)\ \frac{P\{^{\mathrm{rec}\,X.\,P}/_X\} \xrightarrow{\alpha} P'}{\mathrm{rec}\,X.\,P \xrightarrow{\alpha} P'}$$

where $\{^t/_x\}$ stands for the substitution of x by t. Alternatively, one can assume a set of *mutually recursive definitions* $\varDelta = \{\, A_i \triangleq P_i \,\}_i$ is available, that defines suitable constants A_i. The corresponding rule is:

$$(\quad)\ \frac{A_i \triangleq P_i \in \varDelta \quad P_i \xrightarrow{\alpha} P'}{A_i \xrightarrow{\alpha} P'}$$

To acquire some confidence with the notation, let us consider a classical and simple example from CCS textbooks, namely the modelling of buffers with limited capacities.

$$
(\quad)\dfrac{(\quad)\dfrac{(\quad)\dfrac{(\quad)\dfrac{(\quad)\dfrac{}{\overline{out}.B^1_0 \xrightarrow{\overline{out}} B^1_0}}{B^1_1 \xrightarrow{\overline{out}} B^1_0}}{B^1_1[\phi_1] \xrightarrow{\overline{a}} B^1_0[\phi_1]} \qquad (\quad)\dfrac{(\quad)\dfrac{(\quad)\dfrac{}{in.B^1_1 \xrightarrow{in} B^1_1}}{B^1_0 \xrightarrow{in} B^1_1}}{B^1_0[\phi_2] \xrightarrow{a} B^1_1[\phi_2]}}{B^1_1[\phi_1]\,|\,B^1_0[\phi_2] \xrightarrow{\tau} B^1_0[\phi_1]\,|\,B^1_1[\phi_2]}}{(\nu a)\Big(\,B^1_1[\phi_1]\,|\,B^1_0[\phi_2]\,\Big) \xrightarrow{\tau} (\nu a)\Big(\,B^1_0[\phi_1]\,|\,B^1_1[\phi_2]\,\Big)}
$$

Fig. 2. Proof of transition $B^{1,1}_{1,0} \xrightarrow{\tau} B^{1,1}_{0,1}$

Example 1. A process B^n modelling an initially empty buffer of capacity n can be defined by letting:

$$
\begin{aligned}
B^n_0 &\triangleq in.B^n_1 \\
B^n_i &\triangleq in.B^n_{i+1} + \overline{out}.B^n_{i-1} \qquad (0 < i < n) \\
B^n_n &\triangleq \overline{out}.B^n_{n-1}
\end{aligned}
$$

taking $B^n \triangleq B^n_0$. The LTS for B^1 is in Fig. 1(a) and for B^2 in Fig. 1(b). A process P put in parallel with B^n can handshake by performing actions $\overline{in}$ and out. If renaming ϕ_1 maps out to a and renaming ϕ_2 maps in to a, then two buffers B^1_0 could be composed in series by writing the process $(\nu a)\Big(\,B^1[\phi_1]\,|\,B^1[\phi_2]\,\Big)$. The corresponding LTS is illustrated in Fig. 1(c), where we write $B^{n,k}_{i,j} = (\nu a)\Big(\,B^n_i[\phi_1]\,|\,B^k_j[\phi_2]\,\Big)$ for brevity. Figure 2 shows the proof of transition $B^{1,1}_{1,0} \xrightarrow{\tau} B^{1,1}_{0,1}$.

Exercise 2. Draw the LTS for the processes $B^1\,|\,B^1$ and $B^{2,2}_{0,0}$.

Coming back to the puzzle from Exercise 1, the light could be modelled as a buffer of capacity one, where action *in* corresponds to “switch the light off” (it is initially on) and action *out* to “switch the light off”. Then the scientists could agree to use the light as a counter: 49 of them will switch the light on only the first time they enter the room and find it off, while one distinguished scientist will count the number of times that she/he finds the light on (and will switch it off). Since the light is initially on, the count can start only after the distinguished scientist has switched the light off for the first time. A first solution is therefore:

$$
\begin{aligned}
\mathit{Bertinoro} &\triangleq (\nu\, \mathit{swOff}, \mathit{swOn})(\,\mathit{LightON}\,|\,C_0\,|\,S\,|\cdots|\,S\,) \\
\mathit{LightON} &\triangleq \mathit{swOff}.\mathit{LightOFF} \\
\mathit{LightOFF} &\triangleq \mathit{swOn}.\mathit{LightON} \\
C_i &\triangleq \overline{\mathit{swOff}}.C_{i+1} + \overline{\mathit{swOn}}.\overline{\mathit{swOff}}.C_i \quad (0 \le i < 50) \\
C_{50} &\triangleq \overline{\mathit{freeAll}}.\mathbf{0} \\
S &\triangleq \overline{\mathit{swOn}}.\mathbf{0} + \overline{\mathit{swOff}}.\overline{\mathit{swOn}}.S
\end{aligned}
$$

where C_0 models the counting scientist and S any other scientist. Note that a scientist can wish to perform two consecutive interactions with the light just to leave its state unchanged. Unfortunately, this way there is no guarantee that consecutive interactions like $\overline{swOff}.\overline{swOn}$ are executed atomically, therefore it is better to modify the protocols in order to constrain the scientists to access the light in mutual exclusion. This can be done by modelling the special room as a one-capacity buffer, where action *in* corresponds to "enter the room" and action *out* to "leave the room": only after the room has been entered it is possible to interact with the light. To make the model more faithful, we also introduce the process for representing a "waiting scientist", i.e. a scientist who does not need to interact any more with the light but can keep entering and leaving the room.

$$
\begin{aligned}
Bertinoro &\triangleq (\nu\, in, out, swOff, swOn)(\, Room | LightON | C_0 | S \mid \cdots \mid S\ )\\
Room &\triangleq B^1\\
LightON &\triangleq swOff.LightOFF\\
LightOFF &\triangleq swOn.LightON\\
C_i &\triangleq \overline{in}.\left(\overline{swOff}.out.C_{i+1} + \overline{swOn}.\overline{swOff}.out.C_i\right) \quad (0 \leq i < 50)\\
C_{50} &\triangleq \overline{freeAll}.\mathbf{0}\\
S &\triangleq \overline{in}.\left(\overline{swOn}.out.WS + \overline{swOff}.\overline{swOn}.out.S\,\right)\\
WS &\triangleq \overline{in}.\tau.out.WS
\end{aligned}
$$

We leave to the reader finding a solution for the case where the initial state of the light is not known in advance, e.g. when the light is modelled as the process $\tau.LightON + \tau.LightOFF$.

A vast literature on CCS has established different criteria for when two processes should be considered as "equivalent". Without entering into the details, we mention two of the most widely used notion of equivalence, namely *strong bisimilarity* and *weak bisimilarity*. Contrary to trace equivalence, bisimilarities can take into account the branching structure of the transition systems, i.e. the points where choices are made. We refer the interested reader to [31, 32] for a wider range of options.

Definition 1 (Strong Bisimilarity). *A binary relation $\mathcal{R}$ over processes is a* strong bisimulation *iff whenever $(P,Q) \in \mathcal{R}$ then for each $\alpha \in$ Act:*

- *if $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha} Q'$ for some Q' such that $(P',Q') \in \mathcal{R}$*
- *if $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha} P'$ for some P' such that $(P',Q') \in \mathcal{R}$.*

Two processes P and Q are strongly bisimilar*, written $P \sim Q$, iff there exists a strong bisimulation $\mathcal{R}$ such that $(P,Q) \in \mathcal{R}$, i.e. $\sim \triangleq \bigcup\{\, \mathcal{R} \mid \mathcal{R}$ is a strong bisimulation $\}$.*

Strong bisimilarity is an equivalence relation and it is a congruence w.r.t. all CCS operators, meaning that if we replace any subterm $P'$of P with a strongly bisimilar term Q' then the result is guaranteed to be strongly bisimilar to P. Notably, $\sim$ admits a logical characterisation in terms of Hennessy-Milner logic, a modal logic of actions for the analysis and verification of reactive systems [2].

Exercise 3. Prove that $B^2 \sim B^1 \,|\, B^1$ and $B^2 \nsim B^{1,1}_{0,0}$.

Strong bisimilarity is coarser than LTS isomorphism, but it still distinguishes too many processes that have essentially the same behaviour. In particular, it is often the case that some additional silent transitions may arise or not depending on different attitudes to modelling the same system. Weak bisimilarity, denoted by $\approx$, relaxes the notion of strong equivalence by allowing to simulate a move also performing additional silent transitions beforehand and afterwards: roughly, letting $\rightarrow^*$ denote the reflexive and transitive closure of $\xrightarrow{\tau}$ (i.e. $\rightarrow^*$ is the relation such that $P \rightarrow^* P'$ iff P' is reachable from P via any number of consecutive silent transitions, possibly none), in the weak case a step $P \xrightarrow{\lambda} P'$ can be simulated via a sequence of steps $Q \rightarrow^* \xrightarrow{\lambda} \rightarrow^* Q'$ and silent transitions $P \xrightarrow{\tau} P'$ can be simulated via a possible empty sequence of silent steps $Q \rightarrow^* Q'$. For example, $B^2 \approx B^{1,1}_{0,0}$. Weak bisimilarity is an equivalence relation that includes strong bisimilarity (in the sense that $P \sim Q$ implies $P \approx Q$ for any processes P and Q), but it is not a congruence (because it is not preserved by the choice operator).

2.2 Pi-calculus, structural congruence and reduction semantics

CCS is Turing powerful, and it can be used at several level of analysis, as a specification language, as a programming language, as a description language, as a type language, etc. However, when one wants to model interactive systems with dynamic changes in connectivity, or networks where processes can move between physical or virtual locations, then the representation distance is quite increased and the modelling activity can become cumbersome.

Milner, Parrow and Walker's *Calculus of Mobile Processes* [49, 48, 56] (i.e., the π-calculus or also pi-calculus) introduces a key ingredient: the possibility to communicate channel names. This way, a process can acquire new communication links, pass its own private channels to other processes, create fresh channels, and much more. Even if the required extension to CCS syntax is to some extent minimal, it opened a still flourishing research thread. Nowadays, there are many variants of π-calculus (monadic, polyadic, synchronous, asynchronous, with mixed choice, higher-order, to name a few) each with a consolidated theory on its own. To appreciate the key difference w.r.t. CCS, let us consider the following puzzle, adapted from [24].

Exercise 4. 100 young, bright computer scientists are kept awake in Bertinoro until all exams will be completed. Their chance to have some sleep is as follows: first each of them is assigned a different id from 1 to 100 and a different room (assume rooms are also numbered from 1 to 100); then the ids are randomly distributed one per room; each scientist is given the possibility to open 50 rooms of her/his choice and look at the ids contained there; if all scientists are able to find their own id, then they are all given access to the rooms, otherwise (even if only one of them is not able to find her/his own id) they will not be able to sleep until all exams have been given. Each scientist is not allowed to look at the ids found in the rooms by her/his colleagues, and they are not allowed to speak to each other once the procedure is started. Before the challenge starts, they have the possibility to discuss together some "protocol" to follow. Can you find an optimal strategy to let them have some sleep with highest probability?

The best possible strategy leaves almost 1/3 of probability to get some sleep. It is based on a simple protocol, equal for all participants: the first room opened by scientist with id i must be room i; at each stage, if an id k different from her/his own is found in a room, then the next room to be opened is room k. The idea is that rooms and the id they contain define a set of permutation cycles: the strategy is "winning" iff all such cycles have length less than or equal to 50. As there can be at most one cycle of length greater than 50, the probability to win coincides with the probability that such a long cycle is not present. If modelled in CCS, the protocol should consider 100 different continuations for each room, one for each possible id contained therein. Using pi-calculus instead, the next room to open can be just communicated.

