A Calculus for Orchestration of Web Services ⋆

May 25, 2008

Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi

Dipartimento di Sistemi e Informatica Università degli Studi di Firenze

{lapadula,pugliese,tiezzi}@dsi.unifi.it

Abstract. We introduce COWS (*Calculus for Orchestration of Web Services*), a foundational language for SOC whose design has been influenced by WS-BPEL, the standard language for orchestration of web services. COWS combines in an original way a number of ingredients borrowed from well-known process calculi, e.g. asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities, while resulting different from any of them. Several examples illustrates COWS peculiarities and show its expressiveness both for modelling imperative and orchestration constructs, e.g. web services, flow graphs, fault and compensation handlers, and for encoding other process and orchestration languages. We also present an extension of the basic language with timed constructs.

⋆ The work presented in this paper is an extended version of [24], and has been partially supported by EU Project "Software Engineering for Service-Oriented Overlay Computers" (SENSORIA, contract IST-3-016004-IP-09).

Table of Contents

1 Introduction

Service-oriented computing (SOC) is an emerging paradigm for developing loosely coupled, interoperable, evolvable systems which exploits the pervasiveness of the Internet and its related technologies. SOC systems deliver application functionality as services to either end-user applications or other services. These very features foster a programming style based on service composition and reusability: new customized service-based applications can be developed on demand by appropriately assembling other existing, possibly heterogeneous, services.

Service definitions are used as templates for creating service instances that deliver application functionality to either end-user applications or other instances. The loosely coupled nature of SOC implies that the connection between communicating instances cannot be assumed to persist for the duration of a whole business activity. Therefore, there is no intrinsic mechanism for associating messages exchanged under a common context or as part of a common activity. Even the execution of a simple request-response message exchange pattern provides no built-in means of automatically associating the response message with the original request. It is up to each single message to provide a form of context thus enabling services to associate the message with others. This is achieved by embedding values in the message which, once located, can be used to correlate the message with others logically forming a same stateful interaction 'session'.

Early examples of technologies that are at least partly service-oriented are CORBA, DCOM, J2EE and IBM WebSphere. A more recent successful instantiation of the SOC paradigm are *web services*. These are autonomous, stateless, platform-independent and composable computational entities that can be published, located and invoked through the Web via XML messages. To support the web service approach, many new languages, most of which based on XML, have been designed, like e.g. business coordination languages (such as WS-BPEL, WSFL, WSCI, WS-CDL and XLANG), contract languages (such as WSDL and SWS), and query languages (such as XPath and XQuery). However, current software engineering technologies for development and composition of web services remain at the descriptive level and do not integrate such techniques as, e.g., those developed for component-based software development. Formal reasoning mechanisms and analytical tools are still lacking for checking that the web services resulting from a composition meet desirable correctness properties and do not manifest unexpected behaviors. The task of developing such verification methods is hindered also by the very nature of the languages used to program the services, which usually provide many redundant constructs and support quite liberal programming styles.

In the last few years, many researchers have exploited the studies on *process calculi* as a starting point to define a clean semantic model and lay rigorous methodological foundations for service-based applications and their composition. Process calculi, being defined algebraically, are inherently compositional and, therefore, convey in a distilled form the paradigm at the heart of SOC. This trend is witnessed by the many process calculi-like formalisms for orchestration and choreography, the two more common forms of web services composition. Most of these formalisms, however, do not suit for the analysis of currently available SOC technologies in their completeness because they only consider a few specific features separately, possibly by embedding *ad*

hoc constructs within some well-studied process calculus (see, e.g., the variants of π-calculus with transactions [2, 22, 23] and of CSP with compensation [9]).

Here, we follow a different approach and exploit WS-BPEL [1], the standard language for orchestration of web services, to drive the design of a new process calculus that we call COWS (*Calculus for Orchestration of Web Services*). Similarly to WS-BPEL, COWS supports shared states among service instances, allows a same process to play more than one partner role and permits programming stateful sessions by correlating different service interactions. However, COWS intends to be a foundational model not specifically tight to web services' current technology. Thus, some WS-BPEL constructs, such as e.g. fault and compensation handlers and flow graphs, do not have a precise counterpart in COWS, rather they are expressed in terms of more primitive operators (see Section 3). Of course, COWS has taken advantage of previous work on process calculi. Its design combines in an original way a number of constructs and features borrowed from well-known process calculi, e.g. asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities, while however resulting different from any of them.

The rest of the paper is organized as follows. Syntax and operational semantics of COWS are defined in Section 2 where we also show many illustrative examples. Section 3 presents the encodings of several imperative and orchestration constructs, while Section 4 illustrates two example applications of our framework to web services. Section 5 presents the encodings of three other orchestration languages, i.e. Orc [32], SCC [3] and ws-calculus [25]. Section 6 introduces an extension of COWS with timed orchestration constructs. This turns out to be a very powerful language that also permits to express, e.g., choices among alternative activities constrained by the expiration of some given timeout. Section 7 concludes the paper by touching upon comparisons with related work and directions for future work.

2 COWS: Calculus for Orchestration of Web Services

Before formally defining our language, we provide some insights on its main features. The basic elements of COWS are *partners* and *operations*. Alike channels in [13], a *communication endpoint* is not atomic but results from the composition of a partner name p and of an operation name o, which can also be interpreted as a specific implementation of o provided by p. This results in a very flexible naming mechanism that allows a same service to be identified by means of different logic names (i.e. to play more than one partner role as in WS-BPEL). For example, the following service

$$p_{slow} \bullet o?\bar{w}.s_{slow} + p_{fast} \bullet o?\bar{w}.s_{fast}$$

accepts requests for the same operation o through different partners with distinct access modalities: process s_{slow} implements a slower service provided when the request is processed through the partner p_{slow}, while s_{fast} implements a faster service provided when the request arrives through p_{fast}. Additionally, it allows the names composing an endpoint to be dealt with separately, as in a request-response interaction, where usually the service provider knows the name of the response operation, but not the partner name of the service it has to reply to. For example, the ping service

$p \cdot o_{req}?\langle x\rangle.x \cdot o_{res}!\langle$"I `live`"$\rangle$ will know at run-time the partner name for the reply activity. This mechanisms is also sufficiently expressive to support implementation of explicit locations: a located service can be represented by using a same partner for all its receiving endpoints. Partner and operation names can be exchanged in communication, thus enabling many different interaction patterns among service instances. However, as in [29], dynamically received names cannot form the communication endpoints used to receive further invocations.

COWS computational entities are called *services*. Typically, a service creates one specific instance to serve each received request. An instance is composed of concurrent threads that may offer a choice among alternative receive activities. Services could be able to receive multiple messages in a statically unpredictable order and in such a way that the first incoming message triggers creation of a service instance which subsequent messages are routed to. Pattern-matching is the mechanism for correlating messages logically forming a same interaction 'session' by means of their same contents. It permits locating those data that are important to identify service instances for the routing of messages and is flexible enough for allowing a single message to participate in multiple interaction sessions, each identified by separate correlation values.

To model and update the shared state of concurrent threads within each service instance, receive activities in COWS bind neither names nor variables. This is different from most process calculi and somewhat similar to [33, 34]. In COWS, however, inter-service communication give rise to substitutions of variables with values (alike [33]), rather than to fusions of names (as in [34]). The range of application of the substitution generated by a communication is regulated by the *delimitation* operator, that is the only binder of the calculus. Additionally, this operator permits to generate fresh names (as the restriction operator of the π-calculus [31]) and to delimit the field of action of the *kill* activity, that can be used to force termination of whole service instances. Sensitive code can however be protected from the effect of a forced termination by using the *protection* operator (inspired by [8]).

2.1 Syntax

The syntax of COWS, given in Table 1, is parameterized by three countable and pairwise disjoint sets: the set of *(killer) labels* (ranged over by $k, k', \ldots$), the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x, y, ...). The set of values is left unspecified; however, we assume that it includes the set of *names*, ranged over by n, m, ..., mainly used to represent partners and operations. The language is also parameterized by a set of *expressions*, ranged over by e, whose exact syntax is deliberately omitted; we just assume that expressions contain, at least, values and variables. Notably, killer labels are *not* (communicable) values. Notationally, we prefer letters $p, p', \ldots$ when we want to stress the use of a name as a partner, $o, o', \ldots$ when we want to stress the use of a name as an operation. We will use w to range over values and variables, u to range over names and variables, and d to range over killer labels, names and variables.

Services are structured activities built from basic activities, i.e. the empty activity $\mathbf{0}$, the kill activity $\mathbf{kill}(\_)$, the invoke activity $\_\cdot\_!\_$ and the receive activity $\_\cdot\_?\_$, by means of prefixing $\_.\_$, choice $\_ + \_$, parallel composition $\_ \mid \_$, protection $\{\!|\_|\!\}$, delimitation

s	::=		(services)
		$\mathbf{kill}(k)$	(kill)
	|	$u \bullet u'!\bar{e}$	(invoke)
	|	g	(input-guarded choice)
	|	$s \mid s$	(parallel composition)
	|	$\{\!\{s\}\!\}$	(protection)
	|	$[d]\, s$	(delimitation)
	|	$* s$	(replication)
g	::=		(input-guarded choice)
		$\mathbf{0}$	(nil)
	|	$p \bullet o?\bar{w}.s$	(request processing)
	|	$g + g$	(choice)

Table 1. COWS syntax

$[\_]\,\_$ and replication $*\,\_$. Notably, as in the Lπ [29], communication endpoints of receive activities are identified statically because their syntax only allows using names and not variables. We adopt the following conventions about the operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice.

Notation $\bar{\cdot}$ stands for tuples of objects, e.g. $\bar{x}$ is a compact notation for denoting the tuple of variables $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$). We assume that variables in the same tuple are pairwise distinct. All notations shall extend to tuples component-wise. In the sequel, we shall omit trailing occurrences of $\mathbf{0}$, writing e.g. $p \bullet o?\bar{w}$ instead of $p \bullet o?\bar{w}.\mathbf{0}$, and use $[d_1, \ldots, d_n]\, s$ in place of $[d_1] \ldots [d_n]\, s$.

The only *binding* construct is delimitation: $[d]\, s$ binds d in the scope s. The occurrence of a name/variable/label is *free* if it is not under the scope of a binder. We denote by $\mathrm{fd}(t)$ the set of names, variables and killer labels that occur free in a term t, and by $\mathrm{fk}(t)$ the set of free killer labels in t. Two terms are *alpha-equivalent* if one can be obtained from the other by consistently renaming bound names/variables/labels. As usual, we identify terms up to alpha-equivalence.

2.2 Operational semantics

The operational semantics of COWS is defined only for *closed* services, i.e. services without free variables/labels (similarly to many real compilers, we consider terms with free variables/labels as programming errors), but of course the rules also involve non-closed services (see e.g. the premises of rules *(del$_{sub}$)*, *(del$_{kill}$)* and *(del$_{pass}$)*).

Formally, the semantics is given in terms of a structural congruence and of a labelled transition relation. The *structural congruence* $\equiv$ identifies syntactically different services that intuitively represent the same service. It is defined as the least congruence relation induced by a given set of equational laws. We explicitly show in Table 2 the laws for replication, protection and delimitation, while omit the (standard) laws for the other operators stating that parallel composition is commutative, associative and has $\mathbf{0}$ as identity element, and that guarded choice enjoys the same properties and, addition-

$$
\begin{array}{rcll}
*\,\mathbf{0} & \equiv & \mathbf{0} & (repl_1)\\
*\,s & \equiv & s \mid *\,s & (repl_2)\\
\{\!|\mathbf{0}|\!\} & \equiv & \mathbf{0} & (prot_1)\\
\{\!|\,\{\!|s|\!\}\,|\!\} & \equiv & \{\!|s|\!\} & (prot_2)\\
\{\!|[d]\,s|\!\} & \equiv & [d]\,\{\!|s|\!\} & (prot_3)\\
[d]\,\mathbf{0} & \equiv & \mathbf{0} & (delim_1)\\
[d_1]\,[d_2]\,s & \equiv & [d_2]\,[d_1]\,s & (delim_2)\\
s_1 \mid [d]\,s_2 & \equiv & [d]\,(s_1 \mid s_2) \quad \text{if } d \notin \mathrm{fd}(s_1) \cup \mathrm{fk}(s_2) & (delim_3)
\end{array}
$$

Table 2. COWS structural congruence (excerpt of laws)

$$
\mathcal{M}(x, v) = \{x \mapsto v\} \qquad \mathcal{M}(v, v) = \emptyset \qquad \frac{\mathcal{M}(w_1, v_1) = \sigma_1 \qquad \mathcal{M}(\bar{w}_2, \bar{v}_2) = \sigma_2}{\mathcal{M}((w_1, \bar{w}_2), (v_1, \bar{v}_2)) = \sigma_1 \uplus \sigma_2}
$$

Table 3. Matching rules

ally, is idempotent. All the presented laws are straightforward. In particular, commutativity of consecutive delimitations implies that the order among the d_i in $[\langle d_1, \ldots, d_n \rangle]\, s$ is irrelevant, thus in the sequel we may use the simpler notation $[d_1, \ldots, d_n]\, s$. Notably, law $(delim_3)$ can be used to extend the scope of names (like a similar law in the π-calculus), thus enabling communication of restricted names, except when the argument d of the delimitation is a free killer label of s_2 (this avoids involving s_1 in the effect of a kill activity inside s_2).

To define the labelled transition relation, we need a few auxiliary functions. First, we exploit a function $[\![\_]\!]$ for evaluating *closed* expressions (i.e. expressions without variables): it takes a closed expression and returns a value. However, $[\![\_]\!]$ cannot be explicitly defined because the exact syntax of expressions is deliberately not specified.

Then, through the rules in Table 3, we define the partial function $\mathcal{M}(\_, \_)$ that permits performing *pattern-matching* on semi-structured data thus determining if a receive and an invoke over the same endpoint can synchronize. The rules state that two tuples match if they have the same number of fields and corresponding fields have matching values/variables. Variables match any value, and two values match only if they are identical. When tuples $\bar{w}$ and $\bar{v}$ do match, $\mathcal{M}(\bar{w}, \bar{v})$ returns a substitution for the variables in $\bar{w}$; otherwise, it is undefined. *Substitutions* (ranged over by σ) are functions mapping variables to values and are written as collections of pairs of the form $x \mapsto v$. Application of substitution σ to s, written $s \cdot \sigma$, has the effect of replacing every free occurrence of x in s with v, for each $x \mapsto v \in \sigma$, by possibly using alpha conversion for avoiding v to be captured by name delimitations within s. We use $|\sigma|$ to denote the number of pairs in σ and $\sigma_1 \uplus \sigma_2$ to denote the union of σ_1 and σ_2 when they have disjoint domains.

We also define a function, named $halt(\_)$, that takes a service s as an argument and returns the service obtained by only retaining the protected activities inside s. $halt(\_)$ is defined inductively on the syntax of services. The most significant case is $halt(\{\!|s|\!\}) = \{\!|s|\!\}$. In the other cases, $halt(\_)$ returns $\mathbf{0}$, except for parallel composition, delimitation

and replication operators, for which it acts as an homomorphism.

$$halt(\mathbf{kill}(k)) \;=\; halt(u_1 \bullet u_2!\bar{e}) \;=\; halt(g) \;=\; \mathbf{0}$$

$$halt(\{\!|s|\!\}) = \{\!|s|\!\} \qquad halt(s_1 \mid s_2) = halt(s_1) \mid halt(s_2)$$

$$halt([d]\, s) = [d]\, halt(s) \qquad halt(*\, s) = *\, halt(s)$$

Finally, in Table 4, we inductively define two predicates: $s{\downarrow_d}$ checks if s can immediately perform a $\mathbf{kill}(k)$; $s{\downarrow^{\ell}_{p\bullet o,\bar{v}}}$, with ℓ natural number, checks existence of potential communication conflicts, i.e. the ability of s of performing a receive activity matching $\bar{v}$ over the endpoint $p \bullet o$ that generates a substitution with fewer pairs than ℓ.

$$\mathbf{kill}(k)\downarrow_k \qquad \frac{s\downarrow_k \;\vee\; s'\downarrow_k}{s \mid s'\downarrow_k} \qquad \frac{s\downarrow_k}{\{\!|s|\!\}\downarrow_k} \qquad \frac{s\downarrow_k \quad d \neq k}{[d]\, s\downarrow_k} \qquad \frac{s\downarrow_k}{*\, s\downarrow_k}$$

$$\frac{|\mathcal{M}(\bar{w},\bar{v})| < \ell}{p\bullet o?\bar{w}.s\downarrow^{\ell}_{p\bullet o,\bar{v}}} \qquad \frac{s\downarrow^{\ell}_{p\bullet o,\bar{v}} \quad d \notin \{p,o\}}{[d]\, s\downarrow^{\ell}_{p\bullet o,\bar{v}}} \qquad \frac{s\downarrow^{\ell}_{p\bullet o,\bar{v}}}{\{\!|s|\!\}\downarrow^{\ell}_{p\bullet o,\bar{v}}}$$

$$\frac{g\downarrow^{\ell}_{p\bullet o,\bar{v}} \;\vee\; g'\downarrow^{\ell}_{p\bullet o,\bar{v}}}{g + g'\downarrow^{\ell}_{p\bullet o,\bar{v}}} \qquad \frac{s\downarrow^{\ell}_{p\bullet o,\bar{v}} \;\vee\; s'\downarrow^{\ell}_{p\bullet o,\bar{v}}}{s \mid s'\downarrow^{\ell}_{p\bullet o,\bar{v}}} \qquad \frac{s\downarrow^{\ell}_{p\bullet o,\bar{v}}}{*\, s\downarrow^{\ell}_{p\bullet o,\bar{v}}}$$

Table 4. Is there an active $\mathbf{kill}(k)$? / Are there conflicting receives along $p \bullet o$ matching $\bar{v}$?

