C🕒WS: A timed service-oriented calculus*

Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi

Dipartimento di Sistemi e Informatica Università degli Studi di Firenze
{lapadula,pugliese,tiezzi}@dsi.unifi.it

Abstract. COWS (*Calculus for Orchestration of Web Services*) is a foundational language for Service Oriented Computing that 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. In this paper, we extend COWS with timed orchestration constructs, this way we obtain a language capable of completely formalizing the semantics of WS-BPEL, the 'de facto' standard language for orchestration of web services. We present the semantics of the extended language and illustrate its peculiarities and expressiveness by means of several examples.

1 Introduction

Service-Oriented Computing (SOC) is an emerging computing paradigm that uses loosely coupled 'services' to support the development of interoperable, evolvable systems and applications, and exploits the pervasiveness of the Internet technologies. Services are computational entities made available on a network as autonomous, platform-independent resources that can be described, published, discovered, and dynamically assembled, as the basic blocks for building applications. Companies like IBM, Microsoft and Sun have invested a lot of efforts to promote their deployment on Web Services, that are one of the present more successful instantiation of the SOC paradigm.

Many research efforts are currently addressed to define clean semantic models and rigorous methodological foundations for SOC applications. A main line of research aims at developing process calculi-like formalisms that provides in a distilled form the paradigm at the heart of SOC (see, e.g., [2–5, 9, 10, 12, 13, 16]). Most of these formalisms, however, do not model the different aspects of currently available SOC technologies in their completeness. One such aspect is represented by *timed activities* that are frequently exploited in service orchestration and are typically used for handling timeouts. For example, in WS-BPEL [21], timeouts turn out to be essential for dealing with service transactions or with message losses. Thus, a service process could wait a callback message for a certain amount of time after which, if no callback has been received, it invokes another operation or throws a fault. However, only a few process calculi for SOC deal with timed activities. In particular, [12, 13] introduce webπ, a timed extension of the π-calculus tailored to study 'web transactions'. [8, 9] present a timed calculus based on a more general notion of time, and an approach to verify WS-BPEL

* This work has been supported by the EU project SENSORIA, IST-2 005-016004.

specifications with compensation/fault constructs. [11] proposes a general purpose task orchestration language that manages timeouts as signals returned by dedicated services after some specified time intervals. Furthermore, all these formalisms, do not take into account such fundamental aspects of SOC as service instantiation and correlation.

To meet the demands arising from modelling SOC middlewares and applications, in [15] we have introduced COWS (*Calculus for Orchestration of Web Services*), a new modelling language that takes its origin from linguistic formalisms with opposite objectives, namely from WS-BPEL, the ‘de facto’ standard language for orchestration of web services, and from well-known process calculi, that represent a cornerstone of current foundational research on specification and analysis of concurrent and mobile systems. In [14] we show that COWS can model different and typical aspects of (web) services technologies, such as, e.g., multiple start activities, receive conflicts, routing of correlated messages, service instances and interactions among them. In this paper, since it is not known to what extent timed computation can be reduced to untimed forms of computation [22], we extend COWS with timed activities. Specifically, we introduce a WS-BPEL-like *wait* activity, that causes execution of the invoking service to be suspended until the time interval specified as an argument has elapsed, and permit using it to choose among alternative behaviours, alike the WS-BPEL *pick activity*. This way, the resulting language, that we call C◷WS (*timed* COWS), can faithfully capture also the semantics of WS-BPEL timed constructs.

For modelling time and timeouts, we draw again our inspiration from the rich literature on timed process calculi (see, e.g., [7, 20] for a survey). Thus, in C◷WS, basic actions are *durationless*, i.e. instantaneous, and the passing of time is modelled by using explicit actions, like in TCCS [18]. Moreover, actions execution is *lazy*, i.e. can be delayed arbitrary long in favour of passing of time, like in lTCCS [19]. Finally, since many distributed systems offer only weak guarantees on the upper bound of inter-location clock drift [1], passing of time is modelled synchronously for services deployed on a same ‘service engine’, and asynchronously otherwise.