From the point of view of the syntax, the only primitives to be changed are action prefixes. In the following we assume an infinite set of names $\mathcal{N}$, ranged by x, y, z, is available. The action prefixes of the pi-calculus represents either the sending $\overline{x}\langle y\rangle$ of a name y along x, or the receiving $x(y)$ of a name y along x, or the silent action τ. Sometimes the matching prefixes $[x = y]$ and mismatching $[x \neq y]$ are also considered: they represent ordinary test for equality and inequality of names, and can be used to follow different alternatives depending on the received names. The use of mismatch prefixes is discouraged because their presence can violate useful monotonicity properties of processes, like the fact that name-substitution does not decrease action capabilities of a process.

Unfortunately, the inference rules of CCS cannot be smoothly extended to pi-calculus and some additional care and machinery is needed. To see why, consider the straight extensions of inference rules for action prefixes, where in the case of input one simply guesses the name z that will be received:

$$(\quad)\ \frac{}{x(y).P \xrightarrow{xz} P\{^{z}/_{y}\}} \qquad\qquad (\quad)\ \frac{}{\overline{x}\langle z\rangle.P \xrightarrow{\overline{xz}} P}$$

Now we should decide which actions should be forbidden under restriction. Take the process $(\nu n)P$ and suppose $P \xrightarrow{xz} P'$:

- if $n \notin \{\, x, z \,\}$ then we can let $(\nu n)P \xrightarrow{xz} (\nu n)P'$;
- if $n = x$ then we must forbid the move;
- if $n = z \neq x$ then we must forbid the move, because n is a private name that cannot be received from the outside.

Now suppose $P \xrightarrow{\overline{xz}} P'$:

- if $n \notin \{\, x, z \,\}$ then we can let $(\nu n)P \xrightarrow{\overline{xz}} (\nu n)P'$;
- if $n = x$ then we must forbid the move;
- if $n = z \neq x$ then what? If we forbid the move, then private names cannot be extruded to other processes, which would be a severe limitation. If we allow the move, then we would like to extrude the scope of n only to the processes that handshake on $\overline{xn}$, hence we should have $(\nu n)P \xrightarrow{\overline{xz}} P'$, where the restriction disappears from the target. On the other hand, when handshake is accomplished, we would like to restore the restriction.

$$
\begin{array}{llll}
S+\mathbf{0}\equiv S & S_1+S_2\equiv S_2+S_1 & S_1+(S_2+S_3)\equiv(S_1+S_2)+S_3 & [a=a]\pi.P\equiv\pi.P \\
P|\mathbf{0}\equiv P & P_1\,|\,P_2\equiv P_2\,|\,P_1 & P_1\,|(P_2\,|P_3)\equiv(P_1\,|P_2)|P_3 & P\equiv P\,|\,!P \\
(\nu a)\mathbf{0}\equiv\mathbf{0} & (\nu a)(\nu b)P\equiv(\nu b)(\nu a)P & P\,|\,(\nu a)Q\equiv(\nu a)(P\,|\,Q)\ \text{ if } a\notin\mathrm{fn}(P) &
\end{array}
$$

Fig. 3. Structural congruence laws for the pi-calculus

The so-called *early operational semantics* solves the problem by introducing different labels for the free output $\overline{x}z$ and the bound output $\overline{x}(z)$ (where the name z is extruded). This in turn have several consequences on the rules for parallel composition: some side conditions are needed in order to avoid that an extruded name captures a free name of a process running in parallel, and two kinds of handshakes are possible, depending on the kind of output that is considered: the handshake between actions xz and $\overline{x}z$ is the ordinary one (as in CCS); the handshake between actions xz and $\overline{x}(z)$ move the restriction (νz) on top of the parallel composition. In general, it emerges the necessity to take into account which are the free names of a process (denoted by $\mathrm{fn}(P)$) and which are its bound names (denoted by $\mathrm{bn}(P)$). In the case of pi-calculus, the only binders are input prefix and restriction, i.e. in both $x(y).P$ and $(\nu y)P$ the name y is bound and its scope is restricted to P. There are further consequences also on the definition of strong bisimilarity, when the actions to be simulated depend on the free names of a process. These caveats make the formal presentation of pi-calculus semantics more complicated and less intuitive than CCS one, when encountered for the first time.

For the above reasons, a different style of presentation is sometimes preferred for pi-calculus (and for many other calculi with name-handling features). It has two main ingredients: a *structural congruence* relation, used to write processes in some canonical form, easier to manipulate; a *reduction* relation that represents only completed interactions, roughly the τ moves.

Let us consider the following syntax for pi-calculus processes:

$$
\begin{array}{rcclclclcl}
\text{(Processes)} & P & ::= & S & | & P_1\,|\,P_2 & | & (\nu x)P & | & !P \\
\text{(Sums)} & S & ::= & \mathbf{0} & | & \pi.P & | & S_1+S_2 & & \\
\text{(Prefixes)} & \pi & ::= & \overline{x}\langle y\rangle & | & x(y) & | & \tau & | & [x=y]\pi
\end{array}
$$

The structural congruence $\equiv$ of pi-calculus is the least congruence relation that satisfies the equalities in Fig. 3 plus alpha-conversion of bound names.[1] The structural congruence allows one to rearrange the syntax of processes so that any two possible interacting entities can be put side by side (in parallel composition). Note in particular that: the order in which we compose processes in sums should not matter; the order in which we compose processes in parallel should not matter; the order in which we restrict names should not matter. Moreover, the scope extrusion law (the rightmost equality in the bottom row of Fig. 3) can broaden the scope of a restricted name before it is communicated. It is not difficult to see that each π-calculus process P can be put in

[1] The laws for alpha-conversion allow for the arbitrary renaming of bound names, but avoiding clashes with free names. In the case of pi-calculus, alpha-conversion means that for any process P and any names x,y,z with $z\notin\mathrm{fn}(P)$ we have $x(y).P\equiv x(z).(P\{^z/_y\})$ and $(\nu y).P\equiv(\nu z)(P\{^z/_y\})$.

a canonical form like $P \equiv (\nu x_1)...(\nu x_k)(S_1|...|S_n|!P_1|...|!P_m)$ for some suitable names $x_1,...,x_k$, sums $S_1,...,S_n$, and processes $P_1,...,P_m$ in canonical forms. Thus, all interactions can now be expressed by considering only a small number of reductions over canonical forms. Essentially there are two rules for basic reductions:

$$(\mathrm{R}\quad)\ \frac{}{\tau.P+S \xrightarrow{\tau} P} \qquad (\mathrm{R}\quad)\ \frac{}{(x(y).P_1+S_1)|(\overline{x}\langle z\rangle.P_2+S_2) \xrightarrow{\tau} P_1\{^z/_y\}|P_2}$$

plus two rules for the so-called reactive contexts (restriction and parallel composition):

$$(\mathrm{R}\quad)\ \frac{P_1 \xrightarrow{\tau} P_1'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2} \qquad (\mathrm{R}\quad)\ \frac{P \xrightarrow{\tau} P'}{(\nu x)P \xrightarrow{\tau} (\nu x)P'}$$

together with the (often implicit) rule for structural congruence:

$$(\mathrm{R}\quad)\ \frac{P \equiv Q \quad Q \xrightarrow{\tau} Q' \quad Q' \equiv P'}{P \xrightarrow{\tau} P'}$$

The version we have presented is the so-called (synchronous) monadic pi-calculus. In the polyadic case, messages can consist of (possibly empty) name tuples $\vec{y}$ and communication requires the number of transmitted values to match exactly the number of received ones. In the case of empty tuples, input and output prefixes $x()$ and $\overline{x}\langle\rangle$ are sometimes written more concisely in CCS-like notation as x and $\overline{x}$.

Coming back to the puzzle (Exercise 4), we consider names x_1,..., x_{100}, and model the fact that room k contains id n by writing the process $R_k \triangleq !\overline{x_k}\langle x_n\rangle$. Then the strategy S_i of player i is defined as follows, where we use parametrised constants for simplicity:

$$\begin{aligned}
S_i &\triangleq S_i^{50}(x_i) \quad (1 \le i \le 100)\\
S_i^t(x) &\triangleq x(y).\Big([y = x_i]\overline{ok_i}\,|\,S_i^{t-1}(y)\Big) \quad (1 \le i \le 100, 1 < t \le 50)\\
S_i^1(x) &\triangleq x(y).[y = x_i]\overline{ok_i} \quad (1 \le i \le 100)
\end{aligned}$$

The guard for success can be written as $G \triangleq ok_1.ok_2\ldots ok_{100}.\overline{ok}$, and the whole system as $(\nu x_1,...,x_{100},ok_1,...,ok_{100})(S_1|\cdots|S_{100}|R_1|\cdots|R_{100}|G)$: after finitely many silent transitions the system will either be ready to handshake on ok, or it will deadlock because some scientists have not been able to find their ids.

From the operational point of view, reduction semantics can be tightly reconciled with LTS semantics, by the so-called *Harmony Lemma* [56]. From the observational point of view, the situation is more complicated, because reduction semantics provides no meaningful "observables". Then, a meaningful abstract equivalence can be defined in terms of some simple predicates, called *barbs*, which express the capabilities to emit and receive on a given channel (but notably, neither the sent / received message nor the target state are observed). By combining barbs and reductions (and closing under contexts), we obtain *strong* and *weak barbed congruences* that can be shown to coincide with the analogous congruences originated from the LTS semantics.

$$\llbracket x \rrbracket_a^{\lambda 2\pi} \triangleq \overline{x}\langle a\rangle$$
$$\llbracket \lambda x.M \rrbracket_a^{\lambda 2\pi} \triangleq a(x,m).\llbracket M \rrbracket_m^{\lambda 2\pi}$$
$$\llbracket M\,N \rrbracket_a^{\lambda 2\pi} \triangleq (\nu m)\Big(\llbracket M \rrbracket_m^{\lambda 2\pi} \,|\, (\nu b)(\, !b(n).\llbracket N \rrbracket_n^{\lambda 2\pi} \,|\, \overline{m}\langle b,a\rangle \,) \Big) \quad \text{for } b \notin \mathrm{fv}(N)$$

Fig. 4. Encoding of λ-calculus in pi-calculus

Example 2. We now provide evidence of the expressiveness of pi-calculus by showing that functional programming can be recovered as a special flavour of interactive system. This is done by encoding λ-calculus in pi-calculus [47].

We recall that *λ-expressions* $M, N, \ldots$ can be either a variable x, the *λ-abstraction* $\lambda x.M$ or the *application* $M\,N$, with obvious notions of free and bound variables. The reduction rules (for the lazy semantics) are:

$$(\beta)\frac{}{(\lambda x.M)N \to M\{^{N}/_{x}\}} \qquad (\mu)\frac{M \to M'}{M\,N \to M'\,N}$$

Roughly, processes can represent both "functions" and "arguments" which are composed in parallel and interact to β-reduce. However, they interact by transmitting *access points* to terms instead of terms themselves. The pi-calculus process that encodes a λ-expression M is parametric w.r.t. the access point a for retrieving arguments, and it is written $\llbracket M \rrbracket_a^{\lambda 2\pi}$. The encoding is shown in Fig. 4.

In the case of a variable x, the corresponding processes sends the access point a to the function available at x. In the case of a λ-abstraction, the process is waiting to receive on a the argument x and the name of a further access point m for further arguments, needed when evaluating M. In the case of application, a fresh server b is installed for handling the requests to the argument N, together with a new access point m for the argument of M, to which b and a are sent. The correspondence can be formalised by showing that $\llbracket (\lambda x.M)N \rrbracket^{\lambda 2\pi}$ and $\llbracket M\{^{x}/_{N}\} \rrbracket^{\lambda 2\pi}$ are related by the weak equivalence $\approx$.