The *labelled transition relation* $\xrightarrow{\alpha}$ is the least relation over services induced by the rules in Table 5, where label α is generated by the following grammar:

$$\alpha ::= \dagger k \quad | \quad (p \bullet o) \triangleleft \bar{v} \quad | \quad (p \bullet o) \triangleright \bar{w} \quad | \quad p \bullet o \lfloor\sigma\rfloor\, \bar{w}\, \bar{v} \quad | \quad \dagger$$

In the sequel, we use $d(\alpha)$ to denote the set of names, variables and killer labels occurring in α, except for $\alpha = p \bullet o \lfloor\sigma\rfloor\, \bar{w}\, \bar{v}$ for which we let $d(p \bullet o \lfloor\sigma\rfloor\, \bar{w}\, \bar{v}) = d(\sigma)$, where $d(\{x \mapsto v\}) = \{x, v\}$ and $d(\sigma_1 \uplus \sigma_2) = d(\sigma_1) \cup d(\sigma_2)$. The meaning of labels is as follows: $\dagger k$ denotes execution of a request for terminating a term from within the delimitation $[k]$, $(p \bullet o) \triangleleft \bar{v}$ and $(p \bullet o) \triangleright \bar{w}$ denote execution of invoke and receive activities over the endpoint $p \bullet o$, respectively, $p \bullet o \lfloor\sigma\rfloor\, \bar{w}\, \bar{v}$ (if $\sigma \neq \emptyset$) denotes execution of a communication over $p \bullet o$ with receive parameters $\bar{w}$ and matching values $\bar{v}$ and with substitution σ to be still applied, $\dagger$ and $p \bullet o \lfloor\emptyset\rfloor\, \bar{w}\, \bar{v}$ denote *computational steps* corresponding to taking place of forced termination and communication (without pending substitutions), respectively. Hence, a *computation* from a closed service s_0 is a sequence of connected transitions of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} s_3 \ldots$$

where, for each i, α_i is either $\dagger$ or $p \bullet o \lfloor\emptyset\rfloor\, \bar{w}\, \bar{v}$ (for some $p, o, \bar{w}$ and $\bar{v}$); services s_i, for each i, will be called *reducts* of s_0.

$$\mathbf{kill}(k) \xrightarrow{\dagger k} \mathbf{0} \quad \textit{(kill)} \qquad\qquad p \bullet o?\bar{w}.s \xrightarrow{(p \bullet o) \rhd \bar{w}} s \quad \textit{(rec)}$$

$$\frac{[\![\bar{e}]\!] = \bar{v}}{p \bullet o!\bar{e} \xrightarrow{(p \bullet o) \lhd \bar{v}} \mathbf{0}} \textit{(inv)} \qquad\qquad \frac{g_1 \xrightarrow{\alpha} s}{g_1 + g_2 \xrightarrow{\alpha} s} \textit{(choice)}$$

$$\frac{s \xrightarrow{p \bullet o \lfloor \sigma \uplus \{x \mapsto v'\} \rfloor \bar{w}\, \bar{v}} s'}{[x]\, s \xrightarrow{p \bullet o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s' \cdot \{x \mapsto v'\}} \textit{(del}_{sub}\textit{)} \qquad\qquad \frac{s \xrightarrow{\dagger k} s'}{[k]\, s \xrightarrow{\dagger} [k]\, s'} \textit{(del}_{kill}\textit{)}$$

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin d(\alpha) \quad s \downarrow_d \Rightarrow \alpha = \dagger, \dagger k}{[d]\, s \xrightarrow{\alpha} [d]\, s'} \textit{(del}_{pass}\textit{)} \qquad\qquad \frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}} \textit{(prot)}$$

$$\frac{s_1 \xrightarrow{(p \bullet o) \rhd \bar{w}} s_1' \qquad s_2 \xrightarrow{(p \bullet o) \lhd \bar{v}} s_2' \qquad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \qquad \neg(s_1 \mid s_2 \downarrow^{|\sigma|}_{p \bullet o, \bar{v}})}{s_1 \mid s_2 \xrightarrow{p \bullet o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \mid s_2'} \textit{(com)}$$

$$\frac{s_1 \xrightarrow{p \bullet o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \qquad \neg(s_2 \downarrow^{|\mathcal{M}(\bar{w}, \bar{v})|}_{p \bullet o, \bar{v}})}{s_1 \mid s_2 \xrightarrow{p \bullet o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \mid s_2} \textit{(par}_{conf}\textit{)} \qquad\qquad \frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid \mathit{halt}(s_2)} \textit{(par}_{kill}\textit{)}$$

$$\frac{s_1 \xrightarrow{\alpha} s_1' \qquad \alpha \neq (p \bullet o \lfloor \sigma \rfloor\, \bar{w}\, \bar{v}), \dagger k}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2} \textit{(par}_{pass}\textit{)} \qquad\qquad \frac{s \equiv s_1 \quad s_1 \xrightarrow{\alpha} s_2 \quad s_2 \equiv s'}{s \xrightarrow{\alpha} s'} \textit{(cong)}$$

Table 5. COWS operational semantics

We comment on salient points. Activity **kill**(k) forces termination of all unprotected parallel activities (rules *(kill)* and *(par$_{kill}$)*) inside an enclosing $[k]$, that stops the killing effect by turning the transition label $\dagger k$ into $\dagger$ (rule *(del$_{kill}$)*). Existence of such delimitation is ensured by the assumption that the semantics is only defined for closed services. Sensitive code can be protected from killing by putting it into a protection $\{\!|\_|\!\}$; this way, $\{\!|s|\!\}$ behaves like s (rule *(prot)*). Similarly, $[d]\, s$ behaves like s, except when the transition label α contains d or when a kill activity for d is active in s and α does not correspond to a kill activity (rule *(del$_{pass}$)*): in such cases the transition should be derived by using rules *(del$_{kill}$)* or *(del$_{sub}$)*. In other words, kill activities are executed *eagerly*. A service invocation can proceed only if the expressions in the argument can be evaluated (rule *(inv)*). A receive activity offers an invocable operation along a given partner name (rule *(rec)*). The execution of a receive permits to take a decision between alternative behaviours (rule *(choice)*). Communication can take place when two parallel services perform matching receive and invoke activities (rule *(com)*). Communication generates a substitution that is recorded in the transition label (for subsequent application), rather than a silent transition as in most process calculi. If more than one matching receive activity is ready to process a given invoke, then only the more defined one (i.e. the receive that generates

the 'smaller' substitution) progresses (rules *(com)* and *(par$_{conf}$)*). This mechanism permits to correlate different service communications thus implicitly creating interaction sessions and can be exploited to model the precedence of a service instance over the corresponding service specification when both can process the same request. When the delimitation of a variable x argument of a receive is encountered, i.e. the whole scope of the variable is determined, the delimitation is removed and the substitution for x is applied to the term (rule *(del$_{sub}$)*). Variable x disappears from the term and cannot be reassigned a value. Execution of parallel services is interleaved (rule *(par$_{pass}$)*), but when a kill activity or a communication is performed. Indeed, the former must trigger termination of all parallel services (according to rule *(par$_{kill}$)*), while the latter must ensure that the receive activity with greater priority progresses (rules *(com)* and *(par$_{conf}$)*). The last rule states that structurally congruent services have the same transitions.

2.3 Examples

We end this section with a few observations and examples aimed at clarifying the peculiarities of our formalism.

Communication of private names. Communication of private names is standard and exploits scope extension as in π-calculus. Notably, receive and invoke activities can interact only if both are in the scopes of the delimitations that bind the variables argument of the receive. Thus, to enable communication of private names, besides their scopes, we must possibly extend the scopes of some variables, as in the following example:

$$\begin{array}{lll} [x]\,(p \bullet o?\langle x\rangle.s \mid s') \mid [n]\,p \bullet o!\langle n\rangle & \equiv & (n \text{ fresh}) \\ [n]\,([x]\,(p \bullet o?\langle x\rangle.s \mid s') \mid p \bullet o!\langle n\rangle) & \equiv & \\ [n]\,[x]\,(p \bullet o?\langle x\rangle.s \mid s' \mid p \bullet o!\langle n\rangle) & \xrightarrow{p \bullet o\,\lfloor\emptyset\rfloor\,\langle x\rangle\,\langle n\rangle} & \\ [n]\,(s \mid s') \cdot \{x \mapsto n\} & & \end{array}$$

Notice that the substitution $\{x \mapsto n\}$ is applied to all terms delimited by $[x]$, not only to the continuation s of the service performing the receive. This is different from most process calculi and accounts for the global scope of variables. This very feature permits to easily model the *delayed input* of fusion calculus [34], which is instead difficult to express in π-calculus.

Delimited killer labels. We require killer labels to be delimited to avoid a single service be capable to stop all the other parallel services which would be unreasonable in a service-oriented setting. Indeed, suppose a service s can perform a **kill**(k) with k undelimited in s. The killing effect could not be stopped, thus, due to a transition labelled by $\dagger k$, the whole service s would be terminated (but for protected activities). Moreover, the effect of **kill**(k) could not be confined to s, thus, if there are other parallel services, the whole service composition might be terminated by **kill**(k).

Protected kill activity. The following simple example illustrates the effect of executing a kill activity within a protection block:

$$[k]\,(\{\!|s_1 \mid \{\!|s_2|\!\} \mid \mathbf{kill}(k)|\!\} \mid s_3) \mid s_4 \xrightarrow{\dagger} [k]\,\{\!|\{\!|s_2|\!\}|\!\} \mid s_4$$

where, for simplicity, we assume that $\mathit{halt}(s_1) = \mathit{halt}(s_3) = \mathbf{0}$. In essence, $\mathbf{kill}(k)$ terminates all parallel services inside delimitation $[k]$ (i.e. s_1 and s_3), except those that are protected at the same nesting level of the kill activity (i.e. s_2).

Interplay between communication and kill activity. Kill activities can break communication, as the following example shows:

$$p \bullet o!\langle n\rangle \mid [k]\,([x]\,p \bullet o?\langle x\rangle.s \mid \mathbf{kill}(k)) \xrightarrow{\dagger} p \bullet o!\langle n\rangle \mid [k]\,[x]\,\mathbf{0}$$

Communication can however be guaranteed by protecting the receive activity, as in

$$\begin{array}{ll} p \bullet o!\langle n\rangle \mid [k]\,([x]\,\{\!|p \bullet o?\langle x\rangle.s|\!\} \mid \mathbf{kill}(k)) & \xrightarrow{\dagger} \\ p \bullet o!\langle n\rangle \mid [k]\,[x]\,\{\!|p \bullet o?\langle x\rangle.s|\!\} & \equiv \\ [x]\,(p \bullet o!\langle n\rangle \mid [k]\,\{\!|p \bullet o?\langle x\rangle.s|\!\}) & \xrightarrow{p \bullet o\,\lfloor\emptyset\rfloor\,\langle x\rangle\,\langle n\rangle} \\ [k]\,\{\!|s \cdot \{x \mapsto n\}|\!\} & \end{array}$$

Conflicting receive activities. This example shows a *persistent service* (implemented by mean of replication), that, once instantiated, enables two conflicting receives:

$$\begin{array}{ll} *\,[x]\,(\,p_1 \bullet o?\langle x\rangle.s_1 \mid p_2 \bullet o?\langle x\rangle.s_2\,) \mid p_1 \bullet o!\langle v\rangle \mid p_2 \bullet o!\langle v\rangle & \xrightarrow{p_1 \bullet o\,\lfloor\emptyset\rfloor\,\langle x\rangle\,\langle v\rangle} \\ *\,[x]\,(\,p_1 \bullet o?\langle x\rangle.s_1 \mid p_2 \bullet o?\langle x\rangle.s_2\,) \mid s_1 \cdot \{x \mapsto v\} \mid p_2 \bullet o?\langle v\rangle.s_2 \cdot \{x \mapsto v\} \mid p_2 \bullet o!\langle v\rangle & \end{array}$$

Now, the persistent service and the created instance, being both able to receive the same tuple $\langle v\rangle$ along the endpoint $p_2 \bullet o$, compete for the request $p_2 \bullet o!\langle v\rangle$. However, our (prioritized) semantics, in particular rule *(com)* in combination with rule *($\mathit{par}_{\mathit{conf}}$)*, allows only the existing instance to evolve (and, thus, prevents creation of a new instance):

$$*\,[x]\,(\,p_1 \bullet o?\langle x\rangle.s_1 \mid p_2 \bullet o?\langle x\rangle.s_2\,) \mid s_1 \cdot \{x \mapsto v\} \mid s_2 \cdot \{x \mapsto v\}$$

Message correlation. Consider now uncorrelated receive activities executed by a same instance, like in the following service:

$$*\,[x]\,p_1 \bullet o_1?\langle x\rangle.[y]\,p_2 \bullet o_2?\langle y\rangle.s$$

The fact that the messages for operations o_1 and o_2 are uncorrelated implies that, e.g., if there are concurrent instances then successive invocations for a same instance can mix up and be delivered to different instances. If one thinks it right, this behaviour can be avoided simply by correlating successive messages by means of some correlation data, e.g. the first received value as in the following service:

$$*\,[x]\,p_1 \bullet o_1?\langle x\rangle.[y]\,p_2 \bullet o_2?\langle y, x\rangle.s$$

No conflict predicate. Rules *(com)* and *(par_{conf})* use the *no conflict* predicate $\_ \downarrow^{\ell}_{p \bullet o, \bar{v}}$ for checking the presence of concurrent conflicting receives. When these rules must be used to infer a transition, a preventive alpha conversion may be necessary. Indeed, condition $p \bullet o?\bar{w}'.s \downarrow^{\ell}_{p \bullet o, \bar{v}}$ might single out patterns that could not really match the transmitted values. These false alarms would block the inference (but allow us to stay on the 'safe' side).

For instance, consider the following term:

$$p \bullet o!\langle n \rangle \mid [x]\, p \bullet o?\langle x \rangle \mid [n]\, p \bullet o?\langle n \rangle \tag{1}$$

Apparently, both receive activities match the invoke activity, but only $p \bullet o?\langle x \rangle$ can synchronize with $p \bullet o!\langle n \rangle$, because the argument of $p \bullet o?\langle n \rangle$ is a restricted name, thus it is certainly different from the name transmitted by the invoke. However, if we try to naively infer the transition corresponding to the synchronization between $p \bullet o!\langle n \rangle$ and $p \bullet o?\langle x \rangle$, we fail due to rules *(com)* or *(par_{conf})*. In fact, $[n]\, p \bullet o?\langle n \rangle \downarrow^{|\{x \mapsto n\}|}_{p \bullet o, \langle n \rangle}$ holds because $\mathcal{M}(n, n)$ produces the substitution $\emptyset$, that is smaller than $\{x \mapsto n\}$, that is produced by $\mathcal{M}(x, n)$.

However, the wanted transition can be inferred by first applying alpha conversion. In fact, (1) can be written as follows:

$$p \bullet o!\langle n \rangle \mid [x]\, p \bullet o?\langle x \rangle \mid [n']\, p \bullet o?\langle n' \rangle$$

Now, it is clear that $p \bullet o?\langle n' \rangle$ is not a conflicting receive, because $\mathcal{M}(n', n)$ is undefined.

The same observations hold if in (1) we replace delimitation of n with that of p or of o. Also the term:

$$[n]\,(\, p \bullet o!\langle n \rangle \mid [x]\, p \bullet o?\langle x \rangle\,) \mid p \bullet o?\langle n \rangle$$

can be dealt with similarly: in all such cases, alpha conversion is necessary for inferring the correct transitions.

Similarities with Lπ (localised π-calculus [29]). Lπ is the variant of π-calculus closest to COWS. In fact, all Lπ constructs have a direct counterpart in COWS and is indeed possible to define an encoding that enjoys *operational correspondence*. More precisely, the syntax of Lπ processes is

$$P ::= \mathbf{0} \quad | \quad a(b).P \quad | \quad \bar{a}b \quad | \quad P \mid P \quad | \quad (\nu a)P \quad | \quad !a(b).P$$

with the constraint that in processes $a(b).P$ and $!a(b).P$ name b may not occur free in P in input position. For simplicity sake, we define the encoding only for Lπ processes such that their bound names are all distinct and different from the free ones (but it can be easily extended to deal with all processes). The crux of the encoding is mapping each Lπ channel name in a COWS communication endpoint, that is composed of variables if the channel name is bound by an input prefix (because in Lπ the name is used as a placeholder and COWS distinguishes between names and variables), and of names otherwise. The actual encoding function $\langle\!\langle \cdot \rangle\!\rangle_S$, that is parameterized by a set of names S, is defined by induction on the syntax of Lπ processes by the clauses in Table 6 (where the endpoint $p_a \bullet o_a$ is sometimes used in place of the tuple $\langle p_a, o_a \rangle$). The encoding of process P is given by service $\langle\!\langle P \rangle\!\rangle_S$ with $S = \emptyset$; as the encoding proceeds, S is used to record the names that have been freed and were initially bound by an input prefix.

$\langle\!\langle a \rangle\!\rangle_{S \cup \{a\}} = x_a \bullet y_a$	$\langle\!\langle a \rangle\!\rangle_S = p_a \bullet o_a \quad \text{if } a \notin S$
$\langle\!\langle \mathbf{0} \rangle\!\rangle_S = \mathbf{0}$	$\langle\!\langle a(b).P \rangle\!\rangle_S = [\langle\!\langle b \rangle\!\rangle_{S'}]\, \langle\!\langle a \rangle\!\rangle_{S'}?\langle\!\langle b \rangle\!\rangle_{S'}.\langle\!\langle P \rangle\!\rangle_{S'} \quad S' = S \cup \{b\}$
$\langle\!\langle (\nu a)P \rangle\!\rangle_S = [\langle\!\langle a \rangle\!\rangle_S]\, \langle\!\langle P \rangle\!\rangle_S$	$\langle\!\langle \bar{a}b \rangle\!\rangle_S = \langle\!\langle a \rangle\!\rangle_S\, !\langle\!\langle b \rangle\!\rangle_S$
$\langle\!\langle P_1 \mid P_2 \rangle\!\rangle_S = \langle\!\langle P_1 \rangle\!\rangle_S \mid \langle\!\langle P_2 \rangle\!\rangle_S$	$\langle\!\langle !a(b).P \rangle\!\rangle_S = *\langle\!\langle a(b).P \rangle\!\rangle_S$

Table 6. Lπ encoding

3 Modelling imperative and orchestration constructs

In this section, we present the encoding of some higher level imperative and orchestration constructs (mainly inspired by WS-BPEL). The encodings illustrate flexibility of COWS and somehow demonstrate expressiveness of the chosen set of primitives.

In the sequel, we will write $Z_{\bar{v}} \triangleq W$ to assign a symbolic name $Z_{\bar{v}}$ to the term W and to indicate the values $\bar{v}$ occurring within W. Thus, $Z_{\bar{v}}$ is a family of names, one for each tuple of values $\bar{v}$. In some examples, we shall write the symbolic name Z (instead of $Z_{\bar{v}}$) and elide the values $\bar{v}$ when they are not important or can be inferred from the context. We use $\hat{n}$ to stand for the endpoint $n_p \bullet n_o$. Sometimes, we write $\hat{n}$ for the tuple $\langle n_p, n_o \rangle$ and rely on the context to resolve any ambiguity.

3.1 Imperative constructs

Suppose to add a *matching with assignment* construct $[w = e]$ to COWS basic activities. Hence, we can also write services of the form $[w = e].s$ whose intended semantics is that, if w and e do match, a substitution is returned that will eventually assign to the variable in w the corresponding value of e, and service s can proceed. In COWS, this meaning can be rendered through the following encoding

$$\langle\!\langle [w = e].s \rangle\!\rangle = [\hat{m}]\, (\hat{m}!\langle e \rangle \mid \hat{m}?\langle w \rangle.\langle\!\langle s \rangle\!\rangle) \tag{2}$$

for $\hat{m}$ fresh. The new construct generalizes standard assignment because it allows values to occur on the left of =, in which case it behaves as a matching mechanism. Similarly, we can encode conditional choice as follows:

$$\langle\!\langle \textbf{if } (e) \textbf{ then } \{s_1\} \textbf{ else } \{s_2\} \rangle\!\rangle \;=\; [\hat{m}]\, (\hat{m}!\langle e \rangle \mid (\hat{m}?\langle \textbf{true} \rangle.\langle\!\langle s_1 \rangle\!\rangle + \hat{m}?\langle \textbf{false} \rangle.\langle\!\langle s_2 \rangle\!\rangle)\,) \tag{3}$$

where **true** and **false** are the values that can result from evaluation of e.

Like the receive activity, matching with assignment does not bind the variables on the left of =, thus it cannot reassign a value to them if a value has already been assigned. Therefore, the behaviour of matching with assignment may differ from standard assignment, even when the former uses only variables on the left of = as the latter does. For example, activity $[x = 1]$ will not necessarily generate substitution $\{x \mapsto 1\}$. In fact, when it will be executed, x could have been previously replaced by a value v in which case execution of the activity corresponds to checking if v and 1 do match. For similar

reasons, activity $[x = x + 1]$ does not have the effect of increasing the value of x by 1, but that of checking if the value of x and that of $x + 1$ do match, which always fails.

Standard variables (that can be repeatedly assigned) can be rendered as services providing 'read' and 'write' operations. When the service variable is initialized (i.e. the first time the 'write' operation is used), an instance is created that is able to provide the value currently stored. When this value must be updated, the current instance is terminated and a new instance is created which stores the new value (alike the memory cell service of [3]). Here is the specification:

$$
\begin{aligned}
Var_{\mathtt{x}} \triangleq\ & [x_v, x_a]\, \mathtt{x} \bullet o_{write}?\langle x_v, x_a\rangle. \\
& [\hat{m}]\, (\hat{m}!\langle x_v, x_a\rangle \mid \\
& \quad * \ [x, y]\, \hat{m}?\langle x, y\rangle. \\
& \quad (y!\langle\rangle \mid [k]\, (* \ [y']\, \mathtt{x} \bullet o_{read}?\langle y'\rangle.\{\!|y'!\langle x\rangle|\!\} \\
& \quad \quad \mid [x', y']\, \mathtt{x} \bullet o_{write}?\langle x', y'\rangle\, . \\
& \quad \quad \quad (\mathbf{kill}(k) \mid \{\!|\hat{m}!\langle x', y'\rangle|\!\}\,)\,)\,)\,)
\end{aligned}
$$

where $\mathtt{x}$ is a public partner name. Service $Var_{\mathtt{x}}$ provides two operations: o_{read}, for getting the current value; o_{write}, for replacing the current value with a new one. To access the service, a user must invoke these operations by providing a communication endpoint for the reply and, in case of o_{write}, the value to be stored. The o_{write} operation can be invoked along the public partner $\mathtt{x}$, which corresponds, the first time, to initialization of the variable. Thus, $Var_{\mathtt{x}}$ uses the delimited endpoint $\hat{m}$ in which to store the current value of the variable. This last feature is exploited to implement further o_{write} operations in terms of forced termination and re-instantiation. Delimitation $[k]$ is used to confine the effect of the kill activity to the current instance, while protection $\{\!|\_|\!\}$ avoids forcing termination of pending replies and of the invocation that will trigger the new instance.