The rest of the paper is organized as follows. The syntax of C◷WS is presented in Section 2, while its operational semantics is introduced in Section 3. Section 4 presents an extension that makes it explicit the notion of service engine and of deployment of services on engines. Section 5 illustrates three example applications of our framework and Section 6 concludes the paper. We refer the interested reader to [14] for further motivations on the design of COWS and C◷WS, for many examples illustrating their peculiarities and expressiveness, for comparisons with other process-based and orchestration formalisms, and for the presentation of a variant of the wait activity, that suspends the invoking service until the absolute time reaches its argument value.

2 C◷WS syntax

The syntax of C◷WS, 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, and a

Table 1. C◷WS syntax

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

set of positive numbers (ranged over by $\delta, \delta', \ldots$), used to represent *time intervals*. 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. Notation $\bar{\cdot}$ stands for tuples of objects, e.g. $\bar{x}$ is a shortening for 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.

Partner names and operation names can be combined to designate *communication endpoints*, written $p \bullet o$, and can be communicated, but dynamically received names can only be used for service invocation (as in the Lπ [17]). Indeed, communication endpoints of receive activities are identified statically because their syntax only allows using names and not variables. *Services* are structured activities built from basic activities, i.e. the empty activity **0**, the kill activity **kill**(_), the invoke activity _ • _!_, the receive activity _•_?_ and the wait activity ◷$_{\_}$, by means of prefixing _._, choice _ + _, parallel composition _ | _, protection $\{\!|\_|\!\}$, delimitation [_] _ and replication $*$ _. The major difference with COWS is that the choice construct can be guarded both by receive activities and by wait activities. In particular, the *wait activity* ◷$_e$ specifies the time interval, whose value is given by evaluation of e, the executing service has to wait for. We adopt the following conventions about the operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice. We shall omit trailing occurrences of **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. In fact, to enable concurrent threads within each service instance to share (part of) the state, receive activities in C◷WS bind neither names nor variables, which is different from most process calculi. Instead, the range of application of the substitutions generated by a communication is regulated by the delimitation operator, that additionally permits to generate fresh names (as the restriction operator of the π-calculus) and to delimit the field of action of kill activities. Thus, the occurrence of a name/variable/label is *free* if it is not under the scope of a delimitation for it. We denote by fk(t) the set of killer labels that occur free in t, and by fd(t) that of free names/variables/killer labels in t. Two terms

Table 2. C◷WS structural congruence (excerpt of laws)

$* \mathbf{0} \equiv \mathbf{0}$	$* s \equiv s \mid * s$	$\{\!\mid \mathbf{0} \mid\!\} \equiv \mathbf{0}$
$\{\!\mid \{\!\mid s \mid\!\} \mid\!\} \equiv \{\!\mid s \mid\!\}$	$\{\!\mid [d]\, s \mid\!\} \equiv [d]\, \{\!\mid s \mid\!\}$	$[d]\, \mathbf{0} \equiv \mathbf{0}$
$[d_1]\, [d_2]\, s \equiv [d_2]\, [d_1]\, s$	$s_1 \mid [d]\, s_2 \equiv [d]\, (s_1 \mid s_2)$	if $d \notin \mathrm{fd}(s_1) \cup \mathrm{fk}(s_2)$

Table 3. Matching rules

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

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.

3 C◷WS operational semantics

The operational semantics of C◷WS is defined over an enriched set of services that also includes those auxiliary services where the argument of wait activities can also be 0. Moreover, the semantics 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. Formally, the semantics is given in terms of a structural congruence and of a labelled transition relation. We assume that evaluation of expressions and execution of basic activities, except for ◷$_e$, are instantaneous (i.e. do not consume time units) and that time elapses between them.

The structural congruence ≡ 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 **0** as identity element, and that guarded choice enjoys the same properties and, additionally, 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, the last law 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