Exercise 5. Write the pi-calculus processes $\llbracket \lambda x.x \rrbracket_a^{\lambda 2\pi}$ and $\llbracket (\lambda x.x)N \rrbracket_a^{\lambda 2\pi}$. Then shows that $\llbracket (\lambda x.x)N \rrbracket_a^{\lambda 2\pi}$ reduces after some steps to $\llbracket N \rrbracket_a^{\lambda 2\pi} \,|\, (\nu b)!b(n).\llbracket N \rrbracket_n^{\lambda 2\pi}$ (which can be shown strongly bisimilar to $\llbracket N \rrbracket_a^{\lambda 2\pi}$, as $(\nu b)!b(n).P$ is clearly inert).

It is evident that pi-calculus provides a rather sophisticated framework for the study of interaction. For example, Sangiorgi proved in his PhD thesis [55] that name mobility can encode some sort of higher-order communication, in the sense of process mobility. Nevertheless, the considered primitives and the overall framework are rather low-level. As a main consequence, when the size of processes increases, it becomes harder to acquire confidence in the correctness of the modelling. This fact opened a major research strand on type systems for process calculi, where different kinds of types and annotations are devised to offer static guarantees about the validity of certain properties, ranging from the absence of communication errors (e.g. receiving a message of mismatched arity or type) to termination and deadlock avoidance. In particular, there is now a renewed interest in the area of service oriented calculi around the notion of *session types*, introduced about a decade ago by Honda, Kubo and Vasconcelos [35].

2.3 A session calculus

One of the problem with pi-calculus is that names are used to encode many different behavioural aspects all in terms of communication. In principle, one should at least distinguish between two different usages: the first one is concerned with some sort of static sorting discipline, like establishing that all names transmitted on x must be integers, or that all names transmitted on y must be names of channels where integers can be sent, or that z can only be used for input; the second one has to do with dynamic prescriptions, like protocol narrations for the peers of a session, establishing e.g. that on channel z must first be sent an integer, then be received a name of a channel where integers can be sent and finally be received another integer.

The idea in [35] is to structure the language so to guarantee that, at any time, each session-like channel is shared between two peers only. This way, the protocol run on one side can be more conveniently checked for compatibility w.r.t. the protocol on the other side, as the two must be related by some form of duality. The key idea is to impose a symmetric form of communication for opening a session.

Example 3. To see how it works, consider the specification of a server that repeatedly receives values, computes some heavy scientific calculation f on it and then returns the result to the caller. If it is written as $S \triangleq !in(x).\overline{out}\langle f(x)\rangle$, then there is a big problem: if two or more clients are around, they could intercept the result of other calculations. For example:

$$
\begin{aligned}
&S\,|\,\overline{in}\langle 1\rangle.out(y_1).P_1\,|\,\overline{in}\langle 2\rangle.out(y_2).P_2\\
&\qquad\xrightarrow{\tau} S\,|\,\overline{out}\langle f(1)\rangle\,|\,out(y_1).P_1\,|\,\overline{in}\langle 2\rangle.out(y_2).P_2\\
&\qquad\xrightarrow{\tau} S\,|\,\overline{out}\langle f(1)\rangle\,|\,\overline{out}\langle f(2)\rangle\,|\,out(y_1).P_1\,|\,out(y_2).P_2\\
&\qquad\xrightarrow{\tau} S\,|\,\overline{out}\langle f(1)\rangle\,|\,P_1\{^{f(2)}/_{y_1}\}\,|\,out(y_2).P_2\\
&\qquad\xrightarrow{\tau} S\,|\,P_1\{^{f(2)}/_{y_1}\}\,|\,P_2\{^{f(1)}/_{y_2}\}
\end{aligned}
$$

The typical way to solve this problem consists of receiving the result on a private channel, freshly established at the moment of the call. The server is thus written like $S \triangleq !in(x,k).\overline{k}\langle f(x)\rangle$, so that each time the result will be sent to a channel specified by the caller, which can, e.g., employ a fresh name for the goal, like in $(\nu k)\overline{in}\langle 1,k\rangle.k(y_1).P_1$. In the monadic case, k can be used both for sending the value and for receiving the result: the server would become $S \triangleq !in(k).k(x).\overline{k}\langle f(x)\rangle$ and the client would become $(\nu k)\overline{in}\langle k\rangle.\overline{k}\langle 1\rangle.k(y_1).P_1$. It can be seen that k plays the role of a session key, with dual usages on the server and on the client. When the pattern $(\nu k)\overline{in}\langle k\rangle.P$ is frequent, it can be more conveniently written as a macro $\overline{in}(k).P$, which is reminiscent of bound output and is symmetric to ordinary input.

Conceptually, the calculus proposed in [35] exposes three different communication pairs. The first consists of prefixes for session acceptance $a(k)$ and session request $\overline{a}(k)$. The corresponding reduction rule is:

$$
(\quad)\ \frac{}{a(k).P\,|\,\overline{a}(k).Q \xrightarrow{\tau} P\,|\,Q}
$$

(note that alpha-conversion is exploited to choose the same name k on both sides before applying the reduction).

Then there are ordinary input $k?(x)$ and output $k!\langle x\rangle$ on a session k (only the former is a binder for x). The corresponding reduction rule is:

$$(\quad)\ \frac{}{k?(x).P\,|\,k!\langle y\rangle.Q \xrightarrow{\tau} P\{^{y}/_{x}\}\,|\,Q}$$

Finally there are primitives for delegation of a session key to a different process: session receiving $k?((k'))$ and session sending $k!\langle\langle k'\rangle\rangle$ (only the former is a binder for k'). The corresponding reduction rule is:

$$(\quad)\ \frac{}{k?((x)).P\,|\,k!\langle\langle k'\rangle\rangle.Q \xrightarrow{\tau} P\{^{k'}/_{x}\}\,|\,Q}$$

Note that after having sent k' on k, process Q is no longer allowed to mention k'.

Sometimes a fourth communication pair is considered that involves label branching over a finite set of predefined labels $\sum_i k?l_i.P_i$ on one side and label selection $k!l.P$ on the opposite side. The corresponding reduction rule is:

$$(\quad)\ \frac{j \in I}{\sum_{i\in I} k?l_i.P_i\,|\,k!l_j.Q \xrightarrow{\tau} P_j\,|\,Q}$$

The remaining rules are the ordinary ones for parallel composition, restriction, structural congruence and recursion (considered in place of replication). Note that the same sequential process can open different sessions and interleave activities within them.

Exercise 6. One young, bright computer scientists is given the possibility to pass the exam if she is able to play chess twice against the state-of-the-art computer player available on the web, without loosing both games. She has never played chess before. Which strategy can she take?

The idea is essentially to let the computer AI play against itself. We can model the web site as follows:

$$\begin{aligned}
Chess &\triangleq start(k).\,(\,Chess\,|\,k?\mathsf{black}.B(k) + k!\mathsf{white}.W(k)\,)\\
B(k) &\triangleq \mathrm{rec}\,X.\,k?(m).k!\langle m :: next(m)\rangle.X\\
W(k) &\triangleq \mathrm{rec}\,X.\,k!\langle next(\epsilon)\rangle.k?(m).X
\end{aligned}$$

Thus, the web site let the human player choose the colour and then, depending on such choice, either it waits for the first move of the human or it starts the game by sending its first move. For simplicity we assume the game protocol consists of sending and receiving the list of moves made so far. The AI will compute its best move by exploiting some function *next* applied on the history of moves. Note that each game runs in its own session k, to avoid mixing the games of different players.

The best strategy would be to open two gaming sessions, choosing to play holding black pieces in one and white pieces in the other, and then always send on one game the latest move performed by the computer player in the other game.

$$Human \triangleq start(k_1).k_1!\mathsf{black}.start(k_2).k_2!\mathsf{white}.P(k_1,k_2)$$
$$P(k_1,k_2) \triangleq \mathrm{rec}\, X.\, k_1?(m).k_2!\langle m\rangle.k_2?(n).k_1!\langle n\rangle.X$$

The main advantage w.r.t. pi-calculus is that sophisticated typing disciplines can now be defined by exploiting the different syntactic categories and primitives. To know more on this topic see Vasconcelos's contribution in this volume [58].

2.4 Orc

Quite independently from traditional process calculi as CCS and pi-calculus, where control and data flow is always encoded in terms of interaction, Cook and Misra have proposed a basic programming language for structured orchestration, called Orc [50, 23, 37], whose primitives meet simplicity with yet great generality. Orc neatly separates orchestration from computation: Orc expressions e should be 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 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 on hidden mechanisms for name handling (creation and passing) and for atomic distributed termination. We recaps Orc basics, borrowing definitions from [50]. Apart from minor differences w.r.t. the literature, we let Orc expressions be defined by the following grammar:

$$\begin{array}{rrcl} \text{(Expressions)} & e & ::= & \mathbf{0} \quad | \quad b \quad | \quad e_1|e_2 \quad | \quad e_1 > x > e_2 \quad | \quad e_2 \ \textbf{where}\ x :\in e_1 \\ \text{(Basic expr.)} & b & ::= & \langle p\rangle \quad | \quad x\langle\vec{p}\rangle \quad | \quad s\langle\vec{p}\rangle \quad | \quad E\langle\vec{p}\rangle \quad | \quad ?k \end{array}$$

where we assume given the following (pairwise disjoint) sets: a set $\mathcal{V}$ of *values*, ranged by v, a set X of *variables*, ranged by x, a set $\mathcal{S}$ of *sites*, ranged by s, a set $\mathcal{E} \triangleq \{ E_i(\vec{x}) \triangleq e_i \}_i$ of *defined expressions*, ranged by E, and a set $\mathcal{K}$ of *invocation keys*, ranged by k. Moreover, we let the set of *parameters* $\mathcal{P} \triangleq \mathcal{V} \cup \mathcal{S} \cup X$, ranged 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 $\mathrm{fv}(e)$. All defined expressions $E(\vec{x}) \triangleq e$ are well-formed, in the sense that $\mathrm{fv}(e) \subseteq \vec{x}$.

Orc semantics is defined in the LTS style, via SOS rules. The basic expressions $?k$ must be considered as run-time syntax: they denote response handlers for site invocations. The set of labels include action $\overline{s}\langle\vec{v}\rangle@k$ for a site invocation to s with parameters $\vec{v}$ and invocation key k, action $v@k$ for the receipt of the value v in response to a site invocation handled by k, action $\langle v\rangle$ for the local publication of value v, and the silent action τ. We let o range over labels.

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, 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.

$$(\quad)\ \frac{k \text{ globally fresh}}{s\langle\vec{v}\rangle \xrightarrow{\overline{s}\langle\vec{v}\rangle @ k} ?k} \qquad (\quad)\ \frac{}{?k \xrightarrow{v@k} \langle v\rangle} \qquad (\quad)\ \frac{}{\langle v\rangle \xrightarrow{\langle v\rangle} \mathbf{0}}$$

While site call is strict, in the sense that actual parameters are evaluated before the call, the evaluation of defined expressions is non-strict. The corresponding inference rule is:

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

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:

$$(\quad)\ \frac{e_1 \xrightarrow{o} e_1'}{e_1\,|\,e_2 \xrightarrow{o} e_1'\,|\,e_2} \qquad (\quad)\ \frac{e_2 \xrightarrow{o} e_2'}{e_1\,|\,e_2 \xrightarrow{o} e_1\,|\,e_2'}$$

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:

$$(\quad)\ \frac{e_1 \xrightarrow{o} e_1' \quad o \neq \langle v\rangle}{e_1 > x > e_2 \xrightarrow{o} e_1' > x > e_2} \qquad (\quad)\ \frac{e_1 \xrightarrow{\langle v\rangle} e_1'}{e_1 > x > e_2 \xrightarrow{\langle v\rangle} (e_1' > x > e_2)\,|\,e_2\{^v/_x\}}$$

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:

$$(\quad)\ \frac{e_2 \xrightarrow{o} e_2'}{e_2 \textbf{ where } x :\in e_1 \xrightarrow{o} e_2' \textbf{ where } x :\in e_1}$$

$$(\quad)\ \frac{e_1 \xrightarrow{o} e_1' \quad o \neq \langle v\rangle}{e_2 \textbf{ where } x :\in e_1 \xrightarrow{o} e_2 \textbf{ where } x :\in e_1'} \qquad (\quad)\ \frac{e_1 \xrightarrow{\langle v\rangle} e_1'}{e_2 \textbf{ where } x :\in e_1 \xrightarrow{o} e_2\{^v/_x\}}$$

Although not evident from the operational semantics, the running implementation of Orc assumes that all concurrent invocations are executed instantaneously and that the asymmetric parallel operator picks the first value available, i.e. rules ( ) and ( ) have higher priorities than the remaining ones). We refer the interested reader to [59] for more details.