Now, suppose temporarily that standard variables, ranged over by $X, Y, \ldots$, may occur in the syntax of COWS anywhere a variable can. We can remove them by using the following encodings. If e contains standard variables $X_1, \ldots X_n$, we can let

$$
\begin{aligned}
\langle\!\langle e\rangle\!\rangle_{\hat{m},\hat{n}} = [\hat{r}_1, \ldots, \hat{r}_n]\, (\ & \mathtt{x}_1 \bullet o_{read}!\hat{r}_1 \mid \cdots \mid \mathtt{x}_n \bullet o_{read}!\hat{r}_n \mid \\
& [x_1, \ldots, x_n]\, (\hat{r}_1?\langle x_1\rangle. \cdots .\hat{r}_n?\langle x_n\rangle. \\
& \quad \hat{m}!\langle e \cdot \{X_i \mapsto x_i\}_{i\in\{1,\ldots,n\}}, \hat{n}\rangle)\,)
\end{aligned}
$$

where $\{X_i \mapsto x_i\}$ denotes substitution of X_i with x_i, endpoint $\hat{m}$ returns the result of evaluating e, and endpoint $\hat{n}$ permits to receive an acknowledgment when the resulting value is assigned to a service variable (of course, we are assuming that $\hat{m}$, $\hat{n}$, $\hat{r}_i$ and x_i are fresh). With this encoding of expression evaluation, the encoding of matching with assignment becomes

$$
\begin{aligned}
\langle\!\langle [w = e].s\rangle\!\rangle &= [\hat{r}, \hat{n}]\, (\langle\!\langle e\rangle\!\rangle_{\hat{r},\hat{n}} \mid \hat{r}?\langle w, \hat{n}\rangle.\langle\!\langle s\rangle\!\rangle) \\
\langle\!\langle [X = e].s\rangle\!\rangle &= [\hat{n}]\, (\langle\!\langle e\rangle\!\rangle_{\mathtt{x}\bullet o_{write},\hat{n}} \mid \hat{n}?\langle\rangle.\langle\!\langle s\rangle\!\rangle)
\end{aligned}
$$

where w is a value v or a variable x, while X is a standard variable. In the sequel, we will write $[\bar{w} = \bar{e}]$, where $\bar{w} = \langle w_1, \ldots, w_n\rangle$ and $\bar{e} = \langle e_1, \ldots, e_n\rangle$, with $\bar{w}$ and $\bar{e}$ that may contain standard variables, for the sequence of assignments $[w_1 = e_1].\cdots.[w_n =$

$e_n]$. The encodings of the remaining constructs, where standard variables may directly occur, are

$$
\begin{aligned}
\langle\!\langle [X]\, s \rangle\!\rangle &= [\mathtt{x}]\,(\mathit{Var}_{\mathtt{x}} \mid \langle\!\langle s \rangle\!\rangle) \\
\langle\!\langle X \bullet u!\bar{e} \rangle\!\rangle &= [x, \hat{r}]\,(\mathtt{x} \bullet o_{read}!\hat{r} \mid \hat{r}?\langle x \rangle.\langle\!\langle x \bullet u!\bar{e} \rangle\!\rangle) \\
\langle\!\langle u \bullet X!\bar{e} \rangle\!\rangle &= [x, \hat{r}]\,(\mathtt{x} \bullet o_{read}!\hat{r} \mid \hat{r}?\langle x \rangle.\langle\!\langle u \bullet x!\bar{e} \rangle\!\rangle) \\
\langle\!\langle u \bullet u'!\langle e_1, \ldots, e_n \rangle \rangle\!\rangle &= [x_1, \hat{r}_1, \hat{m}_1, \ldots, x_n, \hat{r}_n, \hat{m}_n]\,(\langle\!\langle e_1 \rangle\!\rangle_{\hat{r}_1,\hat{m}_1} \mid \ldots \mid \langle\!\langle e_n \rangle\!\rangle_{\hat{r}_n,\hat{m}_n} \mid \\
&\qquad \hat{r}_1?\langle x_1, \hat{m}_1 \rangle.\cdots.\hat{r}_n?\langle x_n, \hat{m}_n \rangle.(u \bullet u'!\langle x_1, \ldots, x_n \rangle))\\
\langle\!\langle p \bullet o?\bar{w}.s \rangle\!\rangle &= [x_1, \ldots, x_n]\; p \bullet o?\bar{w} \cdot \{X_i \mapsto x_i\}_{i \in \{1,\ldots,n\}}. \\
&\qquad [\hat{r}_1, \ldots, \hat{r}_n]\,(\mathtt{x}_1 \bullet o_{write}!\langle x_1, \hat{r}_1 \rangle \mid \ldots \mid \mathtt{x}_n \bullet o_{write}!\langle x_n, \hat{r}_n \rangle \mid \\
&\qquad\qquad \hat{r}_1?\langle\rangle.\cdots.\hat{r}_n?\langle\rangle.\langle\!\langle s \rangle\!\rangle) \\
&\quad \text{if } \bar{w} \text{ contains standard variables } X_1, \ldots X_n, \text{ and } x_1, \ldots x_n \text{ are fresh} \\
\langle\!\langle \textstyle\sum_{i \in N} p_i \bullet o_i?\bar{w}_i.s_i \rangle\!\rangle &= [x_1^i, \ldots, x_{n_i}^i]_{\,i \in N} \textstyle\sum_{i \in N} p_i \bullet o_i?\bar{w}_i \cdot \{X_h^i \mapsto x_h^i\}_{h \in \{1,\ldots,n_i\}}. \\
&\qquad [\hat{r}_1, \ldots, \hat{r}_{n_i}]\,(\mathtt{x}_1^i \bullet o_{write}!\langle x_1^i, \hat{r}_1 \rangle \mid \ldots \mid \mathtt{x}_{n_i}^i \bullet o_{write}!\langle x_{n_i}^i, \hat{r}_{n_i} \rangle \mid \\
&\qquad\qquad \hat{r}_1?\langle\rangle.\cdots.\hat{r}_{n_i}?\langle\rangle.\langle\!\langle s_i \rangle\!\rangle) \\
&\quad \text{if } \bar{w}_i \text{ contains standard variables } X_1^i, \ldots X_{n_i}^i, \text{ and } x_1^i, \ldots x_{n_i}^i \text{ are} \\
&\quad \text{fresh, for each } i \in N
\end{aligned}
$$

This way, occurrences of standard variables can be completely removed. It is worth noticing that in the encoding of $p \bullet o?\bar{w}.s$, standard variables X_i occurring within $\bar{w}$ are replaced by auxiliary fresh variables x_i that are then used to update the corresponding standard variables. This means that standard variables are not used for correlation purposes. This choice is motivated by the fact that, differently from standard variables, correlation variables are write-once variables. The encoding of $\sum_{i \in N} p_i \bullet o_i?\bar{w}_i.s_i$ is complicated by the fact that in COWS choice operators must be guarded by receives, namely their branches cannot contain delimitation as a leading operator. Therefore, delimitations resulting from the encodings of the branches are moved outside the choice.

Sequential composition can be encoded alike in CCS [30, Chapter 8]. However, due to the asynchrony of invoke and kill activities, the notion of well-termination must be relaxed wrt CCS. Firstly, we settle that services may indicate their termination by exploiting the invoke activity $x_{done} \bullet o_{done}!\langle\rangle$, where x_{done} is a distinguished variable and o_{done} is a distinguished name. Secondly, we say that a service s is *well-terminating* if, for every reduct s' of s and fresh partner p, $s' \cdot \{x_{done} \mapsto p\} \xrightarrow{(p \bullet o_{done}) \lhd \langle\rangle}$ implies that if $s' \cdot \{x_{done} \mapsto p\} \xrightarrow{\alpha}$ then $\alpha = (p' \bullet o) \lhd \bar{v}$, for some p', o and $\bar{v}$. Notably, well-termination does not demand a service to terminate, but only that whenever the service can perform activity $p \bullet o_{done}!\langle\rangle$, then it terminates except for, possibly, some parallel pending invoke activities. As usual, the encoding of sequential composition relies on the assumption that all calculus operators (in particular, parallel composition) can be rendered as to preserve well-termination. Notice that services performing some kill activity are implicitly regarded as well-terminating since they cannot draw a termination signal. Finally, if we only consider well-terminating services, then, for a fresh p, we can let:

$$\langle\!\langle s_1; s_2 \rangle\!\rangle \;=\; [p]\,(\langle\!\langle s_1 \cdot \{x_{done} \mapsto p\} \rangle\!\rangle \mid p \bullet o_{done}?\langle\rangle.\langle\!\langle s_2 \rangle\!\rangle)$$

Of course, iterative constructs can be encoded by exploiting the previous encodings. In the sequel, we shall use the derived constructs with no more ado.

3.2 Web services

Web services over the Internet are applications of paramount importance for SOC. A web service is a process accessible through the web (and its related technologies) that creates one specific instance to serve each received request. Service instances can be executed concurrently or sequentially, and may also share (part of) the state. Typically, an instance is composed of concurrent threads that may offer a choice (i.e. a *pick activity* in web service jargon) among alternative receive activities. Web services could be able of receiving multiple messages in a statically unpredictable order and in such a way that the first incoming message triggers creation of a service instance which subsequent messages are routed to. This would require all those receive activities that can be immediately executed (according to [1], Section 16.3, there are *multiple start activities*) to share a non-empty set of variables (the so-called *correlation set*). The examples of this section illustrate how different services' execution modalities, e.g. concurrent vs. sequential execution, local vs. shared state, can be modelled in COWS (see [16] for another account of this topic in a process language for SOC).

In COWS, a web service can be modelled by a term of the form $*\,[\bar{d}\,]\,s$, where tuple $\bar{d}$ contains all the free variables of s and, possibly, a killer label to enable forcing termination of service instances. The use of replication enables providing as many concurrent instances as needed, while that of delimitation permits modelling the state (by restricting the scope of variables). This means that the previous term corresponds to a service whose instances are *concurrently executed without a shared state*. For instance, consider the following service definition:

$$*\,[x_1,\ldots,x_n]\,p\bullet o?\langle x_1\rangle.s$$

If we put it in parallel with the invocation $p\bullet o!\langle v_1\rangle$, the resulting system can evolve as follows:

$$\begin{array}{l} *\,[x_1,\ldots,x_n]\,p\bullet o?\langle x_1\rangle.s \mid p\bullet o!\langle v_1\rangle \;\rightarrow \\ *\,[x_1,\ldots,x_n]\,p\bullet o?\langle x_1\rangle.s \mid [x_2,\ldots,x_n]\,s\cdot\{x_1\mapsto v_1\} \end{array}$$

Each time an invocation is processed, a new service instance with private variables $x_1,\ldots,x_n$ is activated. For example, if we have two concurrent invocations (and there are not conflicting receives), we get

$$\begin{array}{l} *\,[x_1,\ldots,x_n]\,p\bullet o?\langle x_1\rangle.s \mid p\bullet o!\langle v_1\rangle \mid p\bullet o!\langle v_2\rangle \;\rightarrow\!\rightarrow \\ *\,[x_1,\ldots,x_n]\,p\bullet o?\langle x_1\rangle.s \mid [x_2,\ldots,x_n]\,s\cdot\{x_1\mapsto v_1\} \mid [x_2,\ldots,x_n]\,s\cdot\{x_1\mapsto v_2\} \end{array}$$

The resulting system is composed of the service definition and of two different instances, each with its own state.

To allow instances of a same service to be *concurrently executed while sharing (part of) the state*, we move the delimitations of the variables to be shared outside the scope of replication. Thus, if $x_1,\ldots,x_k$ are shared and $x_{k+1},\ldots,x_n$ are not, the previous example can be modified as follows:

$$[x_1,\ldots,x_k]\;*\;[x_{k+1},\ldots,x_n]\,p\bullet o?\langle x_1\rangle.s$$

After a parallel request $p \bullet o!\langle v_1 \rangle$ has been processed, we have:

$$[x_2, \ldots, x_k]\,(\,*\,[x_{k+1}, \ldots, x_n]\,p \bullet o?\langle v_1 \rangle.s \cdot \{x_1 \mapsto v_1\} \mid [x_{k+1}, \ldots, x_n]\,s \cdot \{x_1 \mapsto v_1\}\,)$$

In this case, since x_1 is shared both by the service definition and by its instances, new instances can be created only if the service definition receives requests along $p \bullet o$ with the same value (i.e. v_1) as the first invocation. In general, however, instantiation variables, such as x_1, are not shared, in order to allow service invocations with different arguments to trigger instance creation. To model this behaviour, we can simply leave instantiation variables within the scope of replication. Consider for example the term:

$$[x_2]\ *\ [x_1, x_3]\,p \bullet o?\langle x_1 \rangle.s$$

If requests $p \bullet o!\langle v_1 \rangle$ and $p \bullet o!\langle v_2 \rangle$ are put in parallel, the resulting system can evolve as follows:

$$[x_2]\ *\ [x_1, x_3]\,p \bullet o?\langle x_1 \rangle.s \mid p \bullet o!\langle v_1 \rangle \mid p \bullet o!\langle v_2 \rangle\ \rightarrow \rightarrow$$
$$[x_2]\,(\,*\,[x_1, x_3]\,p \bullet o?\langle x_1 \rangle.s \mid [x_3]\,s \cdot \{x_1 \mapsto v_1\} \mid [x_3]\,s \cdot \{x_1 \mapsto v_2\}\,)$$

After two computation steps, two instances, each with a local state (i.e. the variable x_3), are activated that share variable x_2.

Suppose now we want to model the fact that service instances can only be *sequentially executed without sharing a state*. We can exploit the sequential operator ';' (introduced in Section 3.1) and a fresh communication endpoint $\hat{r}$ (to signal termination of an instance). For example, consider the term:

$$[\hat{r}]\,(\,\hat{r}!\langle\rangle \mid *\,\hat{r}?\langle\rangle.[x_1, \ldots, x_n]\,((p \bullet o?\langle x_1 \rangle.s)\,;\,\hat{r}!\langle\rangle\,)\,)$$

After processing a parallel request $p \bullet o!\langle v_1 \rangle$, the resulting system becomes

$$[\hat{r}]\,(\,*\,\hat{r}?\langle\rangle.[x_1, \ldots, x_n]\,((p \bullet o?\langle x_1 \rangle.s)\,;\,\hat{r}!\langle\rangle\,) \mid [x_2, \ldots, x_n]\,(\,s \cdot \{x_1 \mapsto v_1\}\,;\,\hat{r}!\langle\rangle\,)\,)$$

Now, another request cannot be processed, and creation of a new service instance is disabled, until the existing instance emits the termination signal $\hat{r}!\langle\rangle$. This guarantees that at most one service instance is executed at a time.

Finally, by combining all the previous patterns, we can also model services whose instances are *sequentially executed and share (part of) a state*, as the following term shows:

$$[\hat{r}, x_2]\,(\,\hat{r}!\langle\rangle \mid *\,\hat{r}?\langle\rangle.[x_1, x_3]\,((p \bullet o?\langle x_1 \rangle.s)\,;\,\hat{r}!\langle\rangle\,)\,)$$

3.3 Fault and compensation handlers

In the SOC approach, fault handling is strictly related to the notion of *compensation*, namely the execution of specific activities (attempting) to reverse the effects of previously executed activities. We consider here a minor variant of the WS-BPEL compensation protocol. To begin with, we extend COWS syntax as shown in the upper part of

$$
\begin{array}{lll}
s ::= & \ldots & \text{(services)} \\
\mid & \mathbf{throw}(\phi) & \text{(fault generator)} \\
\mid & \mathbf{undo}(\iota) & \text{(compensate)} \\
\mid & [s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_\iota & \text{(scope)}
\end{array}
$$

$$
\begin{array}{l}
\langle\!\langle [s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_\iota \rangle\!\rangle_k \;= \\
\quad [\widehat{throw}]\ (\ \langle\!\langle \mathbf{catch}(\phi_1)\{s_1\}\rangle\!\rangle_k \mid \ \ldots\ \mid \langle\!\langle \mathbf{catch}(\phi_n)\{s_n\}\rangle\!\rangle_k \mid \\
\qquad\qquad [k_\iota]\,\langle\!\langle s\rangle\!\rangle_{k_\iota}\ ;(\ x_{done} \bullet o_{done}!\langle\rangle \mid [k']\,\{\!|\widehat{undo}?\langle\iota\rangle.\langle\!\langle s_c\rangle\!\rangle_{k'}|\!\}\ )\ ) \\
\\
\langle\!\langle \mathbf{catch}(\phi)\{s\}\rangle\!\rangle_k \;= \widehat{throw}?\langle\phi\rangle.[k']\,\langle\!\langle s\rangle\!\rangle_{k'} \\
\\
\langle\!\langle \mathbf{undo}(\iota)\rangle\!\rangle_k \;= \widehat{undo}!\langle\iota\rangle \mid x_{done} \bullet o_{done}!\langle\rangle \\
\\
\langle\!\langle \mathbf{throw}(\phi)\rangle\!\rangle_k \;= \{\!|\widehat{throw}!\langle\phi\rangle|\!\} \mid \mathbf{kill}(k)
\end{array}
$$

Table 7. Syntax and encoding of fault and compensation handling

Table 7. The *scope* activity $[s : \mathbf{catch}(\phi_1)\{s_1\} : \ldots : \mathbf{catch}(\phi_n)\{s_n\} : s_c]_\iota$ permits explicitly grouping activities together. The declaration of a scope activity contains a unique scope identifier ι, a service s representing the normal behaviour, an optional list of fault handlers, and a compensation handler s_c. The *fault generator* activity $\mathbf{throw}(\phi)$ can be used by a service to rise a fault signal ϕ. This signal will trigger execution of activity s', if a construct of the form $\mathbf{catch}(\phi)\{s'\}$ exists within the same scope. The *compensate* activity $\mathbf{undo}(\iota)$ can be used to invoke a compensation handler of an inner scope named ι that has already completed normally (i.e. without faulting). Compensation can only be invoked from within a fault or a compensation handler. As in WS-BPEL, we fix two syntactic constraints: handlers do not contain scope activities and for each $\mathbf{undo}(\iota)$ occurring in a service there exists at least an inner scope ι.

Now we show that fault and compensation handling can be easily encoded in COWS, thus it is not necessary to extend its syntax. The most interesting cases of the encoding are shown in the lower part of Table 7 (in the remaining cases, the encoding acts as an homomorphism). The two distinguished endpoints $\widehat{throw}$ and $\widehat{undo}$ are used for exchanging fault and compensation signals, respectively. Each scope identifier ι or fault signal ϕ can be used to activate scope compensation or fault handling, respectively.

The encoding $\langle\!\langle \cdot \rangle\!\rangle_k$ is parameterized by the identifier k of the closest enclosing scope, if any. The parameter is used when encoding a fault generator, to launch a kill activity that forces termination of all the remaining activities of the enclosing scope, and when encoding a scope, to delimit the field of action of inner kill activities. The compensation handler s_c of scope ι is installed when the normal behaviour s successfully completes, but it is activated only when signal $\widehat{undo}!\langle i\rangle$ occurs. Similarly, if during normal execution a fault ϕ occurs, a signal $\widehat{throw}!\langle\phi\rangle$ triggers execution of the corresponding fault handler (if any). Installed compensation handlers are protected from killing by means of $\{\!|\_|\!\}$. Notably, the compensate activity can immediately terminate (thus enabling possible sequential compositions); this, of course, does not mean that the corresponding handler is terminated.