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

$\mathbf{kill}(k) \downarrow_{kill}$	$\dfrac{s \downarrow_{kill} \ \vee \ s' \downarrow_{kill}}{s \mid s' \downarrow_{kill}}$	$\dfrac{s \downarrow_{kill}}{\{\!\mid s \mid\!\} \downarrow_{kill}}$	$\dfrac{s \downarrow_{kill}}{[d]\, s \downarrow_{kill}}$	$\dfrac{s \downarrow_{kill}}{*\, s \downarrow_{kill}}$
$\dfrac{\mid \mathcal{M}(\bar{w}, \bar{v}) \mid < \ \ell}{p \bullet o ? \bar{w}.s \downarrow^{\ell}_{p \bullet o, \bar{v}}}$	$\dfrac{s \downarrow^{\ell}_{p \bullet o, \bar{v}} \quad d \notin \{p, o\}}{[d]\, s \downarrow^{\ell}_{p \bullet o, \bar{v}}}$	$\dfrac{s \downarrow^{\ell}_{p \bullet o, \bar{v}}}{\{\!\mid s \mid\!\} \downarrow^{\ell}_{p \bullet o, \bar{v}}}$		
$\dfrac{g \downarrow^{\ell}_{p \bullet o, \bar{v}} \ \vee \ g' \downarrow^{\ell}_{p \bullet o, \bar{v}}}{g + g' \downarrow^{\ell}_{p \bullet o, \bar{v}}}$	$\dfrac{s \downarrow^{\ell}_{p \bullet o, \bar{v}} \ \vee \ s' \downarrow^{\ell}_{p \bullet o, \bar{v}}}{s \mid s' \downarrow^{\ell}_{p \bullet o, \bar{v}}}$	$\dfrac{s \downarrow^{\ell}_{p \bullet o, \bar{v}}}{*\, s \downarrow^{\ell}_{p \bullet o, \bar{v}}}$		

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 $\mid \sigma \mid$ 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 $halt(\{\!\mid s \mid\!\}) = \{\!\mid s \mid\!\}$. In the other cases, *halt*(_) returns $\mathbf{0}$, except for parallel composition, delimitation and replication operators, for which it acts as an homomorphism.

$$halt(\mathbf{kill}(k)) \ = \ halt(u_1 \bullet u_2 ! \bar{e}) \ = \ halt(g) \ = \ \mathbf{0} \qquad halt(s_1 \mid s_2) = halt(s_1) \mid halt(s_2)$$

$$halt(\{\!\mid s \mid\!\}) = \{\!\mid s \mid\!\} \qquad halt([d]\, s) = [d]\, halt(s) \qquad halt(*\, s) = *\, halt(s)$$

Finally, in Table 4, we inductively define two predicates: $s \downarrow_{kill}$ checks if s can immediately perform a kill activity; $s \downarrow^{\ell}_{p \bullet o, \bar{v}}$, 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.

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

$$\begin{array}{lcl} \hat{\alpha} & ::= & \alpha \ \mid \ \delta \\ \alpha & ::= & \dagger k \ \mid \ (p \bullet o) \lhd \bar{v} \ \mid \ (p \bullet o) \rhd \bar{w} \ \mid \ p \bullet o \lfloor \sigma \rfloor \, \bar{w}\, \bar{v} \ \mid \ \dagger \end{array}$$

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. $\hat{\alpha}$ denotes taking place of *computational activities* α or *time elapsing* δ (recall that δ is a

Table 5. C◷WS operational semantics

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

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

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

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

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

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

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

$$\mathbf{0} \xrightarrow{\delta} \mathbf{0} \; \textit{(nil}_{\textit{elaps}}\textit{)} \qquad\qquad * \, s \xrightarrow{\delta} * \, s \; \textit{(repl)} \qquad\qquad u \bullet u'!\bar{e} \xrightarrow{\delta} u \bullet u'!\bar{e} \; \textit{(inv}_{\textit{elaps}}\textit{)}$$

$$\frac{s \xrightarrow{\delta} s'}{\{\!|s|\!\} \xrightarrow{\delta} \{\!|s'|\!\}} \; \textit{(prot}_{\textit{elaps}}\textit{)} \qquad\qquad \frac{s \xrightarrow{\delta} s'}{[d]\, s \xrightarrow{\delta} [d]\, s'} \; \textit{(scope}_{\textit{elaps}}\textit{)} \qquad\qquad p \bullet o?\bar{w}.s \xrightarrow{\delta} p \bullet o?\bar{w}.s \; \textit{(rec}_{\textit{elaps}}\textit{)}$$

$$\text{◷}_0.s \xrightarrow{\dagger} s \; \textit{(wait}_{\textit{tout}}\textit{)} \qquad\qquad \frac{\llbracket e \rrbracket \neq \delta'}{\text{◷}_e.s \xrightarrow{\delta} \text{◷}_e.s} \; \textit{(wait}_{\textit{err}}\textit{)} \qquad\qquad \frac{\delta \leqslant \llbracket e \rrbracket}{\text{◷}_e.s \xrightarrow{\delta} \text{◷}_{\llbracket e - \delta \rrbracket}.s} \; \textit{(wait}_{\textit{elaps}}\textit{)}$$