Example 4. We borrow from [50] some simple examples of Orc declarations. In the following we assume the existence of a site *timer* that receives an integer t and return a void datum $\langle\rangle$ after t units of time, 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

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

  specifies a service that notifies address a with both news selected from *cnn* and from *bbc* in a unique message.
- Declaration $\mathit{Delay}(s,d,t) \triangleq \mathit{timer}\langle t\rangle >> \langle x\rangle$ **where** $x :\in s\langle d\rangle$ specifies a service that contact site s with argument d but delays the response up to t time units, in the sense that even if the response is available before t time units then it will be published only after the timer expires, while if it is available after the timer already expired, then it is readily published. Note that the timer invocation does not depend on x and therefore it is activated concurrently with the invocation to s.
- Declaration $\mathit{UnfairPick}(s_1,s_2,d,t) \triangleq \langle x\rangle$ **where** $x :\in (s_1\langle d\rangle\,|\,\mathit{Delay}\langle s_2,d,t\rangle)$ specifies a service that contact both sites s_1 and s_2 with argument d but privileges the response from site s_1, in the sense that if it arrives before t time units then the response of s_2, if any, is ignored. For example

  $$\mathit{UnfairMail}(a,d) \triangleq \mathit{UnfairPick}\langle \mathit{cnn},\mathit{bbc},d,20\rangle > x > \mathit{email}\langle x,a\rangle$$

  specifies a service that notifies address a with only one of the news selected either from *cnn* or from *bbc*, but preferably from *cnn* (that has 20 time units of advantage).

Exercise 7. A classic problem in non-strict evaluation is the so-called *parallel-or*. Suppose there are two sites s_1 and s_2 that publish some booleans. Write an Orc expression *POR* that publishes the value *false* only if both sites return *false*, the value *true* as soon as either site returns *true*, and otherwise it never publishes a value. In the solution it can be assumed: (1) the existence of a site *ift*(b) that receives a boolean value and returns

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

$$
(e_3 \textbf{ where } y :\in e_2) \textbf{ where } x :\in e_1 \sim (e_3 \textbf{ where } x :\in e_1) \textbf{ where } y :\in e_2 \text{ if } x \notin \mathrm{fv}(e_2) \text{ and } y \notin \mathrm{fv}(e_1)
$$

$$
(e_2 > y > e_3) \textbf{ where } x :\in e_1 \sim (e_2 \textbf{ where } x :\in e_1) > y > e_3 \;\; \text{if } x \notin \mathrm{fv}(e_3)
$$

Fig. 5. Some strongly bisimilar Orc expressions

true if b is *true*, and otherwise it does not respond; (2) the existence of a site $or(b_1, b_2)$ that return the inclusive logical or of the two booleans received as arguments. Note that *POR* must publish one result at most.

The abstract semantics of Orc can be defined in terms of strong and weak bisimilarities and gives rise to interesting equivalences, some of which are in Fig. 5.

3 A Calculus of Sessions and Pipelines

In the previous section we have seen different frameworks for the modelling of interaction, sessions, orchestration and cancellation of activities, each offering elegant and flexible solutions to tackle specific issues. Starting from these premises, the objective of a coordinated effort within the EU funded project S [57] was to synthesise so-called *core calculi for SOC*, where all the above aspects are dealt with in a uniform and structured way. One of the outcomes of the project is CaSPiS [8], a *Calculus of Sessions and Pipelines*, which evolved as an improved refinement of SCC [7] (*Service Centred Calculus*). CaSPiS exploits *nested sessions* and *pipelines* as natural tools for structuring client-service communication and orchestration, respectively. Activity cancellation is built around the ability of peers to leave sessions and to program suitable handlers for such cases. We discuss below the essential guidelines around the design of CaSPiS, emphasising the differences w.r.t. the calculi in Section 2:

Interaction: Contrary to CCS and pi-calculus where the same form of communication is used for different purposes, in CaSPiS a few forms of basic interactions are distinguished and regulated on their own. For example, services are globally available and can be invoked independently from the surrounding context, while ordinary input-output communication is context sensitive and implicitly driven.

Sessions: When CaSPiS is viewed as a programmable abstraction of SOC, the idea is to relieve programmers from the burden of dealing with session keys. Therefore the choice made in CaSPiS is to handle sessions as an implicit mechanism for enclosing the communications between a caller and its callee, avoiding external interferences. Like in Section 2.3, a name-scoping mechanism is used to handle sessions, but contrary to the calculus in [35], each CaSPiS process has its own implicit current session and it is possible neither to program interleaved communications in different sessions, nor session delegation. However, sessions can be nested (e.g. when calling a service within a session, the interaction will take place in a dedicated subsession) and it is allowed to pass values from nested sessions up.

Orchestration: As in Orc, orchestration is kept separate from interaction and pipelines are seen as a convenient mechanism for modelling the flow of data between local processes: it is more general than sequential composition, better suited w.r.t. concurrency and does not require the explicit and improper use of channels for orchestration tasks. Here a more sophisticated form of pipeline is introduced, which is well-integrated with the other features of CaSPiS, not considered in Orc.

Cancellation: Orc's asymmetric parallel operator provides a convenient form of cancellation for pending activities, but whose effect is purely local: the operational semantics is designed in such a way that if a site has been invoked, but the local handler for its response is cancelled, then the response cannot show up. This is maybe fine if only one-way or request-response interactions are considered, but not in CaSPiS, where the cancellation of activities could leave some pending peers in the middle of long-running interactions. Likewise signing a contract implies some obligations, opening a session implies notifying the peer when leaving the session before its conclusion. Also inspired by some recent work on process calculi for modelling transactions [13, 14, 41, 19, 18], CaSPiS comprises a novel mechanism for ensuring the notification of any activity cancellation, for which we are not aware of any similar counterpart in the literature on process calculi.

Readability and typeability: To tackle the complexity of SOC systems, it should be possible to structure complex processes in smaller parts tailored to specific issues and it should be possible to guarantee the compliance of the whole process by checking the compliance of its parts. Moreover, type checking and type inference systems should be available that automatically detect protocol inconsistencies and communication errors. For type systems to be effective, it is important that the abstraction distance w.r.t. the concrete formalism is not too large, so that any detected type problem can be immediately explained, tracked and understood over the underlying processes. Some preliminary investigation has shown that quite simple type systems can be developed for CaSPiS that guarantee nice properties (communication safety, client progress and deadlock freedom). A prototype tool, called TypeSes[2] for type inference is also available (see [45]).

3.1 A CaSPiS walk-through

We introduce CaSPiS primitives in an incremental way. Let $\mathcal{N}_{\mathrm{srv}}$ and $\mathcal{N}_{\mathrm{sess}}$ be two disjoint countable sets, respectively of *service* names, ranged by s and of *session* names, ranged by r. We assume $\mathcal{N}_{\mathrm{srv}}$ and $\mathcal{N}_{\mathrm{sess}}$ are included in a larger set of *names* $\mathcal{N}$, ranged by n, and let $x, y, ..., u, v...$ range over $\mathcal{N} \setminus \mathcal{N}_{\mathrm{sess}}$.

In the following we shall exploit the notion of a *context*, written $\mathbb{C}[\![\cdot]\!]$, i.e. a process term with one hole $[\![\cdot]\!]$. We write $\mathbb{C}[\![P]\!]$ for the process where the hole is textually replaced by process P. The contexts we are interested in are called *static*, and 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). See Section 3.2 for the exact definition.

[2] http://www.di.unipi.it/ mezzina

Service definition and invocation. Service definitions and invocations resemble CCS prefixes. Thus $s.P$ defines a service s and we write $\overline{s}.Q$ for invoking s. The similarity with CCS is merely syntactical, because after the handshake P and Q are not quite separate continuations, but rather protocols that will interact in a fresh, private session. The name of the session is not to be mentioned in P and Q, and therefore 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. Services are typically one-shot, in the sense that when invoked, a new instance serving the request is created, but the service in no longer available. This choice facilitates service updates. Replication (or recursion) can be used to specify persistent services, like $!s.P$. 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.

Session sides. The handshake between $s.P$ and $\overline{s}.Q$ leads to the creation of a fresh session name r that can be viewed as a private, synchronous channel binding caller and callee. Since client and service may be far apart, a session naturally comes with two sides, written $r \triangleright P$ and $r \triangleright Q$, with r bound somewhere above them. For example, starting from $R|\mathbb{C}_1[\![\, s.P \,]\!]|\mathbb{C}_2[\![\, \overline{s}.Q \,]\!]$ we can arrive to $R|(\nu r)(\mathbb{C}_1[\![\, r \triangleright P \,]\!]|\mathbb{C}_2[\![\, r \triangleright Q \,]\!])$. Similarly, starting from $\mathbb{C}_1[\![\, s_1.(P_1|\overline{s}_2.P_2) \,]\!]|\mathbb{C}_2[\![\, \overline{s}_1.Q \,]\!]|\mathbb{C}_3[\![\, s_2.R \,]\!]$ we can arrive in two steps to $(\nu r_1, r_2)(\mathbb{C}_1[\![\, r_1 \triangleright (P_1|r_2 \triangleright P_2) \,]\!]|\mathbb{C}_2[\![\, r_1 \triangleright Q \,]\!]|\mathbb{C}_3[\![\, r_2 \triangleright R \,]\!])$, where P_1 interacts with Q and P_2 interacts with R. Sometimes, especially when type systems are considered, *polarities* $+$ and $-$ are attached to session sides in order to mark the caller and the callee. In the example above, we should have written, e.g., $(\nu r_1, r_2)(\mathbb{C}_1[\![\, r_1^+ \triangleright (P_1|r_2^- \triangleright P_2) \,]\!]|\mathbb{C}_2[\![\, r_1^- \triangleright Q \,]\!]|\mathbb{C}_3[\![\, r_2^+ \triangleright R \,]\!])$

Intra-session communication. Two peers P and Q running on opposite session sides of r can exchange messages. Since the peer is uniquely determined, input and output primitives are, respectively, abstraction prefixes $(?\vec{x})P$ or concretion prefixes $\langle\vec{v}\rangle Q$. For example, $\mathbb{C}_1[\![\, r \triangleright \langle 5\rangle P \,]\!]|\mathbb{C}_2[\![\, r \triangleright (?x)Q \,]\!]$ can evolve to $\mathbb{C}_1[\![\, r \triangleright P \,]\!]|\mathbb{C}_2[\![\, r \triangleright Q\{^5/_x\} \,]\!]$.