Other kinds of faults could be handled similarly. For example, an invoke activity $\mathbf{inv}(u_1 \cdot u_2, \bar{e})$ that generates a fault ϕ_{undef} when its argument $\bar{e}$ is undefined, could be

$$
\begin{array}{rcll}
s & ::= & \ldots \mid [\overline{fl}]\, ls \mid \sum_{i\in I} p_i \bullet o_i?\bar{w}_i.s_i & \text{(services)} \\
ls & ::= & (jc) \overset{sjf}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e}) \mid s \Rightarrow (\overline{fl}, \bar{e}) \mid ls \mid ls & \text{(linked services)} \\
jc & ::= & \mathbf{true} \mid \mathbf{false} \mid fl \mid \neg jc \mid jc \vee jc \mid jc \wedge jc & \text{(join conditions)} \\
sjf & ::= & yes \mid no & \text{(supp. join failure)}
\end{array}
$$

$$
\langle\!\langle [\overline{fl}]\, ls \rangle\!\rangle = [\overline{fl}]\, \langle\!\langle ls \rangle\!\rangle \qquad \langle\!\langle ls_1 \mid ls_2 \rangle\!\rangle = \langle\!\langle ls_1 \rangle\!\rangle \mid \langle\!\langle ls_2 \rangle\!\rangle \qquad \langle\!\langle s \Rightarrow (\overline{fl}, \bar{e}) \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle; [\overline{fl} = \bar{e}]
$$

$$
\langle\!\langle (jc) \overset{yes}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e}) \rangle\!\rangle = \mathbf{if}\ (jc)\ \mathbf{then}\ \{\langle\!\langle s \rangle\!\rangle; [\overline{fl} = \bar{e}]\}\ \mathbf{else}\ \{[outLinkOf(s) = \overline{\mathbf{false}}]\}
$$

$$
\langle\!\langle (jc) \overset{no}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e}) \rangle\!\rangle = \mathbf{if}\ (jc)\ \mathbf{then}\ \{\langle\!\langle s \rangle\!\rangle; [\overline{fl} = \bar{e}]\}\ \mathbf{else}\ \{\mathbf{throw}(\phi_{join\_f})\}
$$

$$
\begin{array}{l}
\langle\!\langle \sum_{i\in\{1..n\}} p_i \bullet o_i?\bar{w}_i.s_i \rangle\!\rangle = p_1 \bullet o_1?\bar{w}_1.[\bigcup_{j\in\{2..n\}} outLinkOf(s_j) = \overline{\mathbf{false}}].\langle\!\langle s_1 \rangle\!\rangle \\
\qquad\qquad + \ldots + p_n \bullet o_n?\bar{w}_n.[\bigcup_{j\in\{1..n-1\}} outLinkOf(s_j) = \overline{\mathbf{false}}].\langle\!\langle s_n \rangle\!\rangle
\end{array}
$$

Table 8. Syntax and encoding of flow graphs

encoded as follows:

$$
\begin{array}{rl}
\langle\!\langle \mathbf{inv}(u_1 \bullet u_2, \bar{e}) \rangle\!\rangle = & [k]\,(u_1 \bullet u_2!\bar{e} \mid [\mathbf{true} = (\bar{e} == undef)].\langle\!\langle \mathbf{throw}(\phi_{undef}) \rangle\!\rangle_k) \\
& \mid \langle\!\langle \mathbf{catch}(\phi_{undef})\{s_{undef}\} \rangle\!\rangle
\end{array}
$$

where k is fresh and, for simplicity, we assume that the fault handler for ϕ_{undef} is defined locally to the invoke activity. By the way, we have elided the parameter from the encoding of $\mathbf{catch}(\phi_{undef})\{s_{undef}\}$ because it is never used.

3.4 Flow graphs

In business process management, flow graphs[1] provide a direct and intuitive way to structure workflow processes, where activities executed in parallel can be synchronized by settling dependencies, called (flow) links, among them. At the beginning of a parallel execution, all involved links are inactive and only those activities with no synchronization dependencies can execute. Once all incoming links of an activity are active (i.e., they have been assigned either a positive or negative state), a guard, called *join condition*, is evaluated. When an activity terminates, the status of the outgoing links, which can be positive, negative or undefined, is determined through evaluation of a *transition condition*. When an activity in the flow graph cannot execute (i.e., the join condition fails), a *join failure* fault is emitted to signal that some activities have not completed. An attribute called 'suppress join failure' can be set to *yes* to ensure that join condition failures do not throw the join failure fault (this way obtaining the so-called *Dead-Path Elimination* effect [1]).

To express the constructs above, we extend the syntax of COWS as illustrated in the upper part of Table 8. A *flow graph activity* $[\overline{fl}]\, ls$ is a delimited *linked service*, where

[1] Here, we refer to the corresponding notion of WS-BPEL rather than to similar synchronization constructs of some process calculi (see e.g. [21]) or to the homonymous graphical notation used for representing processes and their interconnection structure (see, e.g., [30, 31]).

the activities within ls can synchronize by means of the flow links in $\overline{fl}$, rendered as (boolean) variables. A linked service is a service equipped with a set of incoming flow links that forms the *join condition*, and a set of outgoing flow links that represents the *transition condition*. Incoming flow links and join condition are denoted by $(jc) \overset{sjf}{\Rightarrow}$. Outgoing links are represented by $\Rightarrow (\overline{fl}_{i\in I}, \bar{e}_{i\in I})$ where each pair (fl_i, e_i) is composed of a flow link fl_i and the corresponding transition (boolean) condition e_i. Attribute sjf permits suppressing possible join failures. Input-guarded summation replaces binary choice, because we want all the branches of a multiple choice to be considered at once.

Again, we show that in fact it is not necessary to extend the syntax because flow graphs can be easily encoded by exploiting the capability of COWS of modelling a state shared among a group of activities. The most interesting cases of the encoding are shown in the lower part of Table 8. The encoding uses the auxiliary function $outLinkOf(s)$, that returns the tuple of outgoing links in s and is inductively defined as follows:

$$
\begin{array}{l}
outLinkOf([\overline{fl}]\, ls) = outLinkOf(ls) \\
outLinkOf(\sum_{i\in\{1..n\}} p_i \bullet o_i?\bar{w}_i.s_i) = outLinkOf(s_1), \ldots, outLinkOf(s_n) \\
outLinkOf((jc) \overset{sjf}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})) = outLinkOf(s)\,,\ \overline{fl} \\
outLinkOf(s \Rightarrow (\overline{fl}, \bar{e})) = outLinkOf(s)\,,\ \overline{fl} \\
outLinkOf(ls_1 \mid ls_2) = outLinkOf(ls_1)\,,\ outLinkOf(ls_2) \\
outLinkOf(\mathbf{0}) = outLinkOf(\mathbf{kill}(k)) = outLinkOf(u_1 \bullet u_2!\bar{e}) = \langle\rangle \\
outLinkOf(s_1 \mid s_2) = outLinkOf(s_1)\,,\ outLinkOf(s_2) \\
outLinkOf(\{\!|s|\!\}) = outLinkOf([d]\, s) = outLinkOf(*\, s) = outLinkOf(s)
\end{array}
$$

Basically, flow graphs are rendered as delimited services, while flow links are rendered as variables. A join condition is encoded as a boolean condition within a conditional construct, where the transition conditions are rendered as the assignment $[\overline{fl} = \bar{e}]$. In case attribute 'suppress join failure' is set to *no*, a join condition failure produces a fault signal that can be caught by a proper fault handler. Choice among (linked) services is implemented in such a way that, when a branch is selected, the links outgoing from the activities of the discarded branches are set to $false$. The same rationale underlies the new encoding of conditional choice that becomes as follows

$$
\begin{array}{rl}
\langle\!\langle \mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\} \rangle\!\rangle = & \mathbf{if}\ (e)\ \mathbf{then}\ \{[outLinkOf(s_2) = \overline{\mathbf{false}}].\langle\!\langle s_1 \rangle\!\rangle\} \\
& \mathbf{else}\ \{[outLinkOf(s_1) = \overline{\mathbf{false}}].\langle\!\langle s_2 \rangle\!\rangle\}
\end{array}
$$

4 Examples

In this section we show two applications of our framework. The former is an example of a service inspired by the well-known game *Rock/Paper/Scissors*, the latter is an example of a shipping service from the WS-BPEL specification [1].

4.1 Rock/Paper/Scissors Service

Consider the following service:

$$\begin{aligned} rps \triangleq\ & *\,[x_{champ\_res}, x_{chall\_res}, x_{id}, x_{thr\_1}, x_{thr\_2}, x_{win}] \\ & (p_{champ} \bullet o_{throw}?\langle x_{champ\_res}, x_{id}, x_{thr\_1}\rangle.x_{champ\_res} \bullet o_{win}!\langle x_{id}, x_{win}\rangle \mid \\ & p_{chall} \bullet o_{throw}?\langle x_{chall\_res}, x_{id}, x_{thr\_2}\rangle.x_{chall\_res} \bullet o_{win}!\langle x_{id}, x_{win}\rangle \mid \\ & Assign) \end{aligned}$$

The task of service rps is to collect two throws, stored in x_{thr_1} and x_{thr_2}, from two different participants, the current champion and the challenger, assign the winner to x_{win} and then send the result back to the two players. The service receives throws from the players via two distinct endpoints, characterized by operation o_{throw} and partners p_{champ} and p_{chall}. The service is of kind "request-response" and is able to serve challenges coming from any pairs of participants. The players are required to provide the partner names, stored in x_{champ_res} and x_{chall_res}, which they will use to receive the result. A challenge is uniquely identified by a challenge-id, here stored in x_{id}, that the partners need to provide when sending their throws. Partner throws arrive randomly. Thus, when a throw is processed, for instance the challenging one, it must be checked if a service instance with the same challenge-id already exists or not. We assume that $Assign$ implements the rules of the game and thus, by comparing x_{thr_1} and x_{thr_2}, assigns the winner of the match by producing the assignment $[x_{win} = x_{champ\_res}]$ or $[x_{win} = x_{chall\_res}]$. Thus, we have

$$\begin{aligned} Assign \triangleq\ & \textbf{if}\ (x_{thr\_1} == \text{“}rock\text{”} \& x_{thr\_2} == \text{“}scissors\text{”}) \\ & \quad \textbf{then}\ \{\ [x_{win} = x_{champ\_res}]\ \} \\ & \quad \textbf{else}\ \{\ \textbf{if}\ (x_{thr\_1} == \text{“}rock\text{”} \& x_{thr\_2} == \text{“}paper\text{”}) \\ & \qquad \textbf{then}\ \{\ [x_{win} = x_{chall\_res}]\ \} \\ & \qquad \textbf{else}\ \{\ \ \ldots \\ & \qquad\quad \} \\ & \quad \} \end{aligned}$$

A partner may simultaneously play multiple challenges by using different challenge identifiers as a means to correlate messages received from the server. E.g., the partner

$$\begin{aligned} & (p_{chall} \bullet o_{throw}!\langle p'_{chall}, 0, \text{“}rock\text{”}\rangle \mid [x]\, p'_{chall} \bullet o_{win}?\langle 0, x\rangle.s_0) \mid \\ & (p_{chall} \bullet o_{throw}!\langle p'_{chall}, 1, \text{“}paper\text{”}\rangle \mid [y]\, p'_{chall} \bullet o_{win}?\langle 1, y\rangle.s_1) \end{aligned}$$

is guaranteed that the returned results will be correctly delivered to the corresponding continuations.

Let us now consider the following match of rock/paper/scissors identified by the correlation value 0:

$$\begin{aligned} s \triangleq\ & rps \mid p_{champ} \bullet o_{throw}!\langle p'_{champ}, 0, \text{“}rock\text{”}\rangle \mid [x]\, p'_{champ} \bullet o_{win}?\langle 0, x\rangle.s_{champ} \\ & \mid p_{chall} \bullet o_{throw}!\langle p'_{chall}, 0, \text{“}scissors\text{”}\rangle \mid [y]\, p'_{chall} \bullet o_{win}?\langle 0, y\rangle.s_{chall} \end{aligned}$$

where p'_{champ} and p'_{chall} denote the players' partner names. Figure 1 shows a customized UML sequence diagram depicting the above scenario. The champion and a challenger

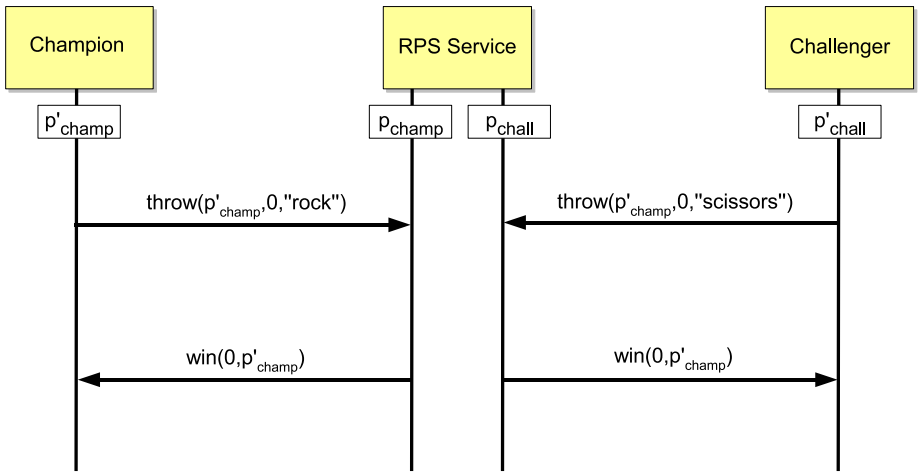

Fig. 1. Graphical representation of a Rock/Paper/Scissors service scenario

participate to the match, play their throws (i.e. "*rock*" and "*scissors*"), wait for the resulting winner, and (possibly) use this result in their continuation processes (i.e. s_{champ} and s_{chall}). Here is a computation produced by selecting the champion's throw:

$$
\begin{array}{l}
s \xrightarrow{p_{champ} \bullet o_{throw}\lfloor\emptyset\rfloor\langle x_{champ\_res}, x_{id}, x_{thr\_1}\rangle\langle p'_{champ}, 0, \text{“}rock\text{”}\rangle} \\
rps \mid [x_{chall\_res}, x_{thr\_2}, x_{win}]\,(\; p'_{champ} \bullet o_{win}!\langle 0, x_{win}\rangle \mid \\
\qquad p_{chall} \bullet o_{throw}?\langle x_{chall\_res}, 0, x_{thr\_2}\rangle.x_{chall\_res} \bullet o_{win}!\langle 0, x_{win}\rangle \mid \\
\qquad Assign \cdot \{x_{champ\_res} \mapsto p'_{champ}, x_{id} \mapsto 0, x_{thr\_1} \mapsto \text{“}rock\text{”}\}\;) \\
\quad \mid [x]\, p'_{champ} \bullet o_{win}?\langle 0, x\rangle.s_{champ} \\
\quad \mid p_{chall} \bullet o_{throw}!\langle p'_{chall}, 0, \text{“}scissors\text{”}\rangle \mid [y]\, p'_{chall} \bullet o_{win}?\langle 0, y\rangle.s_{chall} \quad \triangleq \quad s'
\end{array}
$$

Below, the challenger's throw is consumed by the existing instance:

$$
\begin{array}{l}
s' \xrightarrow{p_{chall} \bullet o_{throw}\lfloor\emptyset\rfloor\langle x_{chall\_res}, 0, x_{thr\_2}\rangle\langle p'_{chall}, 0, \text{“}scissors\text{”}\rangle} \\
rps \mid [x_{win}]\,(\; p'_{champ} \bullet o_{win}!\langle 0, x_{win}\rangle \mid p'_{chall} \bullet o_{win}!\langle 0, x_{win}\rangle \mid \\
\qquad Assign \cdot \{x_{champ\_res} \mapsto p'_{champ}, x_{id} \mapsto 0, x_{thr\_1} \mapsto \text{“}rock\text{”}, \\
\qquad\qquad x_{chall\_res} \mapsto p'_{chall}, x_{thr\_2} \mapsto \text{“}scissors\text{”}\}\;) \\
\quad \mid [x]\, p'_{champ} \bullet o_{win}?\langle 0, x\rangle.s_{champ} \\
\quad \mid [y]\, p'_{chall} \bullet o_{win}?\langle 0, y\rangle.s_{chall}
\end{array}
$$

In the computation above, rules *(com)* and (par_{conf}) allow only the existing instance to evolve (thus, creation of a new conflicting instance is avoided). Once $Assign$ determines that p_{champ} won, the substitutive effects of communication transforms the system as

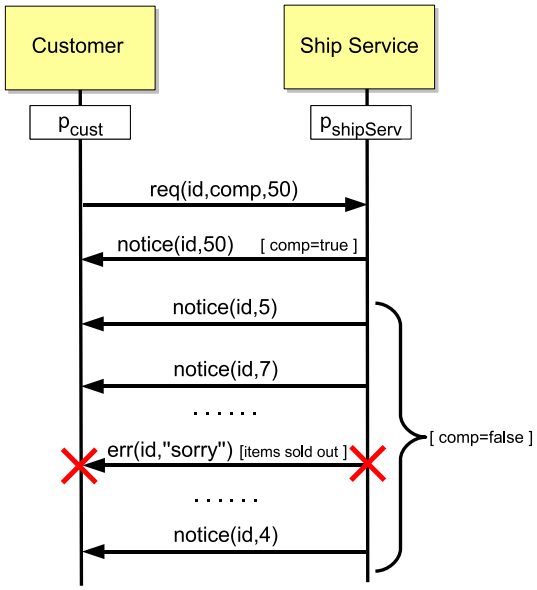

Fig. 2. Graphical representation of a shipping service scenario

follows:

$$\begin{aligned} s'' \triangleq\ & rps \mid p'_{champ} \bullet o_{win}!\langle 0, p_{champ}\rangle \mid p'_{chall} \bullet o_{win}!\langle 0, p_{champ}\rangle \\ & \mid [x]\, p'_{champ} \bullet o_{win}?\langle 0, x\rangle.s_{champ} \\ & \mid [y]\, p'_{chall} \bullet o_{win}?\langle 0, y\rangle.s_{chall} \end{aligned}$$

At the end, the name of the resulting winner is sent to both participants as shown by the following computation:

$$\begin{array}{l} s'' \xrightarrow{p'_{champ} \bullet o_{win}\, \lfloor\emptyset\rfloor\, \langle 0,x\rangle\, \langle 0,p_{champ}\rangle} \xrightarrow{p'_{chall} \bullet o_{win}\, \lfloor\emptyset\rfloor\, \langle 0,y\rangle\, \langle 0,p_{champ}\rangle} \\ rps \mid s_{champ} \cdot \{x \mapsto p_{champ}\} \mid s_{chall} \cdot \{y \mapsto p_{champ}\} \end{array}$$

4.2 Shipping Service

We consider an extended version of the shipping service described in the official specification of WS-BPEL [1] (Section 15.1). This example covers most of the language features we are interested in, including correlation sets, variables, flow control structures, fault and compensation handling.