$$\frac{g_1 \xrightarrow{\delta} g_1' \qquad g_2 \xrightarrow{\delta} g_2'}{g_1 + g_2 \xrightarrow{\delta} g_1' + g_2'} \; \textit{(pick)} \qquad\qquad \frac{s_1 \xrightarrow{\delta} s_1' \qquad s_2 \xrightarrow{\delta} s_2'}{s_1 \mid s_2 \xrightarrow{\delta} s_1' \mid s_2'} \; \textit{(par}_{\textit{sync}}\textit{)}$$

time interval). $\dagger k$ denotes execution of a request for terminating a term from within the delimitation $[k]$. $(p \bullet o) \lhd \bar{v}$ and $(p \bullet o) \rhd \bar{w}$ 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 $\bar{w}$, matching values $\bar{v}$ and substitution

σ to be still applied. † and $p \bullet o \lfloor \emptyset \rfloor\, \bar{w}\, \bar{v}$ denote taking place of timeout/forced termination or communication (without pending substitutions), respectively. A *computation* from a closed service s_0 is a sequence of connected transitions of the form

$$s_0 \xrightarrow{\hat{\alpha}_1} s_1 \xrightarrow{\hat{\alpha}_2} s_2 \xrightarrow{\hat{\alpha}_3} s_3 \ldots$$

where, for each i, $\hat{\alpha}_i$ can be δ, † or $p \bullet o \lfloor \emptyset \rfloor\, \bar{w}\, \bar{v}$ (for some $p, o, \bar{w}$ and $\bar{v}$).

The rules in the upper part of Table 5 model computational activities, those in the lower part model time passing. We prefer to keep separate the two sets of rules to make it evident that C◷WS is a ‘conservative’ extension of COWS (indeed, the rules in the upper part are exactly those of COWS). We now 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 † (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 is active in s and α does not correspond to a kill activity or a timeout (rule *(del$_{pass}$)*): in such cases the transition should be derived by using rules *(del$_{sub}$)* or *(del$_{kill}$)*. 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 then one matching is possible, the receive that needs fewer substitutions is selected to progress (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$_{conf}$)*). Rule *(cong)* states that structurally congruent services have the same transitions.

Time can elapse while waiting on invoke/receive activities, rules *(inv$_{elaps}$)* and *(rec$_{elaps}$)*. When time elapses, but the timeout is still not expired, the argument of wait activities ◷ is updated (rule *(wait$_{elaps}$)*). Time elapsing cannot make a choice within a pick activity (rule *(pick)*), while the occurrence of a timeout can. This is signalled by label † (thus, it is a computation step), that is 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 *(nil$_{elaps}$)*, the wait activity *(wait$_{err}$)*, replication *(repl)*, protection *(prot$_{elaps}$)* and delimitation *(scope$_{elaps}$)*. Furthermore, 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. Notably, in agreement with its eager semantics, the kill activity does not allow time to pass.

We end this section with a simple example aimed at clarifying some peculiarities of our formalism and at specifying *timeouts* as described in [11, 22]. Consider the service:

$$\begin{array}{l}[x,y,k]\,(\,p\bullet o_1?\langle x\rangle.(\,p\bullet o_2?\langle x,y\rangle + \text{⏲}_{10}.\mathbf{kill}(k)\,)\ \mid\ \{\!|p'\bullet o_3!\langle x\rangle|\!\}\ \mid\ p'\bullet o_4!\langle x,y\rangle\,)\\ \mid\ [n]\,p\bullet o_1!\langle n\rangle\end{array}$$