If we now reconsider the service for scientific calculations from Example 3, then we can write it just as $S \triangleq !in.(?x)\langle f(x)\rangle$. Then if two clients $\overline{in}.\langle 1\rangle(?y_1)P_1$ and $\overline{in}.\langle 2\rangle(?y_2)P_2$ are present, there is no risk of interference, because the two interactions are served separately.

$$
\begin{aligned}
& S\,|\,\overline{in}.\langle 1\rangle(?y_1)P_1\,|\,\overline{in}.\langle 2\rangle(?y_2)P_2 \\
& \quad \xrightarrow{\tau} (\nu r_1)(S\,|\,r_1 \triangleright (?x)\langle f(x)\rangle\,|\,r_1 \triangleright \langle 1\rangle(?y_1)P_1\,|\,\overline{in}.\langle 2\rangle(?y_2)P_2\,) \\
& \quad \xrightarrow{\tau} (\nu r_1, r_2)(S\,|\,r_1 \triangleright (?x)\langle f(x)\rangle\,|\,r_2 \triangleright (?x)\langle f(x)\rangle\,|\,r_1 \triangleright \langle 1\rangle(?y_1)P_1\,|\,r_2 \triangleright \langle 2\rangle(?y_2)P_2\,) \\
& \quad \xrightarrow{\tau} (\nu r_1, r_2)(S\,|\,r_1 \triangleright \langle f(1)\rangle\,|\,r_2 \triangleright (?x)\langle f(x)\rangle\,|\,r_1 \triangleright (?y_1)P_1\,|\,r_2 \triangleright \langle 2\rangle(?y_2)P_2\,) \\
& \quad \xrightarrow{\tau} (\nu r_1, r_2)(S\,|\,r_1 \triangleright \langle f(1)\rangle\,|\,r_2 \triangleright \langle f(2)\rangle\,|\,r_1 \triangleright (?y_1)P_1\,|\,r_2 \triangleright (?y_2)P_2\,) \\
& \quad \xrightarrow{\tau} (\nu r_1, r_2)(S\,|\,r_1 \triangleright \mathbf{0}\,|\,r_2 \triangleright \langle f(2)\rangle\,|\,r_1 \triangleright P_1\{^{f(1)}/_{y_1}\}\,|\,r_2 \triangleright (?y_2)P_2\,) \\
& \quad \xrightarrow{\tau} (\nu r_1, r_2)(S\,|\,r_1 \triangleright \mathbf{0}\,|\,r_2 \triangleright \mathbf{0}\,|\,r_1 \triangleright P_1\{^{f(1)}/_{y_1}\}\,|\,r_2 \triangleright P_2\{^{f(2)}/_{y_2}\}\,)
\end{aligned}
$$

Note that the initial processes are much simpler than those considered in Example 3, where session identifiers k should appear explicitly. Moreover, the session side construct must be considered as run-time syntax, as all the more complex processes traversed by the above computation.

Inter-session communication. It is quite useful to have the possibility to make the responses obtained upon some service invocation available to the parent session, e.g. to collect the fares offered from different providers and compare them to choose the best one. To this purpose, another prefix is available in CaSPiS, called *return prefix*, written $\langle \vec{v} \rangle^{\uparrow} P$, which can be seen as a concretion at the level of the parent session, i.e. $r \rhd \langle \vec{v} \rangle^{\uparrow} P$ can be read as $\langle \vec{v} \rangle | r \rhd P$, except for the fact that P cannot execute until $\vec{v}$ has been consumed.

Pipelining. CaSPiS exploits a generalised form of Orc sequencing operator, called *pipeline* and written $P > Q$, 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 concretions available in P: instead of being available to the peer of the current session, they are given in input to Q, which is typically guarded by some abstraction prefix. For example, Orc sequencing operator can be written as $P > (?x)Q$.

Note that in a term like $(r \rhd (P > Q)) | r \rhd R$, process P can input from R and output to Q. This is clearly different from $((r \rhd P) > Q) | r \rhd R$, where P can input from R and output to R, but can pass values to Q using return prefixes. In combination with the return operator, pipeline allows to make the responses obtained upon some service invocation available locally, to some suitable continuation. In the example of the service for scientific calculations, a client such as $\overline{in}.\langle 1 \rangle (?y_1) P_1$ would run P_1 in the session established with S, i.e. it will reduce after some steps to $r_1 \rhd P_1\{f(1)/y_1\}$. Instead the client $(\overline{in}.\langle 1 \rangle (?y_1) \langle y_1 \rangle^{\uparrow}) > (?y_1) P_1$ will reduce, after some further steps, to $(r_1 \rhd \mathbf{0} > (?y_1) P_1) | P_1\{f(1)/y_1\}$, which can be read as $P_1\{f(1)/y_1\}$, because $(r_1 \rhd \mathbf{0} > (?y_1) P_1)$ is a terminated process.

Cancellation. Processes must be able to abandon their current sessions in full autonomy. The command close is used to terminate the enclosing session side. A terminated session enters the special state $\blacktriangleright P$ that recursively terminates any other session side nested in P. Note that the execution of a close can depend on some local choice as well as be guarded by the input of some data from the opposite session side.

Closure notification. The distinguishing feature of CaSPiS is the presence of novel primitives to handle (unexpected or programmed) session termination. In fact, even if processes can abandon their current sessions, we would like sessions units to represent a controlled and safe form interactions, and therefore their peers should be somehow notified. 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 this purpose, the more general syntax for invocation is $\overline{s}_k.Q$: it mentions a name k at which the handler of the client-side is listening. Symmetrically, the more general syntax for service definition is $s_k.P$, which mentions a name k at which the handler of the service-side is listening. Upon creation of a session, the pair of names (k_1, k_2) is thus associated with the fresh session r, identifying a pair of *termination handlers*, one for each side. The more general syntax for sessions is thus $r \rhd_k P$ where the subscript k refers to the termination handler of the opposite side.

Then, after a close is executed, a notification $\dagger(k)$ is sent to the termination-handler service k listening at the *opposite* side of the session to manage the appropriate actions.

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

To sum up the above discussion: $s_{k_1}.P | \overline{s}_{k_2}.Q$ can evolve to $(\nu r)(r \rhd_{k_2} P | r \rhd_{k_1} Q)$. (Note that the handlers have been exchanged between the peers.) Then, if say P executes close, the termination handler k_2 of the caller will be activated, and vice versa, if Q terminates, then k_1 will be activated. For example: $r \rhd_{k_2} (\mathsf{close} | P) | r \rhd_{k_1} (Q | k_2 \cdot \mathsf{close})$ can evolve to $\blacktriangleright P | \dagger(k_2) | r \rhd_{k_1} (Q | k_2 \cdot \mathsf{close})$, then to $\blacktriangleright P | r \rhd_{k_1} (Q | \mathsf{close})$ and finally to $\blacktriangleright P | \blacktriangleright Q | \dagger(k_1)$. 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 even worst situation of dangling session sides pending forever.

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. Roughly, this is obtained by allowing: 1) output and return prefixes whose values are structured, exploiting a signature Σ of *constructors*, ranged by f (each coming with a fixed arity); 2) input prefixes where plain input variables $?x$ are generalised by *patterns* that exploit the constructors in Σ. 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 PDF (Portable Document Format) or PS (PostScript) format.

3.2 Close-free fragment

We start presenting the fragment of CaSPiS without cancellation and closure notification, whose syntax is in Fig. 6. The operators are listed in decreasing order of precedence. 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, while the remaining operators are *static*.

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 empty sum is 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 $\langle\rangle^{\uparrow} P$.

The structural congruence relation $\equiv$ is defined as the least congruence that includes alpha-equivalence and the laws in Fig. 7. This set of laws comprises the structural rules for parallel composition and restriction, plus the obvious extension of restriction's scope extrusion law to pipelines and sessions.

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

Fig. 6. Syntax of close-free CaSPiS

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

Fig. 7. Structural congruence laws

The reduction semantics is given by exploiting suitable contexts surrounding the active redexes. A context is *static* if its hole does not occur under a dynamic operator. Moreover, we say that a context is *session-immune* if its hole does not occur under a session operator, and *pipeline-immune* if its hole does not occur under a right-sided pipeline operator. In the following we let $\mathbb{C}[\![\cdot]\!]$ range over static contexts, $\mathbb{S}[\![\cdot]\!]$ over static session-immune contexts, and $\mathbb{P}[\![\cdot]\!]$ over contexts that are static, session-immune and pipeline-immune. Roughly, a static session-immune context $\mathbb{S}[\![\cdot]\!]$ cannot "intercept" abstraction and return prefixes, while a static, session-immune and pipeline-immune context $\mathbb{P}[\![\cdot]\!]$ cannot "intercept" concretion prefixes. Analogous definitions apply to the case of two-holes contexts $\mathbb{C}[\![\cdot,\cdot]\!]$.

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

$$
(\qquad)\frac{r \ \text{ fresh for } \ \mathbb{C}[\![\cdot,\cdot]\!], P, Q}{\mathbb{C}[\![\, s.P, \overline{s}.Q \,]\!] \xrightarrow{\tau} (\nu r)\mathbb{C}[\![\, r \triangleright P, r \triangleright Q \,]\!]}
$$

The second reduction regards intra-session communication. Below we let $\mathbb{C}_r[\![\cdot,\cdot]\!]$ be a context of the form $\mathbb{C}[\![\, r \triangleright \mathbb{P}[\![\cdot]\!], r \triangleright \mathbb{S}[\![\cdot]\!] \,]\!]$ (for some $\mathbb{P}[\![\cdot]\!]$ and $\mathbb{S}[\![\cdot]\!]$), which captures the most general situation in which intra-session communication can happen. 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 $F\sigma = V$. Moreover, we implicitly require that names in F and V are not bound by $\mathbb{C}_r[\![\cdot,\cdot]\!]$

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

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

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

Finally, there are two more rules for pipeline orchestration, handling the "redirection" of concretions and returns.

$$(\mathrm{P}\qquad)\ \frac{Q \equiv \mathbb{S}[\![\, (F)Q' + \sum_j \pi_j Q_i \,]\!] \quad \sigma = \mathrm{match}(F,V)}{\mathbb{C}[\![\, \mathbb{P}[\![\, \langle V\rangle P + \sum_i \pi_i P_i \,]\!] > Q \,]\!] \xrightarrow{\tau} \mathbb{C}[\![\, \mathbb{S}[\![\, Q'\sigma \,]\!] \,|\, (\mathbb{P}[\![\, P \,]\!] > Q) \,]\!]}$$

$$(\mathrm{PR}\qquad)\ \frac{Q \equiv \mathbb{S}[\![\, (F)Q' + \sum_j \pi_j Q_i \,]\!] \quad \sigma = \mathrm{match}(F,V)}{\mathbb{C}[\![\, \mathbb{P}[\![\, r \rhd \mathbb{S}_1[\![\, \langle V\rangle^{\uparrow} P + \sum_i \pi_i P_i \,]\!] \,]\!] > Q \,]\!] \xrightarrow{\tau} \mathbb{C}[\![\, \mathbb{S}[\![\, Q'\sigma \,]\!] \,|\, (\mathbb{P}[\![\, r \rhd \mathbb{S}_1[\![\, P \,]\!] \,]\!] > Q) \,]\!]}$$

The presence of contexts in the reduction rules accounts for the execution of silent transitions under restriction, parallel composition, etc, while we omit deliberately the obvious rule for structural congruence, which is the same as the rule (R) of pi-calculus (see Section 2.2). The LTS semantics and the Lemma that reconciles the silent transitions of the two semantics can be found in [8].

Example 5. The one-way and request-response invocation patterns from web services (to service s with argument V) can be easily encoded as $\overline{s}.\langle V\rangle\langle\rangle^{\uparrow}$ and $\overline{s}.\langle V\rangle(?x)\langle x\rangle^{\uparrow}$, respectively. Note that in both cases a value is returned (possibly void) that can be used to activate a suitable continuation, if any. The one-way pattern can also be rendered in a fully asynchronous fashion by writing $\overline{s}.(\langle V\rangle \,|\, \langle\rangle^{\uparrow})$.

The following e-shop example is adapted from [5], where it is 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.

Example 6. We model a simple e-shop application S that exchanges information with customers C and the data base D that stores item prices. The service *price* 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.

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

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 costumer contains the right price.