The shipping service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The service offers two types of shipment: shipments where the items are held and shipped together and shipments where the items are shipped piecemeal until all of the order is fulfilled. Figure 2 illustrates a scenario where the shipping service interacts with a customer service. The shipping service is

specified in COWS as follows:

$$
\begin{array}{l}
*\,[x_{id}, x_{complete}, x_{itemsTot}] \\
\quad p_{shipServ} \cdot o_{req}?\langle x_{id}, x_{complete}, x_{itemsTot}\rangle. \\
\quad \textbf{if}\ (x_{complete})\ \textbf{then}\ \{\ p_{cust} \cdot o_{notice}!\langle x_{id}, x_{itemsTot}\rangle\ \} \\
\qquad\qquad \textbf{else}\ \{\ [\ s\ :\ \textbf{catch}(\phi_{noItems})\{\ \textbf{undo}(\iota_{price})\ |\ p_{cust} \cdot o_{err}!\langle x_{id}, \text{“}sorry\text{”}\rangle\ \}\ :\ \mathbf{0}\ ]_{\iota}\ \}
\end{array}
$$

where the normal behaviour s is

$$
\begin{array}{l}
[x_{ratio}]\,(\ [\ s_{shipPriceCalc}\ :\ s_{shipPriceComp}\ ]_{\iota_{price}}\ ; \\
\qquad [o_{while}]\,(\ p_{shipServ} \cdot o_{while}!\langle 0\rangle\ | \\
\qquad\qquad *\,[x_c]\ \ p_{shipServ} \cdot o_{while}?\langle x_c\rangle. \\
\qquad\qquad\qquad \textbf{if}\ (x_c < x_{itemsTot})\ \textbf{then}\ \{\ [x_{itemsCount}] \\
\qquad\qquad\qquad\qquad [x_{itemsCount} = rand()]. \\
\qquad\qquad\qquad\qquad \textbf{if}\ (x_{itemsCount} \leqslant 0)\ \textbf{then}\ \{ \\
\qquad\qquad\qquad\qquad\qquad [x_{ratio} = x_c\ /\ x_{itemsTot}].\textbf{throw}(\phi_{noItems})\ \} \\
\qquad\qquad\qquad\qquad \textbf{else}\ \{\ p_{cust} \cdot o_{notice}!\langle x_{id}, x_{itemsCount}\rangle\ | \\
\qquad\qquad\qquad\qquad\qquad p_{shipServ} \cdot o_{while}!\langle x_c + x_{itemsCount}\rangle\ \}\ \} \\
\qquad\qquad\qquad \textbf{else}\ \{\mathbf{0}\} \\
\qquad\qquad)\ \\
\qquad)
\end{array}
$$

$p_{shipServ}$ is the partner associated to the shipping service, o_{req} is the operation used to receive the shipping request, and $\langle x_{id}, x_{complete}, x_{itemsTot}\rangle$ is the tuple of variables used for the request shipping message: x_{id} stores the order identifier, that is used to correlate the ship notice(s) with the ship order, $x_{complete}$ stores a boolean indicating whether the order is to be shipped complete or not, and $x_{itemsTot}$ stores the total number of items in the order. Shipping notices and error messages to customers are sent using partner p_{cust} and operations o_{notice} and o_{err}, respectively. A notice message is a tuple composed of the order identifier and the number of items in the shipping notice. When partial shipment is acceptable, x_c is used to record the number of items already shipped. Replication and the internal operation o_{while} are used to model iteration.

Our example extend that in [1] by allowing the service to generate a fault in case the shipping company has ended the stock of items (this is modelled by function *rand*() returning an integer less or equal to 0). The fault is handled by sending an error message to the customer and by compensating the inner scope ι_{price}, that has already completed successfully. Function *rand*() returns a random integer number and represents an internal interaction with a back-end system. For the sake of simplicity, we don't describe this interaction. Moreover, we don't show services $s_{shipPriceCalc}$ and $s_{shipPriceComp}$. Basically, the former calculates the shipping price according to the value assigned to $x_{itemsTot}$ and sends the result to the accounts department. The latter is the corresponding compensation activity, that sends information about the non-shipped items to the accounts department and sends a refund to the customer according to the ratio (stored in x_{ratio}) between the shipped items (stored in x_c) and the required ones (stored in $x_{itemsTot}$).

5 Encoding other formal languages for orchestration

We present here the encodings in COWS of three orchestration languages: Orc, SCC and ws-CALCULUS. The first language is based on the functional paradigm and has already proved to be capable of expressing the most common workflow patterns; the second one is a language for SOC centered on the explicit modelling of services' interaction sessions and their dynamic creation; the last one exploits message correlation and turned out to be suitable to model in a quite direct way the semantics of whole WS-BPEL [25, 26].

5.1 Encoding Orc

We present here the encoding of Orc [32], a recently proposed task orchestration language with applications in workflow, business process management, and web service orchestration. We will show that the encoding enjoys a property of operational correspondence. This is another sign of COWS expressiveness because it is known that Orc can express the most common workflow patterns identified in [35]. Orc syntax is:

$$(\mathit{Expressions}) \quad f, g \ ::= \ \mathbf{0} \mid S(w) \mid E(w) \mid f > x > g \mid f \mid g \mid g \ \mathbf{where}\ x :\in f$$

$$(\mathit{Parameters}) \quad w \ ::= \ x \mid v$$

where S ranges over *site* names, E over *expression* names, x over variables, and v over values. Each expression name E has a unique declaration of the form $E(x) \triangleq f$. Expressions can be composed by means of sequential composition $\cdot > x > \cdot$, symmetric parallel composition $\cdot \mid \cdot$, and asymmetric parallel composition $\cdot$ **where** $x :\in \cdot$ starting from the elementary expressions $\mathbf{0}$, $S(w)$ (site call) and $E(w)$ (expression call). The variable x is *bound* in g for the expressions $f > x > g$ and g **where** $x :\in f$. Variable x is *free* in f if it is not bound in f. We use $fv(f)$ to denote the set of variables which occur free in f.

Evaluation of expressions may call a number of sites and returns a (possibly empty) stream of values. The asynchronous operational semantics of Orc is given by the labelled transition relation $\overset{l}{\hookrightarrow}$ defined in Table 9, where label τ indicates an internal event while label $!v$ indicates the value v resulting from evaluating an expression. A site call can progress only when the actual parameter is a value (rule *(SiteCall)*); it elicits one response. While site calls use a call-by-value mechanism, expression calls use a call-by-name mechanism (rule *(Def)*), namely the actual parameter replaces the formal one and then the corresponding expression is evaluated. Symmetric parallel composition $f \mid g$ consists of concurrent evaluations of f and g (rules *(Sym1)* and *(Sym2)*). Sequential composition $f > x > g$ activates a concurrent copy of g with x replaced by v, for each value v returned by f (rules *(Seq1)* and *(Seq2)*). Asymmetric parallel composition g **where** $x :\in f$ prunes threads selectively. It starts in parallel both f and the part of g that does not need x (rules *(Asym1)* and *(Asym2)*). The first value returned by f is assigned to x and the continuation of f and all its descendants are then terminated (rule *(Asym3)*).

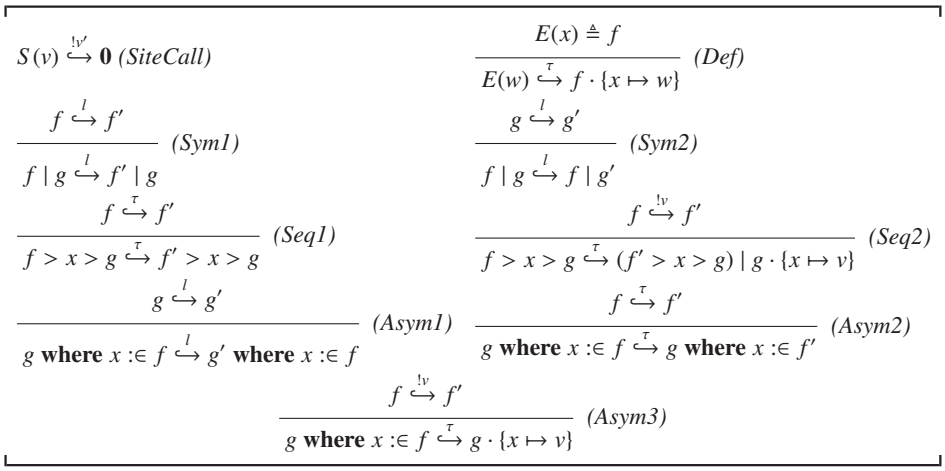

Table 9. Orc asynchronous operational semantics

Notably, the presented operational semantics slightly simplifies that described in [32], in a way that does not misrepresent the properties of the encoding. Indeed, in the original semantics, a site call involves three steps: invocation of the site, response from the site, and publication of the result. Here, instead, a site call is performed in one step, that corresponds to the immediate publication of the result.

The encoding of Orc expressions in COWS exploits function $\langle\!\langle \cdot \rangle\!\rangle_{\hat{r}}$ shown in Table 10. The function is defined by induction on the syntax of expressions and is parameterized by the communication endpoint $\hat{r}$ used to return the result of expressions evaluation. Thus, a site call is rendered as an invoke activity that sends a pair made of the parameter of the invocation and the endpoint for the reply along the endpoint $\hat{S}$ corresponding to site name S. Expression call is rendered similarly, but we need two invoke activities: $\hat{E}!\langle \hat{r}, \hat{r}' \rangle$ activates a new instance of the body of the declaration, while $z!\langle w \rangle$ sends the value of the actual parameter (when this value will be available) to the created instance, by means of a private endpoint stored in z received from the encoding of the corresponding expression declaration along the private endpoint $\hat{r}'$ previously sent. Sequential composition is encoded as the parallel composition of the two components sharing a delimited endpoint, where a new instance of the component on the right is created every time that on the left returns a value along the shared endpoint. Symmetric parallel composition is encoded as parallel composition, where the values produced by the two components are sent along the same return endpoint. Finally, asymmetric parallel composition is encoded in terms of parallel composition in such a way that, whenever the encoding of f returns its first value, this is passed to the encoding of g and a kill activity is enabled. Due to its eager semantics, the kill will terminate what remains of the term corresponding to the encoding of f.

Moreover, for each site S, we define the service:

$$* [x, y] \, \hat{S} ?\langle x, y \rangle . y!\langle e_x^S \rangle \qquad (1)$$

$$\langle\!\langle \mathbf{0} \rangle\!\rangle_{\hat{r}} = \mathbf{0} \qquad \langle\!\langle S(w) \rangle\!\rangle_{\hat{r}} = \hat{S}\,!\langle w, \hat{r} \rangle \qquad \langle\!\langle E(w) \rangle\!\rangle_{\hat{r}} = [\hat{r}'\,]\,(\hat{E}\,!\langle \hat{r}, \hat{r}' \rangle \mid [z]\,\hat{r}'\,?\langle z \rangle.z\,!\langle w \rangle)$$

$$\langle\!\langle f > x > g \rangle\!\rangle_{\hat{r}} = [\hat{r}_f\,]\,(\langle\!\langle f \rangle\!\rangle_{\hat{r}_f} \mid *\,[x]\,\hat{r}_f\,?\langle x \rangle.\langle\!\langle g \rangle\!\rangle_{\hat{r}}) \qquad \langle\!\langle f \mid g \rangle\!\rangle_{\hat{r}} = \langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid \langle\!\langle g \rangle\!\rangle_{\hat{r}}$$

$$\langle\!\langle g \textbf{ where } x :\in f \rangle\!\rangle_{\hat{r}} = [\hat{r}_f, x]\ (\ \langle\!\langle g \rangle\!\rangle_{\hat{r}} \mid [k]\,(\ \langle\!\langle f \rangle\!\rangle_{\hat{r}_f} \mid \hat{r}_f\,?\langle x \rangle.\textbf{kill}(k)\ )\ )$$

Table 10. Orc encoding

that receives along the endpoint $\hat{S}$ a value (stored in x) and an endpoint (stored in y) to be used to send back the result, and returns the evaluation of e^S_x, an unspecified expression corresponding to S and depending on x.

Similarly, for each expression declaration $E(x) \triangleq f$ we define the service:

$$*\,[y, z]\,\hat{E}?\langle y, z \rangle.[\hat{r}\,]\,(z\,!\langle \hat{r} \rangle \mid [x]\,(\hat{r}?\langle x \rangle \mid \langle\!\langle f \rangle\!\rangle_y)\,) \qquad (2)$$

Here, the received value (stored in x) is processed by the encoding of the body of the declaration, that is activated as soon as the expression is called.

Finally, the encoding of an Orc expression f, written $[\![f]\!]_{\hat{r}}$, is the parallel composition of $\langle\!\langle f \rangle\!\rangle_{\hat{r}}$, of a service of the form (1) or (2) for each site or expression called in f, in any of the expressions called in f, and so on recursively.

We can prove that there is a formal correspondence, based on the operational semantics, between Orc expressions and the COWS services resulting from their encoding. To simplify the proof, we found it convenient to extend the syntax of Orc expressions with $f \cdot \{x \mapsto y\}$, that behaves as the expression obtained from f by replacing all free occurrences of x with y. Correspondingly, we add the following rule to those defining the operational semantics:

$$\frac{f \overset{l}{\hookrightarrow} f'}{f \cdot \{x \mapsto y\} \overset{l}{\hookrightarrow} f' \cdot \{x \mapsto y\}}\ \textit{(Sub)}$$

Next proposition, that can be easily proved by induction on the syntax of expressions, states that there is an operational correspondence between the extended semantics and the original one.

Proposition 1. *If $y \notin fv(f)$, then $f \overset{l}{\hookrightarrow} f'$ iff $f \cdot \{x \mapsto y\} \overset{l}{\hookrightarrow} f' \cdot \{x \mapsto y\}$.*

Expression $f \cdot \{x \mapsto y\}$ is encoded as the parallel composition of the encoding of f with a receive activity $\hat{r}'?\langle x \rangle$, that initializes the shared variable x, and with an invoke activity $\hat{r}'\,!\langle y \rangle$, that forwards the value of variable y (when it will be available) along the private endpoint $\hat{r}'$. Formally, it is defined as

$$\langle\!\langle f \cdot \{x \mapsto y\} \rangle\!\rangle_{\hat{r}} = [\hat{r}'\,]\,(\hat{r}'\,!\langle y \rangle \mid [x]\,(\hat{r}'\,?\langle x \rangle \mid \langle\!\langle f \rangle\!\rangle_{\hat{r}})\,)$$

The *operational correspondence* between Orc expressions and the COWS services resulting from their encoding can be characterized by two propositions, which we call

completeness and *soundness*. The former states that all possible executions of an Orc expression can be simulated by its encoding, while the latter states that the initial step of a COWS term resulting from an encoding can be simulated by the corresponding Orc expression so that the continuation of the encoding can evolve in the encoding of the expression continuation. By letting $s \stackrel{\alpha}{\Longrightarrow} s'$ to mean that there exist two services, s_1 and s_2, such that s_1 is a reduct of s, $s_1 \stackrel{\alpha}{\longrightarrow} s_2$ and s' is a reduct of s_2, the two properties can be stated as follows.

Theorem 1 (Completeness). *Given an Orc expression f and a communication endpoint $\hat{r}$, $f \stackrel{l}{\hookrightarrow} f'$ implies $[\![f]\!]_{\hat{r}} \equiv \langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s \stackrel{\alpha}{\Longrightarrow} \langle\!\langle f' \rangle\!\rangle_{\hat{r}} \mid s$, where $\alpha = \hat{r} \triangleleft \langle v \rangle$ if $l = !v$, and $\alpha = (p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v})$ if $l = \tau$.*

Proof: The proof proceeds by induction on the length of the inference of $f \stackrel{l}{\hookrightarrow} f'$.
Base Step: We reason by case analysis on the axioms of the operational semantics.

*(**SiteCall**)* In this case $f = S(v)$, $l = !v'$ and $f' = 0$. By encoding definition, $\langle\!\langle S(v) \rangle\!\rangle_{\hat{r}} \mid s = \hat{S}\,!\langle v, \hat{r} \rangle \mid s$, where $s = *\,[x, y]\,\hat{S}\,?\langle x, y \rangle.y!\langle v' \rangle$. Thus, $\hat{S}\,!\langle v, \hat{r} \rangle \mid s \xrightarrow{\hat{S} \lfloor \emptyset \rfloor \langle x,y \rangle \langle v,\hat{r} \rangle} \hat{r}!\langle v' \rangle \mid s \xrightarrow{\hat{r} \triangleleft \langle v' \rangle} \mathbf{0} \mid s$. Since $\langle\!\langle \mathbf{0} \rangle\!\rangle_{\hat{r}} = \mathbf{0}$, we can conclude.

*(**Def**)* In this case $f = E(w)$ with $E(x) \triangleq g$, $l = \tau$ and $f' = g \cdot \{x \mapsto w\}$. By encoding definition, $\langle\!\langle E(w) \rangle\!\rangle_{\hat{r}} \mid s = [\hat{r}']\,(\hat{E}!\langle \hat{r}, \hat{r}' \rangle \mid [z']\,\hat{r}'?\langle z' \rangle.z'!\langle w \rangle) \mid s$, where $s = *\,[y, z]\,\hat{E}?\langle y, z \rangle.[\hat{r}'']\,(z!\langle \hat{r}'' \rangle \mid [x]\,(\hat{r}''?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_y))$. Thus, $\langle\!\langle E(w) \rangle\!\rangle_{\hat{r}} \mid s \xrightarrow{\hat{E} \lfloor \emptyset \rfloor \langle y,z \rangle \langle \hat{r},\hat{r}' \rangle} [\hat{r}']\,([z']\,\hat{r}'?\langle z' \rangle.z'!\langle w \rangle \mid [\hat{r}'']\,(\hat{r}'!\langle \hat{r}'' \rangle \mid [x]\,(\hat{r}''?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_{\hat{r}}))) \mid s \triangleq s'$. Then, $s' \xrightarrow{\hat{r}' \lfloor \emptyset \rfloor \langle z' \rangle \langle \hat{r}'' \rangle} [\hat{r}'']\,(\hat{r}''!\langle w \rangle \mid [x]\,(\hat{r}''?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_{\hat{r}})) \mid s \triangleq s''$. In case $w = z''$, since $\langle\!\langle g \cdot \{x \mapsto z''\} \rangle\!\rangle_{\hat{r}} \equiv [\hat{r}'']\,(\hat{r}''!\langle z'' \rangle \mid [x]\,(\hat{r}''?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_{\hat{r}}))$, we can directly conclude. Instead, in case $w = v'$, we have $s'' \xrightarrow{\hat{r}'' \lfloor \emptyset \rfloor \langle x \rangle \langle v' \rangle} \langle\!\langle g \rangle\!\rangle_{\hat{r}} \cdot \{x \mapsto v'\} \mid s$. Thus, since $\langle\!\langle g \rangle\!\rangle_{\hat{r}} \cdot \{x \mapsto v'\} \equiv \langle\!\langle g \cdot \{x \mapsto v'\} \rangle\!\rangle_{\hat{r}}$, we can conclude.

Inductive Step: We reason by case analysis on the last applied inference rule of the operational semantics.

*(**Sym1**)* In this case, $f = f_1 \mid f_2$, $f' = f_1' \mid f_2$. By the premise of the rule *(Sym1)*, $f_1 \stackrel{l}{\hookrightarrow} f_1'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s = \langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid \langle\!\langle f_2 \rangle\!\rangle_{\hat{r}} \mid s$. By induction, $\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid s' \stackrel{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\hat{r}} \mid s'$ where $s \equiv s' \mid s''$. By rules *(par$_{pass}$)* or *(par$_{conf}$)*, we can conclude.

*(**Sym2**)* Similar to the previous case.

*(**Seq1**)* In this case, $f = f_1 > x > f_2$, $l = \tau$ and $f' = f_1' > x > f_2$. By the premise of the rule *(Seq1)*, $f_1 \stackrel{\tau}{\hookrightarrow} f_1'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s = [\hat{r}']\,(\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}'} \mid *\,[x]\,\hat{r}'?\langle x \rangle.\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}}) \mid s$. By induction, $\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}'} \mid s' \stackrel{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\hat{r}'} \mid s'$ where $s \equiv s' \mid s''$. By rules *(par$_{conf}$)* and *(del$_{pass}$)*, we can conclude that $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s \stackrel{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}}{\Longrightarrow} \langle\!\langle f_1' > x > f_2 \rangle\!\rangle_{\hat{r}} \mid s$.

*(**Seq2**)* In this case, $f = f_1 > x > f_2$, $l = !v$ and $f' = (f_1' > x > f_2) \mid f_2 \cdot \{x \mapsto v\}$. By the premise of the rule *(Seq2)*, $f_1 \stackrel{!v}{\hookrightarrow} f_1'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s =$

$[\hat{r}']\,(\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}'} \mid *[x]\,\hat{r}'?\langle x\rangle.\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}}) \mid s$. By induction, $\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}'} \mid s' \xrightarrow{\hat{r}' \lhd \langle v\rangle} \langle\!\langle f_1' \rangle\!\rangle_{\hat{r}'} \mid s'$ where $s \equiv s' \mid s''$. Thus, $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s \xrightarrow{\hat{r}'\lfloor\emptyset\rfloor\langle x\rangle\langle v\rangle} [\hat{r}']\,(\langle\!\langle f_1' \rangle\!\rangle_{\hat{r}'} \mid *[x]\,\hat{r}'?\langle x\rangle.\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}} \mid \langle\!\langle f_2 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}}) \mid s \equiv \langle\!\langle f_1' > x > f_2 \rangle\!\rangle_{\hat{r}} \mid \langle\!\langle f_2 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}} \mid s$.