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

$$\begin{array}{ll}[n,x]\,(\,[y,k]\,(\,p\bullet o_1?\langle x\rangle.(\,p\bullet o_2?\langle x,y\rangle + \text{⏲}_{10}.\mathbf{kill}(k)\,) & \\ \qquad\qquad \mid\ \{\!|p'\bullet o_3!\langle x\rangle|\!\}\ \mid\ p'\bullet o_4!\langle x,y\rangle\,) & \xrightarrow{p\bullet o_1\,\lfloor\emptyset\rfloor\,\langle x\rangle\,\langle n\rangle} \\ \qquad \mid\ p\bullet o_1!\langle n\rangle\,) & \\ [n,y,k]\,(\,(\,p\bullet o_2?\langle n,y\rangle + \text{⏲}_{10}.\mathbf{kill}(k)\,)\ \mid\ \{\!|p'\bullet o_3!\langle n\rangle|\!\}\ \mid\ p'\bullet o_4!\langle n,y\rangle\,) & \xrightarrow{6} \\ [n,y,k]\,(\,(\,p\bullet o_2?\langle n,y\rangle + \text{⏲}_{4}.\mathbf{kill}(k)\,)\ \mid\ \{\!|p'\bullet o_3!\langle n\rangle|\!\}\ \mid\ p'\bullet o_4!\langle n,y\rangle\,) & \xrightarrow{4} \\ [n,y,k]\,(\,(\,p\bullet o_2?\langle n,y\rangle + \text{⏲}_{0}.\mathbf{kill}(k)\,)\ \mid\ \{\!|p'\bullet o_3!\langle n\rangle|\!\}\ \mid\ p'\bullet o_4!\langle n,y\rangle\,) & \xrightarrow{\dagger} \\ [n,y,k]\,(\,\mathbf{kill}(k)\ \mid\ \{\!|p'\bullet o_3!\langle n\rangle|\!\}\ \mid\ p'\bullet o_4!\langle n,y\rangle\,) & \xrightarrow{\dagger} \\ [n]\,\{\!|p'\bullet o_3!\langle n\rangle|\!\} & \end{array}$$

When the communication takes place, a timer starts and the substitution $\{x\mapsto n\}$ is applied to all terms delimited by $[x]$, not only to the continuation of the service performing the receive. Then, the time elapses until the timeout expires, this way the receive along $p\bullet o_2$ is discarded. Finally, the kill activity removes the unprotected invoke activity.

4 Service deployment

In the language presented so far, time passes synchronously for all services in parallel, thus we can think of as all services run on a same service *engine*. As a consequence, the services share the same clock and can be tightly coupled. Instead, existing SOC systems are loosely coupled because they are usually deployed on top of distributed systems that offer only weak guarantees on the upper bound of inter-location clock drift. Consider for example a scenario including a customer service and a service provider composed of two subservices – one used as an interface to interact with external services, the other being an internal service that performs queries in a database – sharing a tuple of variables, operations, ... The scenario could be modelled by the term

$$customer\ \mid\ [\bar{d}_{shared}]\,(\,provider\_interface \mid provider\_internal\_service\,)$$

The customer and the provider service are loosely coupled and can be deployed on different engines, while the provider subservices are tightly coupled and must be colocated. To emphasize these aspects, we introduce explicitly the notions of service engine and of *deployment* of services on engines and we will write the previous term as follows:

$$\{\, customer \,\} \ \mid\ \{\, [\bar{d}_{shared}]\, (\, provider\_interface \mid provider\_internal\_service \,)\, \}$$

Formally, we extend the language syntax with the syntactic category of (*service*) *engines* (alike the 'machines' of [12]) defined as follows:

$$\mathbb{E} \ ::= \ 0 \quad | \quad \{s\} \quad | \quad [n]\,\mathbb{E} \quad | \quad \mathbb{E} \,|\, \mathbb{E}$$

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 3 with the abelian monoid laws for engines parallel composition and with the following laws:

$$\begin{array}{llll} \{s\} \ \equiv\ \{s'\} \quad \text{if } s \ \equiv\ s' & \{\mathbf{0}\} \ \equiv\ 0 & \{[n]\, s\} \ \equiv\ [n]\,\{s\} & [n]\,0 \ \equiv\ 0 \\ [n]\,[m]\,\mathbb{E} \ \equiv\ [m]\,[n]\,\mathbb{E} & \multicolumn{3}{l}{\mathbb{E} \mid [n]\,\mathbb{F} \ \equiv\ [n]\,(\mathbb{E} \mid \mathbb{F}) \quad \text{if } n \notin \mathrm{fd}(\mathbb{E})} \end{array}$$

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 $\longrightarrow$ among engines through the rules shown in Table 6. 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 considering 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

Table 6. Operational semantics of C⏲WS plus engines (additional rules)

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

$$\frac{\mathbb{E} \equiv \mathbb{E}' \qquad \mathbb{E}' \longrightarrow \mathbb{F}' \qquad \mathbb{F}' \equiv \mathbb{F}}{\mathbb{E} \longrightarrow \mathbb{F}}\ \textit{(cong}_E\textit{)} \qquad\qquad \frac{\mathbb{E} \longrightarrow \mathbb{E}'}{\mathbb{E} \mid \mathbb{F} \longrightarrow \mathbb{E}' \mid \mathbb{F}}\ \textit{(par}_{async}\textit{)}$$

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

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$_E$)*.