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

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

Exercise 8. 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 conclude by hinting at two important properties of CaSPiS processes P that do not contain the session operator $r \rhd [\![\cdot]\!]$. Let Q be any process reachable from P via any number of reductions and let r any session in Q, then: 1) there are exactly two session sides for r in Q (*dyadic session*), 2) it is never the case that one of the session side for r is nested into the other (*session acyclicity*). We refer to [8] for the formal presentation of such properties.

3.3 Full calculus

We can now present the full syntax and semantics of CaSPiS. 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. 8. The difference w.r.t. Fig. 6 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 $k \cdot P$. Like in the case of $r \rhd P$, we reserve $r \rhd_k P$ and $\blacktriangleright P$ as run-time syntax. 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 $r \rhd_k P$.

The structural rules listed in Fig. 9 enrich the set of rules already introduced for the close-free fragment. The law $\blacktriangleright \dagger(k) \equiv \dagger(k)$ is motivated by subtle race conditions on the order of closings due to the nesting of sessions (see [8] for an example). The remaining rules serve the purpose of letting signals $\dagger(k)$ freely move within a term to reach the corresponding listeners, and distributing the terminated session $\blacktriangleright$ over static operators. Note that, as usual, structural congruence can be exploited to move to top level all restrictions that are not in the scope of a dynamic operator.

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

Fig. 8. Syntax of full CaSPiS.

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

Fig. 9. Structural congruence rules for $\dagger(k)$ and $\blacktriangleright$.

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.

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

Rule (P) is left unchanged, while we need to annotate the sessions appearing in rules (S), (SR) and (PR) (and in the notation $\mathbb{C}_r[\![\cdot,\cdot]\!]$) with suitable termination handlers k and k_1.

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

$$
(\text{S}\quad)\frac{}{\mathbb{C}[\![r \rhd_k \mathbb{S}[\![\mathsf{close}]\!]]\!] \xrightarrow{\tau} \mathbb{C}[\![\dagger(k) | \blacktriangleright \mathbb{S}[\![\mathbf{0}]\!]]\!]}
$$

or to the termination of an enclosing session:

$$
(\text{T}\quad)\frac{}{\mathbb{C}[\![\blacktriangleright (r \rhd_k P)]\!] \xrightarrow{\tau} \mathbb{C}[\![\blacktriangleright P | \dagger(k)]\!]}
$$

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

$$
(\text{T}\quad)\frac{}{\mathbb{C}[\![\dagger(k) | k \cdot P]\!] \xrightarrow{\tau} \mathbb{C}[\![P]\!]}
$$

The session closing primitives do not guarantee *per se* that forever-dangling, one-sided sessions never arise, in the same way as deadlock can arise in pi-calculus processes or sequential programs may diverge. 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. Moreover, we can allow rather liberal choices of $\mathbb{C}[\![\cdot]\!]$, that may contain extra actions the termination handler may wish to take upon invocation, e.g., further signalling to other listeners (a sort of compensation, in the language of long-running transactions).

Again, the full technical details can be found in [8], 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 property guarantees that *any possibly unbalanced term reachable from a balanced term can get balanced in a finite number of reductions*.

For a process P that contains no session constructs, we require e.g. that for any $Q \equiv P$ and for each s and k: (a) s_k. may only occur in Q in subterms of the form $s_k.\mathbb{S}_1[\![k \cdot \mathbb{S}_2[\![\mathsf{close}]\!]]\!]$ and analogously for $\overline{s}_k$.; (b) in Q there is at most one occurrence of the listener for k.

For example, obvious "graceful" usages for service invocation and service definition are $(\nu k_1)\overline{s}_{k_1}.(P_1|k_1 \cdot \mathsf{close})$ and $(\nu k_2)s_{k_2}.(P_2|k_2 \cdot \mathsf{close})$, respectively. The process *News* from next example (adapted from [8]) also fits the requirements for the graceful property.

Example 7. Let *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 resemble $!BBC.!(\nu n)\langle n\rangle$, etc.). Let us consider the process *News* below that exposes a *news collector* service *collect*:

$$\begin{aligned} News \triangleq\ !(\nu k)collect_k.\ (\ k \cdot \mathsf{close}\ \mid\ & (\nu k_1)\overline{BBC}_{k_1}.(!(?x)\langle x\rangle^{\uparrow}\ \mid\ k_1 \cdot (\mathsf{close} \mid \dagger(k))) \\ \mid\ & (\nu k_2)\overline{CNN}_{k_2}.(!(?x)\langle x\rangle^{\uparrow}\ \mid\ k_2 \cdot (\mathsf{close} \mid \dagger(k)))\ ) \end{aligned}$$

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

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

$$HeavyReader \triangleq (\nu k')\overline{collect_{k'}}.(!(?y)\langle y\rangle^{\uparrow} | k' \cdot \mathsf{close})$$

Instead the client below receives only the first news produced either by *BBC* or *CNN* and then abandons the session:

$$EasyReader \triangleq (\nu k')\overline{collect_{k'}}.((?y)\langle y\rangle^{\uparrow}\mathsf{close} \,|\, k' \cdot \mathsf{close})$$

It is worth mentioning that there are at least two obvious alternatives to the mechanism we have chosen. One would be to use close as a primitive for terminating instantaneously *both* the client-side and service-side sessions. But 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 a reasonable compromise: each party can exit a session autonomously but it is obliged to inform the other party.

3.4 Other variants

Some variants of CaSPiS have been recently considered in recent literature, that introduce suitable restrictions to favour analysis and verification of processes. We mention a few significant works.

In [15], it is assumed that: (1) service definitions can only be present at the top level and cannot 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 by opening new nested sessions. In [1], 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 [45] by introducing general recursion at the level of session protocols and using the type system to prevent communication errors.

In [38] a security-oriented extension of the work in [15] 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 [25] 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 prototype implementations of CaSPiS have been proposed in [4, 10].

$$\llbracket x \rrbracket_a^{\lambda 2c} \triangleq \overline{x}.\langle a \rangle$$
$$\llbracket \lambda x.M \rrbracket_a^{\lambda 2c} \triangleq a.(?x)(?m)\llbracket M \rrbracket_m^{\lambda 2c}$$
$$\llbracket M\,N \rrbracket_a^{\lambda 2c} \triangleq (\nu m)\Big(\llbracket M \rrbracket_m^{\lambda 2c} | (\nu b)(\,!b.(?n)\llbracket N \rrbracket_n^{\lambda 2c} | \overline{m}.\langle b \rangle \langle a \rangle \,) \Big) \;\text{ for } b \notin \mathrm{fv}(N)$$

Fig. 10. Encoding of λ-calculus in CaSPiS

4 Application examples

In this section we show a few intuitive encoding of paradigmatic calculi in CaSPiS and of a simple fragment of CaSPiS in pi-calculus, but without proving any strong formal correspondence.

4.1 From lambda-calculus to CaSPiS

We start by showing that the close-free fragment of CaSPiS is expressive enough to encode λ-calculus, in a similar way as done, e.g., in pi-calculus (see Example 2).

The encoding is summarised in Fig. 10, where $\llbracket M \rrbracket_a^{\lambda 2c}$ denotes the CaSPiS process modelling the λ-expression M with arguments retrieved through the service a. Notably the encoding uses just monadic messaging without exploiting pipelines, choices, return prefixes and pattern matching.

From the point of view of syntax, the main differences w.r.t. the pi-calculus encoding are: (i) service definitions replace input prefixes; (ii) service invocations replace output prefixes. From the point of view of semantics, the more important differences are: (i) each service invocation opens a new session where the computation can progress; (ii) the session can be nested at different levels of depth and are never closed.

Exercise 9. Write the CaSPiS processes $\llbracket \lambda x.x \rrbracket_a^{\lambda 2c}$ and $\llbracket (\lambda x.x)N \rrbracket_a^{\lambda 2c}$. Then write all reduction steps of $\llbracket (\lambda x.x)N \rrbracket_a^{\lambda 2c}$.

4.2 From pi-calculus to CaSPiS

Quite interestingly, choice-free pi-calculus can be encoded in the close-free fragment of CaSPiS. In fact, pi-calculus communication primitives can be seen as services with minimal protocols. Note however that encoding the pi-calculus process $a(x).P$ as the CaSPiS process $a.(?x)C$ (for C the encoding of P) would unnecessarily run C in a nested session. To avoid this problem, and to make the encoding more elegant, it suffices to exploit pipelines and pattern-matching. For simplicity we focus on the monadic pi-calculus without sum. The problem with choices is due to the fact that CaSPiS sums can be applied only to abstraction, concretion and return prefixes, but not to service definition and invocations.

The encoding of pi-calculus processes is defined rather straightforwardly in Fig. 11.

$$
\begin{aligned}
[\![\mathbf{0}]\!]^{\pi 2c} &\triangleq \mathbf{0}\\
[\![x(y).P]\!]^{\pi 2c} &\triangleq x.(?y)\langle y\rangle^{\uparrow} > (?y)[\![P]\!]^{\pi 2c}\\
[\![\overline{x}\langle y\rangle.P]\!]^{\pi 2c} &\triangleq \overline{x}.\langle y\rangle\langle\rangle^{\uparrow} > ()[\![P]\!]^{\pi 2c}\\
[\![\tau.P]\!]^{\pi 2c} &\triangleq \langle\rangle > ()[\![P]\!]^{\pi 2c}\\
[\![[x=y]\pi.P]\!]^{\pi 2c} &\triangleq \langle x\rangle > (y)[\![\pi.P]\!]^{\pi 2c}\\
[\![P_1 \,|\, P_2]\!]^{\pi 2c} &\triangleq [\![P_1]\!]^{\pi 2c} \,|\, [\![P_2]\!]^{\pi 2c}\\
[\![(\nu x)P]\!]^{\pi 2c} &\triangleq (\nu x)[\![P]\!]^{\pi 2c}\\
[\![!P]\!]^{\pi 2c} &\triangleq ![\![P]\!]^{\pi 2c}
\end{aligned}
$$

Fig. 11. Encoding of π-calculus in CaSPiS

$$
\begin{aligned}
[\![E(x) \triangleq e]\!]^{o2c} &\triangleq E.(?x)[\![e]\!]^{o2c}\\
[\![\mathbf{0}]\!]^{o2c} &\triangleq \mathbf{0}\\
[\![\langle p\rangle]\!]^{o2c} &\triangleq [\![p]\!]_{\mathrm{v}}\\
[\![E(p)]\!]^{o2c} &\triangleq \overline{E}.\langle p\rangle!(?x_r)\langle x_r\rangle^{\uparrow}\\
[\![s(p)]\!]^{o2c} &\triangleq [\![p]\!]_{\mathrm{v}} > (?x_p)\overline{s}.\langle x_p\rangle(?x_r)\langle x_r\rangle^{\uparrow}\\
[\![x(p)]\!]^{o2c} &\triangleq [\![x]\!]_{\mathrm{v}} > (?s)[\![s(p)]\!]^{o2c}\\
[\![e_1 \,|\, e_2]\!]^{o2c} &\triangleq [\![e_1]\!]^{o2c} \,|\, [\![e_2]\!]^{o2c}\\
[\![e_1 > x > e_2]\!]^{o2c} &\triangleq [\![e_1]\!]^{o2c} > (?x)[\![e_2]\!]^{o2c}\\
[\![e_2 \textbf{ where } x :\in e_1]\!]^{o2c} &\triangleq (\nu wh, re, k)(\; \overline{wh}_k.([\![e_1]\!]^{o2c} > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close}\,)\,|\\
&\qquad\qquad wh.()\mathsf{close} \,|\, (\nu x)([\![e_2]\!]^{o2c} \,|\, re.(?x_1)!x.\langle x_1\rangle)\;)
\end{aligned}
$$

Fig. 12. Encoding of Orc in CaSPiS

Exercise 10. The encoding of λ-calculus in pi-calculus $[\![M]\!]_a^{\lambda 2\pi}$ can be combined with the above encoding of pi-calculus in CaSPiS to obtain an encoding $[\![M]\!]_a^{\lambda 2\pi 2c}$ of λ-calculus in CaSPiS. After giving the explicit definition of $[\![x]\!]_a^{\lambda 2\pi 2c}$, $[\![\lambda x.M]\!]_a^{\lambda 2\pi 2c}$ and $[\![MN]\!]_a^{\lambda 2\pi 2c}$, compare the encoding with the one defined in Fig. 10 and explain the main differences, if any.