(Asym2) In this case, $f = f_1$ **where** $x :\in f_2$, $l = \tau$ and $f' = f_1$ **where** $x :\in f_2'$. By the premise of the rule *(Asym2)*, $f_2 \overset{\tau}{\hookrightarrow} f_2'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s = [\hat{r}', x]\,(\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid [k]\,(\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}'} \mid \hat{r}'?\langle x\rangle.\mathbf{kill}(k))) \mid s$. By induction, $\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}'} \mid s' \xrightarrow{p \bullet o \lfloor\emptyset\rfloor \bar{w}\,\bar{v}} \langle\!\langle f_2' \rangle\!\rangle_{\hat{r}'} \mid s'$ where $s \equiv s' \mid s''$. Thus, by rule *(del_{pass})*, we can conclude that $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s \xrightarrow{p \bullet o \lfloor\emptyset\rfloor \bar{w}\,\bar{v}} [\hat{r}', x]\ (\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid [k]\,(\langle\!\langle f_2' \rangle\!\rangle_{\hat{r}'} \mid \hat{r}'?\langle x\rangle.\mathbf{kill}(k))) \mid s \equiv \langle\!\langle f_1 \textbf{ where } x :\in f_2' \rangle\!\rangle_{\hat{r}} \mid s$.

(Asym1) Similar to the previous case.

(Asym3) In this case, $f = f_1$ **where** $x :\in f_2$, $l = \tau$ and $f' = f_1 \cdot \{x \mapsto v\}$. By the premise of the rule *(Asym3)*, $f_2 \overset{!v}{\hookrightarrow} f_2'$. By encoding definition, $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s = [\hat{r}', x]\ (\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid [k]\,(\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}'} \mid \hat{r}'?\langle x\rangle.\mathbf{kill}(k))) \mid s$. By induction, $\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}'} \mid s' \xrightarrow{\hat{r}' \lhd \langle v\rangle} \langle\!\langle f_2' \rangle\!\rangle_{\hat{r}'} \mid s'$ where $s \equiv s' \mid s''$. Then, by encoding definition and rule *(del_{pass})*, $s''' \triangleq [\hat{r}', x]\ (\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid [k]\,(\langle\!\langle f_2' \rangle\!\rangle_{\hat{r}'} \mid \hat{r}'?\langle x\rangle.\mathbf{kill}(k)) \mid \hat{r}'!\langle v\rangle) \mid s$ is a reduct of $\langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s$. We can conclude that $s''' \xrightarrow{\hat{r}'\lfloor\emptyset\rfloor\langle x\rangle\langle v\rangle} [\hat{r}']\ (\langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}} \mid [k]\,(\langle\!\langle f_2' \rangle\!\rangle_{\hat{r}'} \mid \mathbf{kill}(k))) \mid s \xrightarrow{\dagger} [\hat{r}']\ (\langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}} \mid [k]\,\mathbf{0}) \mid s \equiv \langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}} \mid s$.

□

Lemma 1. *Given an Orc expression f and a communication endpoint $\hat{r}$, $[\![f]\!]_{\hat{r}} \xrightarrow{\hat{r}!\langle v\rangle}\!\!\!\!\rightarrow$.*

Proof: By a straightforward induction on the definition of the encoding.

□

Theorem 2 (Soundness). *Given an Orc expression f and a communication endpoint $\hat{r}$, $[\![f]\!]_{\hat{r}} \equiv \langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid s \xrightarrow{p \bullet o \lfloor\emptyset\rfloor \bar{w}\,\bar{v}} s'$ implies that there exists an Orc expression f' such that $f \overset{l}{\hookrightarrow} f'$ and $s' \overset{\alpha}{\Longrightarrow} \langle\!\langle f' \rangle\!\rangle_{\hat{r}} \mid s$, where $\alpha = \hat{r} \lhd \langle v\rangle$ if $l = !v$, and $\alpha = (p' \bullet o' \lfloor\emptyset\rfloor \bar{w}'\,\bar{v}')$ if $l = \tau$.*

Proof: The proof proceeds by induction on the definition of the encoding $\langle\!\langle f \rangle\!\rangle_{\hat{r}}$.
Base Step: We reason by case analysis on the non-inductive cases of the definition.

($f = \mathbf{0}$) In this case $\langle\!\langle f \rangle\!\rangle_{\hat{r}} = \mathbf{0}$ and $s = \mathbf{0}$. Since $[\![f]\!]_{\hat{r}} \equiv \mathbf{0} \xrightarrow{\alpha}\!\!\!\!\rightarrow$, we can trivially conculde.

($f = S(w)$) In this case $\langle\!\langle f \rangle\!\rangle_{\hat{r}} = \hat{S}!\langle w, \hat{r}\rangle$ and $s = *[x, y]\,\hat{S}?\langle x, y\rangle.y!\langle v'\rangle$. If $w = z$ then $[\![f]\!]_{\hat{r}} \xrightarrow{\alpha}\!\!\!\!\rightarrow$, thus we can trivially conculde. If $w = v$ then $[\![f]\!]_{\hat{r}} \equiv \hat{S}!\langle v, \hat{r}\rangle \mid s \xrightarrow{\hat{S}\lfloor\emptyset\rfloor\langle x,y\rangle\langle v,\hat{r}\rangle} \hat{r}!\langle v'\rangle \mid s = s'$. By rule *(SiteCall)* we have $l = !v'$ and $f' = \mathbf{0}$. Thus, $s' \xrightarrow{\hat{r}\lhd\langle v'\rangle} \mathbf{0} \mid s$. Since $\langle\!\langle \mathbf{0} \rangle\!\rangle_{\hat{r}} = \mathbf{0}$, we can conclude.

($f = E(w)$) Assuming $E(x) \triangleq g$, we have $\langle\!\langle E(w) \rangle\!\rangle_{\hat{r}} = [\hat{r}']\,(\hat{E}!\langle \hat{r}, \hat{r}'\rangle \mid [z]\,\hat{r}'?\langle z\rangle.z!\langle w\rangle)$ and $s = *[y', z']\,\hat{E}?\langle y', z'\rangle.[\hat{r}'']\,(z'!\langle \hat{r}''\rangle \mid [x]\,(\hat{r}''?\langle x\rangle \mid \langle\!\langle g \rangle\!\rangle_{y'}))$. Then, $[\![f]\!]_{\hat{r}} \xrightarrow{\hat{E}\lfloor\emptyset\rfloor\langle y',z'\rangle\langle \hat{r},\hat{r}'\rangle} [\hat{r}']\,([z]\,\hat{r}'?\langle z\rangle.z!\langle w\rangle) \mid [\hat{r}'']\,(\hat{r}'!\hat{r}'' \mid [x]\,(\hat{r}''?\langle x\rangle \mid \langle\!\langle g \rangle\!\rangle_{\hat{r}})) \mid s = s'$. By rule *(Def)* we have $l = \tau$ and $f' = g \cdot \{x \mapsto w\}$. Moreover, $s' \xrightarrow{\hat{r}'\lfloor\emptyset\rfloor\langle z\rangle\langle \hat{r}''\rangle} [\hat{r}'']\,(\hat{r}''!\langle w\rangle \mid [x]\,(\hat{r}''?\langle x\rangle \mid \langle\!\langle g \rangle\!\rangle_{\hat{r}})) \mid s \triangleq s''$. If $w = y$, we directly conclude, since $s'' = \langle\!\langle g \cdot \{x \mapsto y\} \rangle\!\rangle_{\hat{r}}$. If $w = v$, we can conclude by $s'' \xrightarrow{\hat{r}''\lfloor\emptyset\rfloor\langle x\rangle\langle v\rangle} \langle\!\langle g \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}}$.

Inductive Step: We reason by case analysis on the inductive cases of the definition.

($f = f_1 \mid f_2$) In this case $\langle\!\langle f \rangle\!\rangle_{\hat{r}} = \langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid \langle\!\langle f_2 \rangle\!\rangle_{\hat{r}}$. By encoding definition, the transition $\xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}}$ cannot be produced by a synchronization between $\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}}$ and $\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}}$. Then, we have only the following cases:

- $\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid s \xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}} s_1$.
  By induction, $f_1 \overset{l}{\hookrightarrow} f_1'$ and $s_1 \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\hat{r}} \mid s$. By rule *(Sym1)*, $f \overset{l}{\hookrightarrow} f_1' \mid f_2 = f'$. By rule *($par_{pass}$)* or *($par_{conf}$)*, we can conclude that $s' \equiv s_1 \mid \langle\!\langle f_2 \rangle\!\rangle_{\hat{r}} \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\hat{r}} \mid s \mid \langle\!\langle f_2 \rangle\!\rangle_{\hat{r}} \equiv \langle\!\langle f' \rangle\!\rangle_{\hat{r}} \mid s$.
- $\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}} \mid s \xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}} s_2$.
  Similar to the previous case.

($f = f_1 > x > f_2$) In this case $\langle\!\langle f \rangle\!\rangle_{\hat{r}} = [\hat{r}'](\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}'} \mid *[x]\,\hat{r}'?\langle x \rangle.\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}})$. Lemma 1 implies that the inference of the transition $\xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}}$ derives from $\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}'} \mid s \xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}} s_1$. By induction, $f_1 \overset{l}{\hookrightarrow} f_1'$ and $s_1 \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\hat{r}'} \mid s$. We have two cases:

- $l = \tau$.
  By rule *(Seq1)*, $f \overset{\tau}{\hookrightarrow} f_1' > x > f_2$. By rules *($par_{conf}$)* and *($del_{pass}$)*, $s' \overset{p' \bullet o' \lfloor \emptyset \rfloor \bar{w}'\, \bar{v}'}{\Longrightarrow} [\hat{r}'](\langle\!\langle f_1' \rangle\!\rangle_{\hat{r}'} \mid *[x]\,\hat{r}'?\langle x \rangle.\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}}) \mid s \equiv \langle\!\langle f_1' > x > f_2 \rangle\!\rangle_{\hat{r}} \mid s$.
- $l = !v$.
  By rule *(Seq2)*, $f \overset{!v}{\hookrightarrow} (f_1' > x > f_2) \mid f_2 \cdot \{x \mapsto v\}$. By rules *(com)* and *(del_{pass})*, $s' \overset{\hat{r}' \lfloor \emptyset \rfloor \langle x \rangle \langle v \rangle}{\Longrightarrow} [\hat{r}'](\langle\!\langle f_1' \rangle\!\rangle_{\hat{r}'} \mid *[x]\,\hat{r}'?\langle x \rangle.\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}}) \mid \langle\!\langle f_2 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}} \mid s \equiv \langle\!\langle (f_1' > x > f_2) \mid f_2 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}} \mid s$.

($f = f_1$ **where** $x :\in f_2$) In this case $\langle\!\langle f \rangle\!\rangle_{\hat{r}} = [\hat{r}', x]\,(\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid [k]\,(\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}'} \mid \hat{r}'?\langle x \rangle.\mathbf{kill}(k)))$. By Lemma 1, we have two cases:

- $\langle\!\langle f_1 \rangle\!\rangle_{\hat{r}} \mid s \xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}} s_1$.
  By induction, $f_1 \overset{l}{\hookrightarrow} f_1'$ and $s_1 \overset{\alpha}{\Longrightarrow} \langle\!\langle f_1' \rangle\!\rangle_{\hat{r}'} \mid s$. By rule *(Asym1)*, $f \overset{l}{\hookrightarrow} f_1'$ **where** $x :\in f_2$. By rules *(del_{pass})* and *(par_{conf})*, $s' \overset{\alpha}{\Longrightarrow} [\hat{r}', x]\ (\langle\!\langle f_1' \rangle\!\rangle_{\hat{r}} \mid [k]\,(\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}'} \mid \hat{r}'?\langle x \rangle.\mathbf{kill}(k))) \mid s \equiv \langle\!\langle f_1' \textbf{ where } x :\in f_2 \rangle\!\rangle_{\hat{r}}$.
- $\langle\!\langle f_2 \rangle\!\rangle_{\hat{r}'} \mid s \xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}} s_2$.
  By induction, $f_2 \overset{l}{\hookrightarrow} f_2'$ and $s_2 \overset{\alpha}{\Longrightarrow} \langle\!\langle f_2' \rangle\!\rangle_{\hat{r}'} \mid s$. We have two cases:
  - $\alpha = p' \bullet o' \lfloor \emptyset \rfloor \bar{w}'\, \bar{v}'$.
    Similar to the previous case.
  - $\alpha = \hat{r}'!\langle v \rangle$.
    By rule *(Asym3)*, $f \overset{\tau}{\hookrightarrow} f_1 \cdot \{x \mapsto v\}$. By rules *(com)*, *($par_{conf}$)* and *($del_{pass}$)*, $s' \overset{\hat{r}' \lfloor \emptyset \rfloor \langle x \rangle \langle v \rangle}{\Longrightarrow} \langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}} \mid [k]\,(\langle\!\langle f_2' \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}'} \mid \mathbf{kill}(k)) = s''$. Then, by rules *(kill)*, *(par_{kill})*, *(del_{kill})* and *(par_{pass})*, $s'' \xrightarrow{\dagger} \langle\!\langle f_1 \cdot \{x \mapsto v\} \rangle\!\rangle_{\hat{r}}$.

($f = g \cdot \{x \mapsto y\}$) In this case $\langle\!\langle f \rangle\!\rangle_{\hat{r}} = [\hat{r}'](\hat{r}'!\langle y \rangle \mid [x]\,(\hat{r}'?\langle x \rangle \mid \langle\!\langle g \rangle\!\rangle_{\hat{r}}))$. Since $\hat{r}'!\langle y \rangle \xrightarrow{\alpha}\!\!\!\!\rightarrow$, we have $\langle\!\langle g \rangle\!\rangle_{\hat{r}} \mid s \xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}} s''$. By induction, $g \overset{l}{\hookrightarrow} g'$ and $s'' \overset{\alpha}{\Longrightarrow} \langle\!\langle g' \rangle\!\rangle_{\hat{r}} \mid s$. By rules *($par_{conf}$)* and *($del_{pass}$)*, we can conclude that $s' \overset{\alpha}{\Longrightarrow} [\hat{r}'](\hat{r}'!\langle y \rangle \mid [x]\,(\hat{r}'?\langle x \rangle \mid \langle\!\langle g' \rangle\!\rangle_{\hat{r}})) \mid s \equiv \langle\!\langle g' \cdot \{x \mapsto y\} \rangle\!\rangle_{\hat{r}} \mid s$.

□

P, Q, T	::=		(processes)
		$\mathbf{0}$	(nil)
	|	$a.P$	(concretion)
	|	$(x)P$	(abstraction)
	|	return $a.P$	(return value)
	|	$a \Rightarrow (x)P : (y)T$	(service definition)
	|	$a\{(x)P\} \Leftarrow_\ell Q$	(service invocation)
	|	$a \triangleright_\ell P$	(session)
	|	$P \mid Q$	(parallel composition)
	|	$(\nu a)P$	(new name)

Table 11. SCC syntax

5.2 Encoding SCC

SCC (Service Centered Calculus [3]) is a formalism aiming to serve as a basis for programming and composing services, while taking into account their dynamic behaviour. SCC integrates complementary aspects borrowed from well-known process calculi, such as π-calculus (names handling primitives), Orc (pipelining and pruning of activities), web-π, cJoin, and Sagas (long running transactions and compensations). A key feature of SCC is the session handling mechanism, that allows for the definition of structured interaction protocols. A *session* permits bi-directional communication between two parties (service- and client-side). Transparently for the programmers, sessions are opened by service invocations, which spawn fresh session parties (locally to each partner). Moreover, sessions can be nested and values can be returned outside sessions. Finally, SCC sessions can be closed by the involved parties, providing a mechanism for process interruption and service cancellation and update.

The syntax of SCC, given in Table 11, is parameterized by a countable set of names, ranged over by a, b, ..., x, y,.... Basically, in SCC processes are defined as parallel compositions of service definitions and service invocations. *Service definitions* take the form $a \Rightarrow (x)P : (y)T$, where a is the service name, x is a formal parameter, P is the actual implementation of the service, and $(y)T$ is the protocol of the termination handler service that will be instantiated on the server-side in service invocation. *Service invocations* are written as $a\{(x)P\} \Leftarrow_\ell Q$: each new value v produced by the client Q will trigger a new invocation of service a; for each invocation, an instance of the process P, with x bound to the actual invocation value v, implements the client-side protocol for interacting with the new instance of a. Name ℓ identifies the termination handler service on the client-side. A service invocation causes activation of a new *session*: a pair of dual fresh names, r and $\tilde{r}$, identifies the two sides of the session. A session side has the form $r \triangleright_\ell P$ and the first time the protocol P invokes the termination handler service identified by ℓ (i.e. that of the other side), the session is closed. Notably, in order to program the session termination, the calculus has a special name close, that is replaced by the name of the termination handler instantiated on the other side at invocation time. Within a session, client and server protocols can communicate whenever a *concretion* is available on one side and an *abstraction* is ready on the other side, i.e. abstractions

(Concretion)	$[\![a.P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= \hat{out}!\langle a\rangle.[\![P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$
(Abstraction)	$[\![(x)P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= [x]\,\hat{in}?\langle x\rangle.[\![P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$
(Return value)	$[\![\mathsf{return}\ a.P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= \hat{ret}!\langle a\rangle.[\![P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$
(Session)	$[\![a \triangleright_\ell P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= [k]\,[\![P]\!]^{K,\{\ell\mapsto k\}}_{\hat{a}_1,\hat{a}_2,\hat{out}}$
	$[\![\tilde{a} \triangleright_\ell P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= [k]\,[\![P]\!]^{K,\{\ell\mapsto k\}}_{\hat{a}_2,\hat{a}_1,\hat{out}}$
(New name)	$[\![(\nu a)P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= [a]\,[\![P]\!]^K_{\hat{in},\hat{out},\hat{ret}}$ if $a \notin dom(K)$
(Parallel comp.)	$[\![P \mid Q]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= [\![P]\!]^K_{\hat{in},\hat{out},\hat{ret}} \mid [\![Q]\!]^K_{\hat{in},\hat{out},\hat{ret}}$
(Service def.)	$[\![a \Rightarrow (x)P : (z)T]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= *\,[x_1,x_2,y_1,y_2,x]\ s\hat{c}c?\langle a,x_1,x_2,y_1,y_2,x\rangle.$ $[k]\,(\,[\![P[y_1/\mathsf{close}]]\!]^{K,\{y_1\mapsto k\}}_{x_2,x_1,\hat{out}}$ $\mid *\,[z]\,\hat{th}?\langle y_2,z\rangle.[\![T[y_1/\mathsf{close}]]\!]^{K,\{y_1\mapsto k\}}_{\_,\_,\hat{out}}\,)$
(Service inv.)	$[\![a\{(x)P\} \Leftarrow_\ell Q]\!]^K_{\hat{in},\hat{out},\hat{ret}}$	$= \begin{cases} [\hat{r}]\,(\,[\![Q]\!]^K_{\hat{in},\hat{r},\hat{ret}} \mid *\,[x]\,\hat{r}?\langle x\rangle. \\ \quad [\ell',\hat{i},\hat{o}]\ s\hat{c}c!\langle a,\hat{i},\hat{o},\ell,\ell',x\rangle. \\ \quad [k]\,[\![P[\ell'/\mathsf{close}]]\!]^{K,\{\ell'\mapsto k\}}_{\hat{i},\hat{o},\hat{out}}\,) & \text{If } a \notin dom(K) \\ [\hat{r}]\,(\,[\![Q]\!]^K_{\hat{in},\hat{r},\hat{ret}} \mid *\,[x]\,\hat{r}?\langle x\rangle. \\ \quad \hat{th}!\langle a,x\rangle.\mathbf{kill}(K(a))\,) & \text{If } a \in dom(K) \end{cases}$

Table 12. SCC encoding

and concretions model input and output, respectively. A value can be returned outside the current session (just one level up) by means of the primitive *return*.

COWS is an asynchronous calculus while SCC relies on synchronous communication, thus, for the sake of the presentation, we define the encoding from SCC to 'synchronous COWS', denoted by $[\![\_]\!]^K_{\hat{in},\hat{out},\hat{ret}}$, and then we exploit the auxiliary encoding $\langle\!\langle\_\rangle\!\rangle$, from the synchronous version of COWS to the original one. The auxiliary encoding acts as an homomorphism except for the the communication activities:

$$\langle\!\langle u \bullet u'!\langle e_1,\ldots e_n\rangle.s\rangle\!\rangle = [\hat{r}]\,(u \bullet u'!\langle e_1,\ldots e_n,\hat{r}\rangle \mid \hat{r}?\langle\rangle.\langle\!\langle s\rangle\!\rangle)$$
$$\langle\!\langle p \bullet o?\langle w_1,\ldots,w_n\rangle.s\rangle\!\rangle = [x]\,(p \bullet o?\langle w_1,\ldots,w_n,x\rangle.(\hat{x}!\langle\rangle \mid \langle\!\langle s\rangle\!\rangle)\,)$$

The encoding $[\![\_]\!]^K_{\hat{in},\hat{out},\hat{ret}}$, shown in Table 12, is inspired by that from the close-free fragment of SCC to π-calculus [3]. The encoding is parametric on three endpoints used to receive values from ($\hat{in}$), send values to ($\hat{out}$), and return values to the enclosing session ($\hat{ret}$). Moreover, it exploits an auxiliary function K, that maps SCC names used to identify termination handler services (ranged over by ℓ) to COWS killer labels (ranged over by k).

We comment on salient points. Concretion and return actions are rendered as outputs on the endpoints $\hat{out}$ and $\hat{ret}$ respectively, while abstractions are rendered as inputs on $\hat{in}$. The three endpoints are set by the encoding of sessions that, to permit closing a session, also introduces a delimitation for a killer label (associated to the session name by updating the function K). To model bi-directional sessions we associate a pair of endpoints $\hat{a}_1$ and $\hat{a}_2$ to each session name a. Parallel composition and restriction operators are mapped homomorphically. In the encoding of restriction, the extra condition $a \notin dom(K)$ avoids name conflicts between the encoded term and the function K, and can always be satisfied by first alpha-converting the a in $(\nu a)P$. To model service definitions and invocations we exploit a distinguished endpoint $\hat{scc}$; each service definition waits invocations on this endpoint, and request messages are routed to the correct service by means of their first field, that is the name of the invoked service and acts as a correlation value. Similarly, we exploits a distinguished endpoint $\hat{th}$ to model termination handler services. A service definition, when it is invoked (along $\hat{scc}$), instantiates the service protocol and the termination handler service, by using the received data. Name close, in the service protocol and in the termination handler, is replaced by a killer label corresponding to the client termination handler name. Finally, for service invocation we differentiates two cases: service invocation and termination handler invocation. In both cases, outputs on the fresh endpoint $\hat{r}$ where process Q produces the parameters for service invocation are intercepted, and each value v triggers an invocation of the service a. In the former case ($a \notin dom(K)$), the invoked endpoint is $\hat{scc}$, and the transmitted data are as follows: the service name a (for correlation purpose), two fresh endpoints $\hat{\imath}$ and $\hat{o}$ for bi-directional session communication, two termination handler names ℓ and ℓ' (the former is installed at client-side, while the latter will be installed at server-side), and the produced value v. After the invocation, the client protocol P is instantiated. In case of invocation of a termination handler service ($a \in dom(K)$), the invoked endpoint is $\hat{th}$, and after this invocation, the corresponding session is closed by means of the kill activity.

5.3 Encoding WS-CALCULUS

The *syntax* of (a simplified version of) WS-CALCULUS, given in Table 13, is parameterized with respect to the following syntactic sets: *properties* (sorts of late bound constants storing some relevant values within service instances, ranged over by p), *values* (basic values and addresses, ranged over by u), *partner links* (variables storing addresses used to identify service partners within an interaction), *operation parameters* (basic variables, partner links and properties, ranged over by w), and *service identifiers* (ranged over by A). Notationally, we will use a to range over *addresses* and r to range over addresses and partner links.

WS-CALCULUS permits to model the interactions among web service instances in a network context. A network of services is a finite set of nodes. Nodes, written as $a :: C$, are uniquely identified by an address a and host components C. *Components* C may be service specifications, instances or requests. The behavioural specification of a service s is written $*s$, while $m \gg s'$ represents a service instance that behaves according to s' and whose properties evaluate according to the (possibly empty) set m of correlation constraints. A *correlation constraint* is a pair, written $p = u$, recording the value u

n	::=	$a :: C$	(nodes)
C	::=	$*s \quad \mid \quad m \gg s$	(components)
	$\mid$	$\langle a, o, \bar{u} \rangle \quad \mid \quad C \mid C$	
m	::=	$\emptyset \quad \mid \quad \{p = u\} \quad \mid \quad m \cup m$	(correl. const.)
s	::=		(services)
		$\mathbf{0}$	(null)
	$\mid$	$\mathbf{exit}$	(exit)
	$\mid$	$\mathbf{ass}\,(\bar{w}, \bar{e})$	(assign)
	$\mid$	$\mathbf{inv}\,(r, o, \bar{w})$	(invoke)
	$\mid$	$\mathbf{rec}\,(r, o, \bar{w})$	(receive)
	$\mid$	$\mathbf{if}\ (e)\ \mathbf{then}\ \{s\}\ \mathbf{else}\ \{s\}$	(switch)
	$\mid$	$s; s$	(sequence)
	$\mid$	$s \mid s$	(flow)
	$\mid$	$\sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i$	(pick)
	$\mid$	$A(\bar{w})$	(call)

Table 13. ws-calculus syntax

assigned to the property p. Properties are used to store values that are important to identify service instances. A service request $\langle a, o, \bar{u} \rangle$ represents an operation invocation that must still be processed and contains the invoker address a, the operation name o and the data $\bar{u}$ for operation execution.

Services are structured activities built from *basic activities*, i.e. instance forced termination **exit**, assignment $\mathbf{ass}\,(\_\,,\_)$, service invocation $\mathbf{inv}\,(\_\,,\_\,,\_)$ and service request processing $\mathbf{rec}\,(\_\,,\_\,,\_)$, by exploiting operators for conditional choice $\mathbf{if}\ (\_)\ \mathbf{then}\ \{\_\}\ \mathbf{else}\ \{\_\}$ (*switch*), sequential composition $\_\,;\,\_$ (*sequence*), parallel composition $\_ \mid \_$ (*flow*), external choice $\sum_{i \in I} \mathbf{rec}\,(\_\,,\_\,,\_)\,;\,\_$ (*pick*) and service call $A(\bar{w})$ (of course, we assume that every service identifier A has a unique definition of the form $A(\bar{w}) \stackrel{def}{=} s$).

The encoding from ws-calculus to COWS, denoted by $\langle\!\langle \cdot \rangle\!\rangle$, is shown in Table 14. We only show the relevant cases; for example, the encodings of structural activities are quite standard and, hence, omitted. Both variables and properties of ws-calculus are encoded as COWS variables, while the encoding of a net, i.e. a finite set of nodes, is the parallel composition of the encodings of its nodes. The encoding of a node is given in term of the encoding of the hosted component parameterized by the address of the node.

This auxiliary parameterized encoding $\langle\!\langle \cdot \rangle\!\rangle_a$ is defined inductively over the syntax of services. The parameter a is used, e.g., both as the partner name to which a given request must be sent and to refer the killer label k_a used to identify each service instance. Service specifications are encoded as COWS persistent services, by exploiting the replication operator. The fact that variables are global to service instances is rendered through the outermost delimitation $[k_a, V(s)]$, where $V(s)$ is the set of free variables and properties of s. Partner links used to identify service partners within an interaction are translated by exploiting the address of the hosting node. Finally, the effect of executing **exit** inside a service instance hosted at a is achieved by forcing termination of the COWS term resulting from the encoding the instance and identified by the label k_a.

$$
\begin{array}{rcl}
\langle\!\langle a :: C \rangle\!\rangle & = & \langle\!\langle C \rangle\!\rangle_a \\
\langle\!\langle *s \rangle\!\rangle_a & = & *\,[k_a, V(s)]\,\langle\!\langle s \rangle\!\rangle_a \\
\langle\!\langle \{\bar{p} = \bar{u}\} \gg s \rangle\!\rangle_a & = & [k_a, V(s)]\,\langle\!\langle s \cdot \{\bar{p} \mapsto \bar{u}\} \rangle\!\rangle_a \\
\langle\!\langle \langle a', o, \bar{u} \rangle \rangle\!\rangle_a & = & \{\!|\, a \bullet o!\langle a', \bar{u} \rangle \,|\!\} \\
\langle\!\langle \mathbf{0} \rangle\!\rangle_a & = & \mathbf{0} \\
\langle\!\langle \mathbf{exit} \rangle\!\rangle_a & = & \mathbf{kill}(k_a) \\
\langle\!\langle \mathbf{ass}\,(\bar{w}, \bar{e}) \rangle\!\rangle_a & = & [\bar{w} = \bar{e}] \\
\langle\!\langle \mathbf{inv}\,(r, o, \bar{w}) \rangle\!\rangle_a & = & r \bullet o!\langle a, \bar{w} \rangle \\
\langle\!\langle \mathbf{rec}\,(r, o, \bar{w}) \rangle\!\rangle_a & = & a \bullet o?\langle r, \bar{w} \rangle
\end{array}
$$

Table 14. ws-calculus encoding

Of course, as in Section 5.1, we could prove that there is a formal correspondence, based on the operational semantics, between ws-calculus nets and the COWS services resulting from their encoding.

6 Timed extensions of COWS

We present here an extension of COWS that permits modeling timed activities. Specifically, we consider the *wait* activity of WS-BPEL which causes execution of the invoking service to be suspended until the time interval specified as an argument has elapsed. For the sake of presentation, we postpone the introduction of attribute *until*, that causes suspension of the invoking service until the absolute time reaches the value specified as an argument.

We assume that the set of values now includes a set of positive numbers (ranged over by $\delta, \delta', \ldots$), used to represent *time intervals*. The syntax of COWS is extended as follows:

$$g \;::=\; \ldots \;\mid\; {}^{\textcircled{\scriptsize t}}{}_{e}.s$$

Basically, guards are extended with the *wait activity* ${}^{\textcircled{\scriptsize t}}{}_{e}$, that specifies the time interval, whose value is given by evaluation of e, the executing service has to wait for. Consequently, the choice construct can now be guarded both by message reception and time-out expiration, like WS-BPEL *pick* activity. We assume that evaluation of expressions and execution of basic activities, except for ${}^{\textcircled{\scriptsize t}}{}_{e}$, are instantaneous (i.e. do not consume time units) and time elapses between them.

The operational semantics of the extended language is defined in terms of the labelled transition relation $\xrightarrow{\hat{\alpha}}$, where $\hat{\alpha}$ stands for α or δ (that models time elapsing),

$$\mathbf{0} \xrightarrow{\delta} \mathbf{0} \quad (nil_{elaps}) \qquad * s \xrightarrow{\delta} * s \quad (repl)$$

$$p \bullet o?\bar{w}.s \xrightarrow{\delta} p \bullet o?\bar{w}.s \quad (rec_{elaps}) \qquad u \bullet u'!\bar{e} \xrightarrow{\delta} u \bullet u'!\bar{e} \quad (inv_{elaps})$$

$$\textcircled{\scriptsize$\vphantom{0}$}_0.s \xrightarrow{\dagger} s \quad (wait_{tout}) \qquad \frac{\delta \leqslant [\![e]\!]}{\textcircled{\scriptsize$\vphantom{0}$}_e.s \xrightarrow{\delta} \textcircled{\scriptsize$\vphantom{0}$}_{[\![e-\delta]\!]}.s} \quad (wait_{elaps})$$

$$\frac{[\![e]\!] \neq \delta'}{\textcircled{\scriptsize$\vphantom{0}$}_e.s \xrightarrow{\delta} \textcircled{\scriptsize$\vphantom{0}$}_e.s} \quad (wait_{err}) \qquad \frac{s \xrightarrow{\delta} s'}{\{\!|s|\!\} \xrightarrow{\delta} \{\!|s'|\!\}} \quad (prot_{elaps})$$

$$\frac{g_1 \xrightarrow{\delta} g_1' \qquad g_2 \xrightarrow{\delta} g_2'}{g_1 + g_2 \xrightarrow{\delta} g_1' + g_2'} \quad (pick) \qquad \frac{s_1 \xrightarrow{\delta} s_1' \qquad s_2 \xrightarrow{\delta} s_2'}{s_1 \mid s_2 \xrightarrow{\delta} s_1' \mid s_2'} \quad (par_{sync})$$

$$\frac{s \xrightarrow{\delta} s'}{[d]\, s \xrightarrow{\delta} [d]\, s'} \quad (scope_{elaps})$$

Table 15. Synchronous timed COWS operational semantics (additional rules)

obtained by adding the rules shown in Table 15 to those in Table 5. Let us briefly comment on the new rules. Time can elapse while waiting on receive/invoke activities, rules *(rec$_{elaps}$)* and *(inv$_{elaps}$)*. When time elapses, but the timeout is still not expired, the argument of wait activities $\textcircled{\scriptsize$\vphantom{0}$}$ is updated (rule *(wait$_{elaps}$)*). Time elapsing cannot make a choice within a pick activity (rule *(pick)*), while the occurrence of a timeout can. Indeed, it is signalled by label †, thus is a computation step, generated by rule *(wait$_{tout}$)* and used by rule *(choice)* to discard the alternative branches. Time elapses synchronously for all services running in parallel: this is modelled by rule *(par$_{sync}$)* and the remaining rules for the empty activity (rule *(nil$_{elaps}$)*), replication (rule *(repl)*), for the wait activity (rule *(wait$_{err}$)*), for protection (rule *(prot$_{elaps}$)*) and delimitation (rule *(scope$_{elaps}$)*). In particular, rule *(wait$_{err}$)* enables time passing for the wait activity also when the expression e used as an argument does not return a positive number; in this case the argument of the wait is left unchanged. Note that, in agreement with its eager semantics, the kill activity does not allow time to pass. Computations can now also include transitions labelled by δ corresponding to time elapsing.

Since time elapses synchronously for all services in parallel, we can think of as all services run on a same service *engine* and share the same clock. By making it explicit the notion of service engine and of deployment of services on engines, it is possible to describe more realistic scenarios. Therefore, we extend the language syntax with the syntactic category of (*service*) *engines* (alike the 'machines' of [22]) defined as follows:

$$\mathbb{E} \;::=\; 0 \;\mid\; \{s\} \;\mid\; [n]\,\mathbb{E} \;\mid\; \mathbb{E} \mid \mathbb{E}$$

$$\frac{s \xrightarrow{\hat{\alpha}} s' \qquad \hat{\alpha} \in \{\delta, \dagger, p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \bar{v}\}}{\{s\} \to \{s'\}}\ (loc) \qquad \frac{\mathbb{E} \to \mathbb{E}'}{[n]\,\mathbb{E} \to [n]\,\mathbb{E}'}\ (res)$$

$$\frac{\mathbb{E} \equiv \mathbb{E}' \qquad \mathbb{E}' \to \mathbb{F}' \qquad \mathbb{F}' \equiv \mathbb{F}}{\mathbb{E} \to \mathbb{F}}\ (cong_E) \qquad \frac{\mathbb{E} \to \mathbb{E}'}{\mathbb{E} \mid \mathbb{F} \to \mathbb{E}' \mid \mathbb{F}}\ (par_{async})$$

$$\frac{s_1 \xrightarrow{(p \bullet o) \lhd \bar{v}} s_1' \qquad s_2 \xrightarrow{(p \bullet o) \rhd \bar{w}} s_2' \qquad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \qquad \mathrm{fv}(\bar{w}) = \bar{x} \qquad \neg(s_2 \downarrow^{|\sigma|}_{p \bullet o, \bar{v}})}{\{s_1\} \mid \{\,[\bar{x}]\, s_2\} \to \{s_1'\} \mid \{s_2' \cdot \sigma\}}\ (com_E)$$

Table 16. Asynchronous timed COWS operational semantics (additional rules)

Each engine $\{s\}$ has its own clock (whose value does not matter and, hence, is not made explicit), that is not synchronized with the clock of other parallel engines (namely, time progresses asynchronously among different engines). Besides, (private) names can be shared among engines, while variables and killer labels cannot. In the sequel, we will only consider *well-formed* engine compositions, i.e. engine compositions where partners used in communication endpoints of receive activities within different service engines are pairwise distinct. The underlying rationale is that each service has its own partner names and that the service and all its instances run within the same engine.

To define the semantics, we first extend the structural congruence of Section 2.2 with the abelian monoid laws for engines parallel composition and with the following laws:

$$\{s\} \equiv \{s'\} \ \text{ if } s \equiv s' \qquad \{\mathbf{0}\} \equiv 0 \qquad \{[n]\, s\} \equiv [n]\{s\} \qquad [n]\, 0 \equiv 0$$
$$[n]\,[m]\,\mathbb{E} \equiv [m]\,[n]\,\mathbb{E} \qquad \mathbb{E} \mid [n]\,\mathbb{F} \equiv [n]\,(\mathbb{E} \mid \mathbb{F}) \ \text{ if } n \notin \mathrm{fd}(\mathbb{E})$$

The first law lifts to engines the structural congruence defined on services, the second law transforms an engine with empty activities into an empty engine, while the third law permits to extrude a private name outside an engine. The remaining laws are standard.

Secondly, we define a reduction relation $\to$ among engines through the rules shown in Table 16. Rule *(loc)* models occurrence of a computation step within an engine, while rule *(res)* deals with private names. Rule *($cong_E$)* says that structurally congruent engines have the same behaviour, while rule *(par_{async})* says that time elapses asynchronously between different engines (indeed, $\mathbb{F}$ and, then, the clocks of its engines remain unchanged after the transition). Rule *(com_E)*, where $\mathrm{fv}(\bar{w})$ are the free variables of $\bar{w}$, enables interaction between services executing within different engines. It combines the effects of rules *(del_{sub})* and *(com)* in Table 5. Indeed, since the delimitations $[\bar{x}]$ for the input variables are singled out, the communication effect can be immediately applied to the continuation s_2' of the service performing the receive. The last premise ensures that, in case of multiple start activities, the message is routed to the correlated service instance rather than triggering a new instantiation.

Notably, computations from a given parallel composition of engines are sequences of (connected) reductions. Communication can take place *intra-engine*, by means of rule *(com)*, or *inter-engine*, by means of rule *(com_E)*. In both cases, since we are only con-

sidering well-formed compositions of engines, checks for receive conflicts are confined to services running within a single engine, the one performing the receive, differently from the language without explicit engines, where checks involve the whole composition of services. Notice that, to communicate a private name between engines, first it is necessary to exploit the structural congruence for extruding the name outside the sending engine and to extend its scope to the receiving engine, then the communication can take place, by applying rules *(com$_E$)*, *(res)* and *(cong$_E$)*.

Finally, we introduce the attribute *until* that modifies the semantics of wait activity so that the invoking service is suspended until the absolute time reaches the value resulting by the evaluation of the argument of the *wait until*, denoted by $\oplus_{U\,e}$. The set of values now includes also a set of *time values*, ranged over by $t, t', \ldots$, that are used to explicitly indicate the value of engines' clock, now denoted by $\{s\}_t$.