Then, write the CaSPiS processes $[\![\lambda x.x]\!]_a^{\lambda 2\pi 2c}$ and $[\![(\lambda x.x)N]\!]_a^{\lambda 2\pi 2c}$, together with all reduction steps of $[\![(\lambda x.x)N]\!]_a^{\lambda 2\pi 2c}$.

4.3 From Orc to CaSPiS

In [7] it was shown how to encode Orc in CaSPiS. Here we essentially rephrase (and simplify) the translation using CaSPiS syntax. An Orc expression may depend on a set of expression definitions, hence the encoding of an Orc expression comprises the encoding of all expression definitions (as processes composed in parallel).

The encoding of Orc expressions is detailed in Fig. 12. A few points are worth some comments. 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 the variable manager of x. Variable managers are created by both sequential composition and asymmetric parallel composition.

The most interesting part of the encoding regards the asymmetric parallel composition. Two fresh services wh and re are used, respectively, to enclose the evaluation of e_1 in a session that can be terminated and to receive the first value provided by e_1 and install the manager for variable x with that value. This is exemplified below, where we omit all restriction to improve readability and write P_1 and P_2 in place of $[\![e_1]\!]^{o2c}$ and $[\![e_2]\!]^{o2c}$, respectively.

$$\begin{array}{l} \overline{wh}_k.(P_1 > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close})\,|\, wh.()\mathsf{close}\,|\, P_2\,|\, re.(?x_1)!x.\langle x_1\rangle \\ \xrightarrow{\tau} r_w \rhd (P_1 > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close})\,|\, r_w \rhd_k ()\mathsf{close}\,|\, P_2\,|\, re.(?x_1)!x.\langle x_1\rangle \end{array}$$

Note that $[\![e_2]\!]^{o2c}$ is executed concurrently, but may rely on value requests to the manager for x. When $[\![e_1]\!]^{o2c}$ produces a concretion, it flows through the pipeline and activates the invocation to re.

$$\begin{array}{l} r_w \rhd (P_1 > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close})\,|\, r_w \rhd_k ()\mathsf{close}\,|\, P_2\,|\, re.(?x_1)!x.\langle x_1\rangle \\ \xrightarrow{\tau} r_w \rhd (\overline{re}.\langle 8\rangle\langle\rangle^{\uparrow} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close}))\,|\, r_w \rhd_k ()\mathsf{close}\,|\, P_2\,|\, re.(?x_1)!x.\langle x_1\rangle \\ \xrightarrow{\tau} r_w \rhd (r \rhd \langle 8\rangle\langle\rangle^{\uparrow} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close}))\,|\, r_w \rhd_k ()\mathsf{close}\,|\, P_2\,|\, r \rhd (?x_1)!x.\langle x_1\rangle \end{array}$$

Note that the service definition for re is not replicated and thus only one request may be issued. The value produced by $[\![e_1]\!]^{o2c}$ is communicated by the client session side of re to its peer session side, which in turn can install a persistent service definition for variable x (its manager w.r.t. requests in $[\![e_2]\!]^{o2c}$). The void return prefix on the client side instance or re is now available and can handshake with the void abstraction on the service side instance of wh, enabling the execution of close.

$$\begin{array}{l} r_w \rhd (r \rhd \langle 8\rangle\langle\rangle^{\uparrow} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close}))\,|\, r_w \rhd_k ()\mathsf{close}\,|\, P_2\,|\, r \rhd (?x_1)!x.\langle x_1\rangle \\ \xrightarrow{\tau} r_w \rhd (r \rhd \langle\rangle^{\uparrow} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close}))\,|\, r_w \rhd_k ()\mathsf{close}\,|\, P_2\,|\, r \rhd !x.\langle 8\rangle \\ \xrightarrow{\tau} r_w \rhd (r \rhd \mathbf{0} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close}))\,|\, r_w \rhd_k \mathsf{close}\,|\, P_2\,|\, r \rhd !x.\langle 8\rangle \end{array}$$

The effect of close is to terminate the enclosing session side and to notify the listener k (within the client side instance of wh), which in turn will terminate the enclosing session side.

$$\begin{array}{l} r_w \rhd (r \rhd \mathbf{0} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close}))\,|\, r_w \rhd_k \mathsf{close}\,|\, P_2\,|\, r \rhd !x.\langle 8\rangle \\ \xrightarrow{\tau} r_w \rhd (r \rhd \mathbf{0} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, k\cdot \mathsf{close}))\,|\, \dagger(k)\,|\, \blacktriangleright \mathbf{0}\,|\, P_2\,|\, r \rhd !x.\langle 8\rangle \\ \equiv \quad r_w \rhd (r \rhd \mathbf{0} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, \dagger(k)\,|\, k\cdot \mathsf{close}))\,|\, P_2\,|\, r \rhd !x.\langle 8\rangle \\ \xrightarrow{\tau} r_w \rhd (r \rhd \mathbf{0} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow} \,|\, \mathsf{close}))\,|\, P_2\,|\, r \rhd !x.\langle 8\rangle \\ \xrightarrow{\tau} \blacktriangleright (r \rhd \mathbf{0} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow}))\,|\, P_2\,|\, r \rhd !x.\langle 8\rangle \end{array}$$

Consequently, the nested session side of *re* is also terminated, but not its peer (because no termination handler was exchanged when the session was created). In fact the manager for x is running inside that peer and we cannot terminate it.

$$
\begin{array}{l}
\blacktriangleright (r \rhd \mathbf{0} \,|\, (P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow})) \,|\, P_2 \,|\, r \rhd !x.\langle 8\rangle \\
\equiv \quad (\blacktriangleright P_1' > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow}) \,|\, P_2 \,|\, r \rhd !x.\langle 8\rangle \\
\rightarrow^* (\blacktriangleright \mathbf{0} > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow}) \,|\, P_2 \,|\, r \rhd !x.\langle 8\rangle \\
\equiv \quad (\mathbf{0} > (?x_1)\overline{re}.\langle x_1\rangle\langle\rangle^{\uparrow}) \,|\, P_2 \,|\, r \rhd !x.\langle 8\rangle
\end{array}
$$

Note that this makes the process not well-balanced, as it contains a dangling session side that cannot terminate, i.e., the encoding we have provided does not satisfy the graceful closure property.

Exercise 11. Write the CaSPiS processes that encodes the Orc expressions $\langle 1\rangle \,|\, \langle 2\rangle > x > \langle x\rangle$ and $\langle x\rangle$ **where** $x :\in \langle 1\rangle \,|\, \langle 2\rangle$. Then write all the CaSPiS processes that can be reached from them via any number of reductions.

Exercise 12. Write the CaSPiS processes that encodes the Orc expression *POR* for the parallel-or (see Exercise 7).

Exercise 13. Modify the encoding shown in Fig. 12 to guarantee the graceful termination property. Start by changing the way in which the manager of x is installed in the encoding of asymmetric parallel composition. Then, remind that $[\![e_1]\!]^{o2c}$ could have opened many other session sides before cancellation occurs and hence find suitable policies for invoking sites, expression definitions and local services.

4.4 From close-free CaSPiS to pi-calculus

We conclude by sketching an encoding of a fragment of CaSPiS in pi-calculus. In particular we restrict to consider the close-free fragment, without pattern matching and with a limited form of pipeline (which essentially coincides with Orc sequencing operator). Moreover we assume session sides are polarised $r^+ \rhd P$ and $r^- \rhd Q$.

The encoding of a CaSPiS process is dependent on its context. In particular, one can imagine that each CaSPiS process has three dedicated channels: one for the input associated with abstraction prefixes, one for output associated with concretion prefixes and one for output associated with return prefixes. Correspondingly, our encoding $[\![P]\!]^{c2\pi}_{in,out,ret}$ is parametric w.r.t. three names *in*, *out* and *ret*. The encoding is shown in Fig. 13, where we write $r^{\overline{p}}$ to denote the dual session of r^p.

The most interesting part of the encoding is concerned with service definition, service invocation, session siding and pipeline. A service invocation is encoded by creating two fresh names r^+ and r^- that will be used for intra-session communication: the service side will use them for input and output, respectively; vice versa the client side will use them for output and input, respectively. The presence of two names instead of just one guarantees that two concurrent processes running on the same session side cannot interact. Consequently, a session side r^p uses the name r^p for input and $r^{\overline{p}}$ for output. Note also the name for return in the nested session coincides with the name used for output by its parent. Finally, a pipeline $P > (?x)Q$ must intercept the output of P and use

$$
\begin{aligned}
[\![\textstyle\sum_i \pi_i P_i]\!]^{c2\pi}_{in,out,ret} &\triangleq \textstyle\sum_i [\![\pi_i P_i]\!]^{c2\pi}_{in,out,ret}\\
[\![(?x)P]\!]^{c2\pi}_{in,out,ret} &\triangleq in(x).[\![P]\!]^{c2\pi}_{in,out,ret}\\
[\![\langle v\rangle P]\!]^{c2\pi}_{in,out,ret} &\triangleq \overline{out}\langle v\rangle.[\![P]\!]^{c2\pi}_{in,out,ret}\\
[\![\langle v\rangle^{\uparrow} P]\!]^{c2\pi}_{in,out,ret} &\triangleq \overline{ret}\langle v\rangle.[\![P]\!]^{c2\pi}_{in,out,ret}\\
[\![u.P]\!]^{c2\pi}_{in,out,ret} &\triangleq u(r^p, r^q).[\![P]\!]^{c2\pi}_{r^p,r^q,out}\\
[\![\overline{u}.P]\!]^{c2\pi}_{in,out,ret} &\triangleq (\nu r^+, r^-)\overline{u}\langle r^+, r^-\rangle.[\![P]\!]^{c2\pi}_{r^-,r^+,out}\\
[\![r^p \rhd P]\!]^{c2\pi}_{in,out,ret} &\triangleq [\![P]\!]^{c2\pi}_{r^p,r^{\overline{p}},out}\\
[\![P > (?x)Q]\!]^{c2\pi}_{in,out,ret} &\triangleq (\nu p)(\ [\![P]\!]^{c2\pi}_{in,p,ret} \,|\, !p(x).[\![Q]\!]^{c2\pi}_{in,out,ret}\ ) \ \text{ for } p \notin \mathrm{fn}(P|Q)\\
[\![P_1 | P_2]\!]^{c2\pi}_{in,out,ret} &\triangleq [\![P_1]\!]^{c2\pi}_{in,out,ret} \,|\, [\![P_2]\!]^{c2\pi}_{in,out,ret}\\
[\![(\nu n)P]\!]^{c2\pi}_{in,out,ret} &\triangleq (\nu n)[\![P]\!]^{c2\pi}_{in,out,ret}\\
[\![!P]\!]^{c2\pi}_{in,out,ret} &\triangleq ![\![P]\!]^{c2\pi}_{in,out,ret}
\end{aligned}
$$

Fig. 13. Encoding of CaSPiS in pi-calculus

it to spawn fresh copies of Q. This is achieved by creating a fresh name p that is used for output by the encoding of P and that is used as input guard of a replicated process that spawns the copies of (the encoding of) Q.

Exercise 14. The encoding of λ-calculus in CaSPiS $[\![M]\!]^{\lambda 2c}_a$ can be combined with the above encoding of CaSPiS in pi-calculus to obtain an encoding $[\![M]\!]^{\lambda 2c2\pi}_a$ of λ-calculus in pi-calculus. After giving the explicit definition of $[\![x]\!]^{\lambda 2c2\pi}_a$, $[\![\lambda x.M]\!]^{\lambda 2c2\pi}_a$ and $[\![MN]\!]^{\lambda 2c2\pi}_a$, compare the encoding with the one defined in Fig. 4 and explain the main differences, if any.