The operational semantics changes accordingly. In particular, the laws for structural congruence over engines become as follows:

$$\{s\}_t \;\equiv\; \{s'\}_t \text{ if } s \equiv s' \qquad\qquad \{[n]\,s\}_t \;\equiv\; [n]\,\{s\}_t$$

The semantics of services is defined in terms of configurations of the form $t \succ s$, for taking into account the absolute time t of the engine where the service s run, and of labelled transitions between configurations $t \succ s \xrightarrow{\hat{\alpha}} t' \succ s'$. The rules in Table 5 and Table 15 are then tailored for using configurations. For example, the rules for the receive activity become as follows:

$$t \succ p\bullet o?\bar{w}.s \xrightarrow{(p\bullet o)\rhd\bar{w}} t \succ s \quad (rec) \qquad t \succ p\bullet o?\bar{w}.s \xrightarrow{\delta} t+\delta \succ p\bullet o?\bar{w}.s \quad (rec_{elaps})$$

Additionally, we have the following two rules for the wait until activity:

$$\frac{t+\delta < [\![e]\!]}{t \succ \oplus_{U\,e}.s \xrightarrow{\delta} t+\delta \succ \oplus_{U\,e}.s}\ (waitUntil_1) \qquad \frac{[\![e-t]\!] = \delta}{t \succ \oplus_{U\,e}.s \xrightarrow{\delta} [\![e]\!] \succ \oplus_0.s}\ (waitUntil_2)$$

$$\frac{[\![e]\!] \neq t'}{t \succ \oplus_{U\,e}.s \xrightarrow{\delta} t+\delta \succ \oplus_{U\,e}.s}\ (waitUntil_{err})$$

which permit time elapsing until the clock of the engine reaches the argument of $\oplus_{U\,e}$. Finally, the semantics of engines' composition is tailored by replacing rules *(loc)* and *(com$_E$)* with the following ones:

$$\frac{t \succ s \xrightarrow{\delta} t+\delta \succ s'}{\{s\}_t \rightarrow \{s'\}_{t+\delta}}\ (loc_{elaps}) \qquad \frac{t \succ s \xrightarrow{\alpha} t \succ s' \quad \alpha \in \{\dagger, p\bullet o\lfloor\emptyset\rfloor\,\bar{w}\,\bar{v}\}}{\{s\}_t \rightarrow \{s'\}_t}\ (loc_{act})$$

$$\frac{\begin{array}{c} t_1 \succ s_1 \xrightarrow{(p\bullet o)\lhd\bar{v}} t_1 \succ s_1' \qquad t_2 \succ s_2 \xrightarrow{(p\bullet o)\rhd\bar{w}} t_2 \succ s_2' \qquad \mathcal{M}(\bar{w},\bar{v}) = \sigma \\ \mathrm{fv}(\bar{w}) = \bar{x} \qquad \neg([\bar{x}]\,s_2 \downarrow^{|\sigma|}_{p\bullet o,\bar{v}}) \end{array}}{\{s_1\}_{t_1} \mid \{[\bar{x}]\,s_2\}_{t_2} \rightarrow \{s_1'\}_{t_1} \mid \{s_2'\cdot\sigma\}_{t_2}}\ (com_E)$$

We end this section with some examples of application of the extended framework. The first example provides a sort of clock service, while the second one is a timed

variant of the *rps* service described in Section 4.1. The remaining examples are use-cases inspired by [11, 12].

A clock service. By using the timed activity, we can define a sort of *clock service* that is set to send a message "*tick*′′ along $\hat{m}$ every 10 time units.

$$[\hat{n}]\,(\,\hat{n}!\langle\rangle \mid *\,\hat{n}?\langle\rangle\,.\,\textcircled{\scriptsize 10}_{10}\,.\,(\,\hat{m}!\langle\text{“}tick''\rangle \mid \hat{n}!\langle\rangle\,)\,)$$

Notice that, however, because invoke activities are executed lazily, it is only guaranteed that *at least* (and *not exactly*) 10 time units elapse between two consecutive emissions of message "*tick*′′.

Timed Rock/Paper/Scissors service. We consider a variant of the *rps* service that exploits the fact that the choice construct can now be guarded both by receive activities and by timed activities (like the *pick* activity of WS-BPEL). Essentially, an instance of service *t_rps*, that is created because of the reception of the first throw of a challenge, waits the reception of the corresponding second throw for at most 30 time units. If this throw arrives within the deadline, the instance behaves as usual. Otherwise, when the timeout expires, the instance declares the sender of the first throw as the winner of the challenge and terminates.

$$\begin{aligned}
t_rps \;\triangleq\; & *\,[x_{champ_res}, x_{chall_res}, x_{id}, x_{thr_1}, x_{thr_2}, x_{win}, k] \\
& (\,(\,p_{champ} \bullet o_{throw}?\langle x_{champ_res}, x_{id}, x_{thr_1}\rangle. \\
& \quad (\,p_{chall} \bullet o_{throw}?\langle x_{chall_res}, x_{id}, x_{thr_2}\rangle. \\
& \quad\quad (\,x_{champ_res} \bullet o_{win}!\langle x_{id}, x_{win}\rangle \mid x_{chall_res} \bullet o_{win}!\langle x_{id}, x_{win}\rangle\,) \\
& \quad\quad + \textcircled{\scriptsize 30}_{30}\,.\,(\,\{\!|x_{champ_res} \bullet o_{win}!\langle x_{id}, x_{champ_res}\rangle|\!\} \mid \mathbf{kill}(k)\,)\,) \\
& \quad + \; p_{chall} \bullet o_{throw}?\langle x_{chall_res}, x_{id}, x_{thr_2}\rangle. \\
& \quad\quad (\,p_{champ} \bullet o_{throw}?\langle x_{champ_res}, x_{id}, x_{thr_1}\rangle. \\
& \quad\quad\quad (\,x_{champ_res} \bullet o_{win}!\langle x_{id}, x_{win}\rangle \mid x_{chall_res} \bullet o_{win}!\langle x_{id}, x_{win}\rangle\,) \\
& \quad\quad\quad + \textcircled{\scriptsize 30}_{30}\,.\,(\,\{\!|x_{chall_res} \bullet o_{win}!\langle x_{id}, x_{chall_res}\rangle|\!\} \mid \mathbf{kill}(k)\,)\,)\,) \\
& \quad \mid Assign\,)
\end{aligned}$$

A Buyer/Seller/Shipper protocol. We illustrate a simple business protocol for purchasing a fixed good. The protocol, graphically represented in Figure 3, involves a buyer, a seller and a shipper. Firstly, *Buyer* asks *Seller* to offer a quote, then, after the *Seller*'s reply, *Buyer* answers with either an acceptance or a rejection message (it sends the latter when the quote is bigger than a certain amount). In case of acceptance, *Seller* sends a confirmation to *Buyer* and asks *Shipper* to provide delivery details. Finally, *Seller* forwards the received delivery information to *Buyer*. Moreover, after *Seller* presents a quote, if *Buyer* does not reply in 30 time units, then *Seller* will abort the transaction. In the end, the whole system is

$$\{Buyer\} \mid \{Seller\} \mid \{Shipper\}$$

where

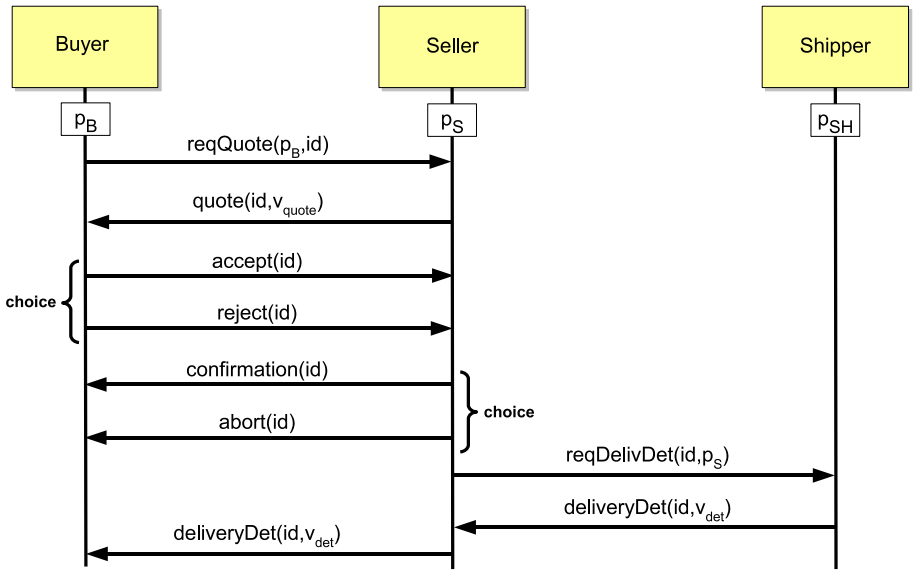

Fig. 3. Graphical representation of the Buyer/Seller/Shipper protocol

$$
\begin{aligned}
\mathit{Buyer} \triangleq\ & [id]\,(\ p_S \bullet o_{reqQuote}!\langle p_B, id\rangle \\
& \mid [x_{quote}]\ p_B \bullet o_{quote}?\langle id, x_{quote}\rangle\ . \\
& \quad [k]\,(\ \textbf{if}\ (x_{quote} \leq 1000) \\
& \qquad \textbf{then}\ \{\ p_S \bullet o_{accept}!\langle id\rangle \\
& \qquad\quad \mid p_B \bullet o_{confirmation}?\langle id\rangle\ . \\
& \qquad\quad\ [x_{det}]\ p_B \bullet o_{deliveryDet}?\langle id, x_{det}\rangle\ \} \\
& \qquad \textbf{else}\ \{\ p_S \bullet o_{reject}!\langle id\rangle\ \} \\
& \quad \mid p_B \bullet o_{abort}?\langle id\rangle\ .\ \textbf{kill}(k)\ )\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Seller} \triangleq\ & *\,[x_B, x_{id}]\ p_S \bullet o_{reqQuote}?\langle x_B, x_{id}\rangle\ . \\
& \quad (\ x_B \bullet o_{quote}!\langle x_{id}, v_{quote}\rangle \\
& \quad\ \mid p_S \bullet o_{accept}?\langle x_{id}\rangle\ . \\
& \qquad (\ x_B \bullet o_{confirmation}!\langle x_{id}\rangle \\
& \qquad\ \mid p_{SH} \bullet o_{reqDelivDet}!\langle x_{id}, p_S\rangle \\
& \qquad\ \mid [x_{det}]\ p_S \bullet o_{deliveryDet}?\langle x_{id}, x_{det}\rangle\ . \\
& \qquad\quad x_B \bullet o_{deliveryDet}!\langle x_{id}, x_{det}\rangle\ ) \\
& \quad + p_S \bullet o_{reject}?\langle x_{id}\rangle \\
& \quad + \textcircled{\scriptsize 30}\ .\ x_B \bullet o_{abort}!\langle x_{id}\rangle\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Shipper} \triangleq\ & *\,[x_{id}, x_S]\ p_{SH} \bullet o_{reqDelivDet}?\langle x_{id}, x_S\rangle\ . \\
& \quad [x_{det}]\,[x_{det} = \mathit{computeDelivDet}(x_S)]\ .\ x_S \bullet o_{deliveryDet}!\langle x_{id}, x_{det}\rangle
\end{aligned}
$$

Function *computeDelivDet*(_) computes the delivery details associated to a seller. Notably, if *Buyer* receives an *abort* message from *Seller*, then it immediately halts its other activities, by means of the killing activity.

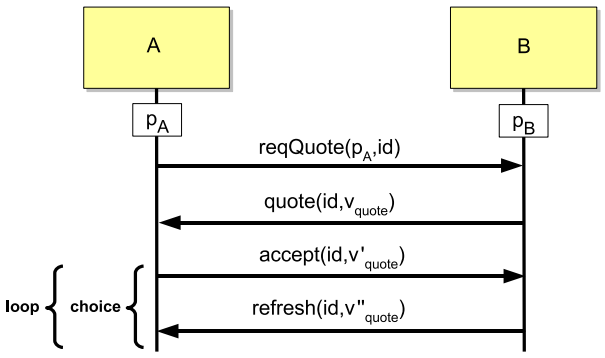

Fig. 4. Graphical representation of the Investment Bank interaction pattern

Investment Bank interaction pattern. We describe a typical interaction pattern in Investment Bank and other businesses, graphically represented in Figure 4. We consider two participants, A and B. A starts by requiring a quote to B, that answers with an initial quote. Then, B enters a loop, sending a new quote every 5 time units until A accepts a quote. Of course, in order to receive new quotes, also A cycles until it sends the quote acceptance message to B. Services A and B are modelled as follows:

$$
\begin{aligned}
A \triangleq\ & p_B \cdot o_{reqQuote}!\langle p_A, id\rangle \\
& \mid [x_{quote}]\, p_A \cdot o_{quote}?\langle id, x_{quote}\rangle\,. \\
& \qquad [\hat{n}]\,(\,\hat{n}!\langle x_{quote}\rangle \\
& \qquad\quad \mid *\,[x]\,\hat{n}?\langle x\rangle\,. \\
& \qquad\qquad [x_{new}]\,(\,\oplus_{rand()}\,.\, p_B \cdot o_{accept}!\langle id, x\rangle \\
& \qquad\qquad\qquad + \; p_A \cdot o_{refresh}?\langle id, x_{new}\rangle\,.\,\hat{n}!\langle x_{new}\rangle\,)\,)
\end{aligned}
$$

$$
\begin{aligned}
B \triangleq\ & *\,[x_A, x_{id}]\;\; p_B \cdot o_{reqQuote}?\langle x_A, x_{id}\rangle\,. \\
& \qquad (\, x_A \cdot o_{quote}!\langle x_{id}, v_{quote}\rangle \\
& \qquad\quad \mid [\hat{n}]\,(\,\hat{n}!\langle v_{quote}\rangle \\
& \qquad\qquad \mid *\,[x]\,\hat{n}?\langle x\rangle\,. \\
& \qquad\qquad\quad [x_{quote}]\,(\, p_B \cdot o_{accept}?\langle x_{id}, x_{quote}\rangle \\
& \qquad\qquad\qquad + \oplus_5\,.\,(\, x_A \cdot o_{refresh}!\langle x_{id}, newQuote(x)\rangle \\
& \qquad\qquad\qquad\quad \mid \hat{n}!\langle newQuote(x)\rangle\,)\,)\,)\,)
\end{aligned}
$$

Function $newQuote(\_)$, given the last quote sent from B to A, computes and returns a new quote. Notably, in both services, the iterative behaviour is modelled by means of a private endpoint (i.e. $\hat{n}$) and the replication operator. At each iteration, A waits a randomly chosen period of time, whose value is returned by function *rand*(), before replying to B. If this time interval is longer than 5 time units, a receive on operation $o_{refresh}$ triggers a new iteration.

Now, consider the system $A \mid B$. If the participant A does not accept the current quote in 5 time units, then a new quote is produced by the participant B, because its timeout has certainly expired. Instead, if we consider the system $\{A\} \mid \{B\}$, the clock of B can be slower than that of A, thus the production of a new quote is not ensured.

7 Concluding remarks

We have introduced COWS, a formalism for specifying and combining services, while modelling their dynamic behaviour (i.e. it deals with service orchestration rather than choreography). COWS borrows many constructs from well-known process calculi, e.g. π-calculus, update calculus, StAC$_i$, and Lπ, but combines them in an original way, thus being different from all existing calculi. COWS permits modelling different and typical aspects of (web) services technologies, such as multiple start activities, receive conflicts, routing of correlated messages, service instances and interactions among them. We have also presented an extension of the basic language with timed constructs.

The correlation mechanism was first exploited in [36], that, however, only considers interaction among different instances of a single business process. Instead, to connect the interaction protocols of clients and of the respective service instances, SCC [3] relies on the explicit modelling of sessions and their dynamic creation (that exploits the mechanism of private names of π-calculus). Interaction sessions are not explicitly modelled in COWS, instead they can be identified by tracing all those exchanged messages that are correlated each other through their same contents (as in [16]). We believe that the mechanism based on correlation sets (also used by WS-BPEL), that exploits business data and communication protocol headers to correlate different interactions, is more robust and fits the loosely coupled world of Web Services better than that based on explicit session references. Another notable difference with SCC is that in COWS services are not necessarily persistent.

Many works put forward enrichments of some well-known process calculus with constructs inspired by those of WS-BPEL. The most of them deal with issues of web transactions such as interruptible processes, failure handlers and time. This is, for example, the case of [22, 23, 27, 28] that present timed and untimed extensions of the π-calculus, called webπ and webπ_∞, tailored to study a simplified version of the scope construct of WS-BPEL. Other proposals on the formalization of flow compensation are [5, 4] that give a more compact and closer description of the *Sagas* mechanism [15] for dealing with long running transactions.

We have focused on service orchestration rather than on service choreography. In [6, 7] both aspects are studied. Other approaches are based on the use of schema languages [14] and Petri nets [17]. In [21] a sort of distributed input-guarded choice of join patterns, called *smooth orchestrators*, gives a simple and effective representation of synchronization constructs. The work closest to ours is [25], where ws-calculus is introduced to formalize the semantics of WS-BPEL. COWS represents a more foundational formalism than ws-calculus in that it does not rely on explicit notions of location and state, it is more manageable (e.g. has a simpler operational semantics) and, at least, equally expressive (as the encoding of ws-calculus in COWS shows, Section 5.3).

As a future work, we plan to continue our programme to lay rigorous methodological foundations for specification and validation of service oriented computing middlewares and applications by developing more powerful type systems. For example, session types and behavioural types are emerging as powerful tools for taking into account behavioural and non-functional properties of computing systems and, in case of services, could permit to express and enforce policies for, e.g., disciplining resources usage, constraining the sequences of messages accepted by services, ensuring service

interoperability and compositionality, guaranteeing absence of deadlock in service composition, checking that interaction obeys a given protocol. Some of the studies developed for π-calculus and other standard process calculi (e.g. [10, 37, 19, 18, 20]) are promising starting points, but they need non trivial adaptations to deal with all COWS peculiar features. For example, one of the major problems we envisage concerns the treatment of killing and protection activities, that are not commonly used in process calculi.

Another promising approach that we plan to investigate is the definition of behavioural equivalences that would provide a precise account of what 'observable behaviour' means for a service. Pragmatically, they could provide a means to establish formal correspondences between different views (abstraction levels) of a service, e.g. the contract it has to honour and its true implementation.

References

1. A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C.K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. Technical report, WS-BPEL TC OASIS, August 2006. `http://www.oasis-open.org/`.
2. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *FMOODS*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.
3. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. SCC: a Service Centered Calculus. In *WS-FM*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.
4. R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *CONCUR*, volume 3653 of *LNCS*, pages 383–397. Springer, 2005.
5. R. Bruni, H.C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL*, pages 209–220. ACM, 2005.
6. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In *ICSOC*, volume 3826 of *LNCS*, pages 228–240. Springer, 2005.
7. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION*, volume 4038 of *LNCS*, pages 63–81. Springer, 2006.
8. M.J. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *COORDINATION*, volume 2949 of *LNCS*, pages 87–104. Springer, 2004.
9. M.J. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, volume 3525 of *LNCS*, pages 133–150. Springer, 2005.
10. M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. In *DCM'06*, 2006. To appear as ENTCS.
11. M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. Structured global programming for communication behaviour. 2006.
12. M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming. Technical report, W3C, 2006. Long version to appear as W3C working note.
13. M. Carbone and S. Maffeis. On the expressive power of polyadic synchronisation in π-calculus. *Nordic J. of Computing*, 10(2):70–98, 2003.
14. S. Carpineti and C. Laneve. A basic contract language for web services. In *ESOP*, volume 3924 of *LNCS*, pages 197–213. Springer, 2006.
15. H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, pages 249–259. ACM Press, 1987.
16. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.
17. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to petri nets. In *Business Process Management*, volume 3649, pages 220–235, 2005.
18. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.
19. N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 439–453. Springer, 2003.
20. N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the π-calculus. In *VMCAI*, volume 3855 of *LNCS*, pages 298–312. Springer, 2006.

21. C. Laneve and L. Padovani. Smooth orchestrators. In *FoSSaCS*, volume 3921 of *LNCS*, pages 32–46. Springer, 2006.
22. C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.
23. C. Laneve and G. Zavattaro. web-pi at work. In *TGC*, volume 3705 of *LNCS*, pages 182–194. Springer, 2005.
24. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *ESOP*, 2006. To appear. `http://www.dsi.unifi.it/~pugliese/DOWNLOAD/cows.pdf`.
25. A. Lapadula, R. Pugliese, and F. Tiezzi. A WSDL-based type system for WS-BPEL. In *COORDINATION*, volume 4038 of *LNCS*, pages 145–163. Springer, 2006.
26. A. Lapadula, R. Pugliese, and F. Tiezzi. A WSDL-based type system for WS-BPEL (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2006. Available at: `http://www.dsi.unifi.it/~pugliese/ DOWNLOAD/wsc-full.ps`.
27. M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *WS-FM*, volume 4184 of *LNCS*, pages 257–272. Springer, 2006.
28. M. Mazzara and R. Lucchi. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2006.
29. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.
30. R. Milner. *Communication and concurrency.* Prentice-Hall, 1989.
31. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.
32. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May 2006. Published online.
33. J. Parrow and B. Victor. The update calculus. In *AMAST*, volume 1349 of *LNCS*, pages 409–423. Springer, 1997.
34. J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Logic in Computer Science*, pages 176–185, 1998.
35. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
36. M. Viroli. Towards a formal foundational to orchestration languages. *ENTCS*, 105:51–71, 2004.
37. N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *1st International Workshop on Security and Rewriting Techniques*, ENTCS, 2006.