Then, write the pi-calculus processes $[\![\lambda x.x]\!]^{\lambda 2c2\pi}_a$ and $[\![(\lambda x.x)N]\!]^{\lambda 2c2\pi}_a$, together with all reduction steps of $[\![(\lambda x.x)N]\!]^{\lambda 2c2\pi}_a$.

A type preserving encoding of (a variant of) CaSPiS in (a variant of) Honda, Vasconcelos and Kubo's session calculus has been recently defined in Leonardo Mezzina's PhD thesis [45].

5 Conclusion and future perspectives

In this tutorial we have tried to contribute along the following directions: (1) to outline several key characteristics of Service-Oriented Computing systems, (2) to sketch the basic principles, techniques and formal tools offered by the theory of process calculi, (3) to show that process calculi can likely offer a convenient formalism for representing SOC systems, but they need to be empowered by novel modelling approaches, developed at the right level of abstraction, (4) to overview some existing proposals and the different guidelines they are driven by, (5) to present in detail one such proposal, namely CaSPiS, and explain the rationale around its design choices, (6) to show how

CaSPiS can be related w.r.t. other well-established formalisms, so that readers more familiar with them can catch similarities and get a better understand of CaSPiS semantics, (7) to show that CaSPiS mechanism of termination handlers is very expressive, disciplined and flexible: even if it may look overcomplicated to use, we emphasise that, up to our knowledge, this is the only proposal able to guarantee a disciplined termination of nested sessions. We conjecture that any mechanism of this kind would be very complicated to handle in say pi-calculus.

We hope the quite informal level of presentation has been appreciated by readers not familiar with process calculi and may serve as a valid basis to learn more, possibly with the help of the many simple exercises populating the technical sections.

Regarding future work, there is still quite a lot of research to be done for refining and consolidating the different process calculi proposed for SOC, for integrating them with other more advanced aspects, like transactions and quality of service, for comparing them and relating them in a formal way. In particular, for CaSPiS, the overall objective is to have a rigorous theoretical framework, with some automatic tools available for type checking, type inference, quantitative analysis and rapid prototyping. We would like also to integrate other techniques, like those based on choreography, contracts, correlation sets, and multiparty sessions, within CaSPiS, possibly finding seamless ways to support such concepts on the existing machinery. Current work is also concerned with graphical encoding and concurrent semantics for SOC calculi, using models based on hierarchical graphs (that best reflect the nesting of sessions and the possibility to operate on the nested session sides as a whole, like when terminating atomically a session side and all its descendants).

References

1. L. Acciai and M. Boreale. A type system for client progress in a service-oriented calculus. In *Festschrift in Honour of Ugo Montanari, on the Occasion of His 65th Birthday*, volume 5065 of *Lect. Notes in Comput. Sci.* Springer Verlag, 2008. To appear.
2. L. Aceto, A. Ingólfsdóttir, K. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.
3. M. Bartoletti, P. Degano, G. Ferrari, and R. Zunino. Types and effects for Resource Usage Analysis. In *Foundations of Software Science and Computation Structures, FOSSACS'07*, volume 4423 of *Lect. Notes in Comput. Sci.*, pages 32–47. Springer Verlag, 2007.
4. L. Bettini, R. De Nicola, and M. Loreti. Implementing Session Centered Calculi. In D. Lea and G. Zavattaro, editors, *Coordination Models and Languages (COORDINATION2008)*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 17–32. Springer Verlag, 2008.
5. C. Bodei, L. Brodo, and R. Bruni. Static detection of logic flaws in service applications. In *Proceedings of ARSPA-WITS 2009, Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, Lect. Notes in Comput. Sci. Springer Verlag, 2009. To appear.
6. E. Bonelli and A. Compagnoni. Multisession session types for a distributed calculus. In *Proc. of TGC'07*, volume 4912 of *Lect. Notes in Comput. Sci.*, pages 240–256. Springer Verlag, 2008.
7. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: a service centered calculus. In *Proc. of WS-FM 2006*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 38–57. Springer Verlag, 2006.

8. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *Proc. of FMOODS'08*, volume 5051 of *Lect. Notes in Comput. Sci.*, pages 19–38. Springer Verlag, 2008.
9. M. Bravetti and G. Zavattaro. A Foundational Theory of Contracts for Multi-party Service Composition. *Fundamenta Informaticae*, 2009. To appear.
10. R. Bruni, R. De Nicola, M. Loreti, and L. Mezzina. Provably correct implementations of services. In *Proceedings of TGC 2008, 4th Symposium on Trustworthy Global Computing*, Lect. Notes in Comput. Sci. Springer Verlag, 2009. To Appear.
11. R. Bruni and I. Lanese. Parametric synchronizations in mobile nominal calculi. *Theoretical Computer Science*, 402(2-3):102–119, 2008.
12. R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty sessions in SOC. In D. Lea and G. Zavattaro, editors, *Proceedings of COORDINATION 2008, 10th International Conference on Coordination Models and Languages*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 67–82. Springer Verlag, 2008.
13. R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: extending Join. In J.-J. Lévy, E. Mayr, and J. Mitchell, editors, *Proceedings of the 3rd IFIP-TCS 2004, 3rd IFIP Intl. Conference on Theoretical Computer Science*, pages 569–582. Kluwer Academic Publishers, 2004.
14. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 209–220. ACM Press, 2005.
15. R. Bruni and L. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In G. Rosu and J. Meseguer, editors, *Proceedings of AMAST 2008, 12th International Conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lect. Notes in Comput. Sci.*, pages 100–115. Springer Verlag, 2008.
16. M. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In R. D. Nicola, editor, *Proc. of the 16th European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 18–32. Springer Verlag, 2007.
17. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of COORDINATION'06*, volume 4038 of *Lect. Notes in Comput. Sci.*, pages 63–81. Springer Verlag, 2006.
18. M. Butler, R. Bruni, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In M. Abadi and L. de Alfaro, editors, *Proceedings of Concur 2005*, volume 3653 of *Lect. Notes in Comput. Sci.*, pages 383–397. Springer Verlag, 2005.
19. M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In A. Abdallah and J. Sanders, editors, *Proceedings of 25 Years of CSP*, volume 3525 of *Lect. Notes in Comput. Sci.*, pages 133–150. Springer Verlag, 2005.
20. L. Caires, H. T. Vieira, and J. C. Seco. The conversation calculus: A model of service oriented computation. In S. Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *Lect. Notes in Comput. Sci.*, pages 269–283. Springer Verlag, 2008.
21. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proc. of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 2–17. Springer Verlag, 2007.
22. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *Proceedings of POPL'08*, pages 261–272, New York, NY, USA, 2008. ACM.
23. W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc. In *Proc. of COORDINATION'06*, volume 4038 of *Lect. Notes in Comput. Sci.*, pages 82–96. Springer Verlag, 2006.

24. E. Curtin and M. Warshauer. The locker puzzle. *The Mathematical Intelligencer*, 28(1):28–31, 2006.
25. R. De Nicola, D. Latella, M. Loreti, and M. Massink. MarCaSPiS: a markovian extension of a calculus for services. In *Proceedings of SOS 2008*, Elect. Notes in Th. Comput. Sci., 2008. To appear.
26. M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A distributed object-oriented language with session types. In *Proc. of TGC'05*, volume 3705 of *Lect. Notes in Comput. Sci.*, pages 299–318. Springer Verlag, 2007.
27. C. Ene and T. Muntean. A broadcast-based calculus for communicating systems. In *Proc. of IPDPS'01*. IEEE Computer Society, 2001.
28. G. Ferrari, R. Guanciale, and D. Strollo. JSCL: A Middleware for Service Coordination. In *Proceedings of FORTE 2006, 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems*, volume 4229 of *Lect. Notes in Comput. Sci.*, pages 46–60. Springer Verlag, 2006.
29. G. Ferrari, R. Guanciale, D. Strollo, and E. Tuosto. Coordination via types in an event-based framework. In *Proc. of FORTE'07*, volume 4574 of *Lect. Notes in Comput. Sci.*, pages 66–80, 2007.
30. S. Gay and M. Hole. Types and subtypes for client-server interactions. In *Proc. of ESOP'99*, volume 1576 of *Lect. Notes in Comput. Sci.*, pages 74–90. Springer Verlag, 1999.
31. R. v. Glabbeek. The linear time – branching time spectrum II; the semantics of sequential systems with silent moves (extended abstract). In E. Best, editor, Proceedings *CONCUR'93,* 4^{th} International Conference on *Concurrency Theory*, Hildesheim, Germany, August 1993, volume 715 of *Lect. Notes in Comput. Sci.*, pages 66–81. Springer Verlag, 1993.
32. R. v. Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.
33. C. Hoare. A model for communicating sequential processes. In *On the Construction of Programs*. Cambridge University Press, 1980.
34. K. Honda. Types for dyadic interaction. In *Proc. of CONCUR'93*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 509–523. Springer Verlag, 1993.
35. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proc. of ESOP'98*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 22–138. Springer Verlag, 1998.
36. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284, 2008.
37. D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, volume 4137 of *Lect. Notes in Comput. Sci.*, pages 477–491. Springer Verlag, 2006.
38. M. Kolundzija. Security types for sessions and pipelines. In R. Bruni and K. Wolf, editors, *Proc. of WS-FM 2008*, volume 5387 of *Lect. Notes in Comput. Sci.*, pages 176–190. Springer Verlag, 2009.
39. I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM'07*, pages 305–314. IEEE Computer Society Press, 2007.
40. C. Laneve and L. Padovani. 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 Verlag, 2008.
41. C. Laneve and G. Zavattaro. Foundations of web transactions. In V. Sassone, editor, *FOSSACS 2005*, volume 3441 of *Lect. Notes in Comput. Sci.*, pages 282–298. Springer Verlag, 2005.

42. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 33–47. Springer Verlag, 2007.
43. A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In D. Lea and G. Zavattaro, editors, *Proc. of the 10th International Conference on Coordination Models and Languages*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 199–215. Springer Verlag, 2008.
44. L. Mezzina. How to infer finite session types in a calculus of services and sessions. In *Proc. of COORDINATION'08*, volume 5052 of *Lect. Notes in Comput. Sci.*, pages 216–231. Springer Verlag, 2008.
45. L. Mezzina. *Typing Services*. PhD in Computer Science and Engineering, IMT Institute for Advanced Studies, Lucca, 2009.
46. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Comput. Sci.* Springer Verlag, 1980.
47. R. Milner. Functions as processes. *Math. Struct. in Comput. Sci.*, 2(2):119–141, 1992.
48. R. Milner. *Communicating and Mobile Systems: The pi-calculus*. Cambridge University Press, 1997.
49. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992.
50. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 2006. To appear. A preliminary version of this paper appeared in the Lecture Notes for NATO summer school, held at Marktoberdorf in August 2004.
51. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.
52. G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.
53. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
54. L. Rosaz. Puzzle corner #70: The 50 prisoners. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 86:229, 2005.
55. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, LFCS, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).
56. D. Sangiorgi and D. Walker. *The pi-calculus: a theory of mobile processes*. Cambridge University Press, 2001.
57. Sensoria Project. Software Engineering for Service-Oriented Overlay Computers. Public Web Site. http://sensoria.fast.de/.
58. V. Vasconcelos. Fundamentals of session types. In G. Z. Marco Bernardo, Luca Padovani, editor, *Formal Methods for Web Services*, Lect. Notes in Comput. Sci. Springer Verlag, 2009. This volume.
59. I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. A timed semantics of Orc. *Theoretical Computer Science*, 402(2-3):234–248, August 2008. DOI: 10.1016/j.tcs.2008.04.037.