5 Examples

In this section, we illustrate three applications of our framework. The first one is an example of a web service inspired by the well-known game *Rock/Paper/Scissors*, while the remaining ones are use-cases inspired by [6]. In the sequel, we will write $Z \triangleq W$ to assign a symbolic name Z to the term W. We will use $\hat{n}$ to stand for the endpoint $n_p \bullet n_o$ and, sometimes, we will write $\hat{n}$ for the tuple $\langle n_p, n_o \rangle$ and rely on the context to resolve any ambiguity. For the sake of readability, in the examples we will use assignment and conditional choice constructs. They can be thought of as 'macros' corresponding to the following C⏲WS encodings

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

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

where $\hat{m}$ is fresh, and **true** and **false** are the values that can result from evaluation of e.

Rock/Paper/Scissors service. Consider the following service:

$$\begin{aligned}
rps \ \triangleq\ * \,[& x_{champ\_res}, x_{chall\_res}, x_{id}, x_{thr\_1}, x_{thr\_2}, x_{win}, k] \\
&(\,(\, p_{champ} \bullet o_{throw}?\langle x_{champ\_res}, x_{id}, x_{thr\_1} \rangle. \\
&\qquad\quad (\, p_{chall} \bullet o_{throw}?\langle x_{chall\_res}, x_{id}, x_{thr\_2} \rangle. \\
&\qquad\qquad\qquad (\, x_{champ\_res} \bullet o_{win}!\langle x_{id}, x_{win} \rangle \mid x_{chall\_res} \bullet o_{win}!\langle x_{id}, x_{win} \rangle\,) \\
&\qquad\quad + \text{⏲}_{30} \,.\, (\, \lbrace\!\vert x_{champ\_res} \bullet o_{win}!\langle x_{id}, x_{champ\_res} \rangle \vert\!\rbrace \mid \mathbf{kill}(k)\,)\,) \\
&\quad + \ p_{chall} \bullet o_{throw}?\langle x_{chall\_res}, x_{id}, x_{thr\_2} \rangle. \\
&\qquad\quad (\, p_{champ} \bullet o_{throw}?\langle x_{champ\_res}, x_{id}, x_{thr\_1} \rangle. \\
&\qquad\qquad\qquad (\, x_{champ\_res} \bullet o_{win}!\langle x_{id}, x_{win} \rangle \mid x_{chall\_res} \bullet o_{win}!\langle x_{id}, x_{win} \rangle\,) \\
&\qquad\quad + \text{⏲}_{30} \,.\, (\, \lbrace\!\vert x_{chall\_res} \bullet o_{win}!\langle x_{id}, x_{chall\_res} \rangle \vert\!\rbrace \mid \mathbf{kill}(k)\,)\,)\,) \\
&\ \mid Assign\,)
\end{aligned}$$

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. An instance of service 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. 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

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

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

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

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:

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

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

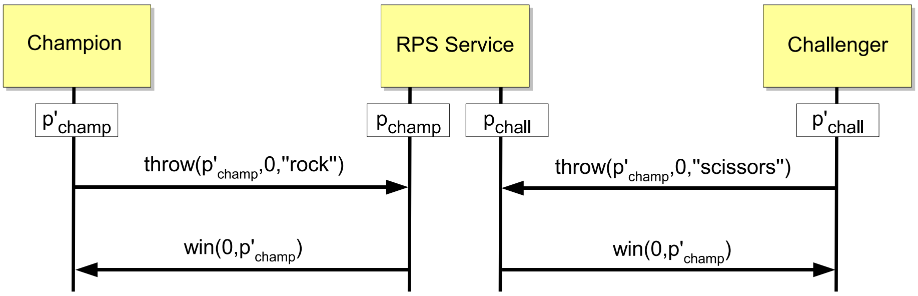

Fig. 1. Graphical representation of the Rock/Paper/Scissors service scenario

pion's throw:

$$\begin{aligned}
& s \xrightarrow{p_{champ} \bullet o_{throw} \lfloor\emptyset\rfloor \langle x_{champ\_res}, x_{id}, x_{thr\_1}\rangle \langle p'_{champ}, 0, \text{“}rock\text{”}\rangle} \\
& rps \mid [x_{chall\_res}, x_{thr\_2}, x_{win}, k] \\
& \qquad\quad ((\, p_{chall} \bullet o_{throw}?\langle x_{chall\_res}, 0, x_{thr\_2}\rangle . \\
& \qquad\qquad\quad (\, p'_{champ} \bullet o_{win}!\langle 0, x_{win}\rangle \;\mid\; x_{chall\_res} \bullet o_{win}!\langle 0, x_{win}\rangle\,) \\
& \qquad\qquad + \text{🕒}_{30} . (\, \{\!\!| p'_{champ} \bullet o_{win}!\langle 0, p'_{champ}\rangle |\!\!\} \mid \mathbf{kill}(k)\,)\,) \\
& \qquad\quad\; \mid \mathit{Assign} \cdot \{x_{champ\_res} \mapsto p'_{champ}, x_{id} \mapsto 0, x_{thr\_1} \mapsto \text{“}rock\text{”}\}\,) \\
& \quad \mid [x]\, p'_{champ} \bullet o_{win}?\langle 0, x\rangle . s_{champ} \\
& \quad \mid p_{chall} \bullet o_{throw}!\langle p'_{chall}, 0, \text{“}scissors\text{”}\rangle \mid [y]\, p'_{chall} \bullet o_{win}?\langle 0, y\rangle . s_{chall} \;\triangleq\; s'
\end{aligned}$$

In case the challenger's throw is not consumed within the deadline, the timeout expires:

$$\begin{aligned}
& s' \xrightarrow{30}\xrightarrow{\dagger} \\
& rps \mid [x_{chall\_res}, x_{thr\_2}, x_{win}, k] \\
& \qquad (\, \{\!\!| p'_{champ} \bullet o_{win}!\langle 0, p'_{champ}\rangle |\!\!\} \mid \mathbf{kill}(k) \\
& \qquad\; \mid \mathit{Assign} \cdot \{x_{champ\_res} \mapsto p'_{champ}, x_{id} \mapsto 0, x_{thr\_1} \mapsto \text{“}rock\text{”}\}\,) \\
& \quad \mid [x]\, p'_{champ} \bullet o_{win}?\langle 0, x\rangle . s_{champ} \\
& \quad \mid p_{chall} \bullet o_{throw}!\langle p'_{chall}, 0, \text{“}scissors\text{”}\rangle \mid [y]\, p'_{chall} \bullet o_{win}?\langle 0, y\rangle . s_{chall}
\end{aligned}$$

Then, the kill activity terminates the instance and the champion is declared to be the winner. Instead, if the challenger's throw is consumed by the existing instance within the deadline, the service evolves as follows:

$$\begin{aligned}
& s' \xrightarrow{5} \xrightarrow{p_{chall} \bullet o_{throw} \lfloor\emptyset\rfloor \langle x_{chall\_res}, 0, x_{thr\_2}\rangle \langle p'_{chall}, 0, \text{“}scissors\text{”}\rangle} \\
& rps \mid [x_{win}]\,(\, p'_{champ} \bullet o_{win}!\langle 0, x_{win}\rangle \mid p'_{chall} \bullet o_{win}!\langle 0, x_{win}\rangle \\
& \qquad\qquad\qquad \mid \mathit{Assign} \cdot \{x_{champ\_res} \mapsto p'_{champ}, x_{id} \mapsto 0, x_{thr\_1} \mapsto \text{“}rock\text{”}, \\
& \qquad\qquad\qquad\qquad\qquad x_{chall\_res} \mapsto p'_{chall}, x_{thr\_2} \mapsto \text{“}scissors\text{”}\}\,) \\
& \quad \mid [x]\, p'_{champ} \bullet o_{win}?\langle 0, x\rangle . s_{champ} \\
& \quad \mid [y]\, p'_{chall} \bullet o_{win}?\langle 0, y\rangle . s_{chall}
\end{aligned}$$

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

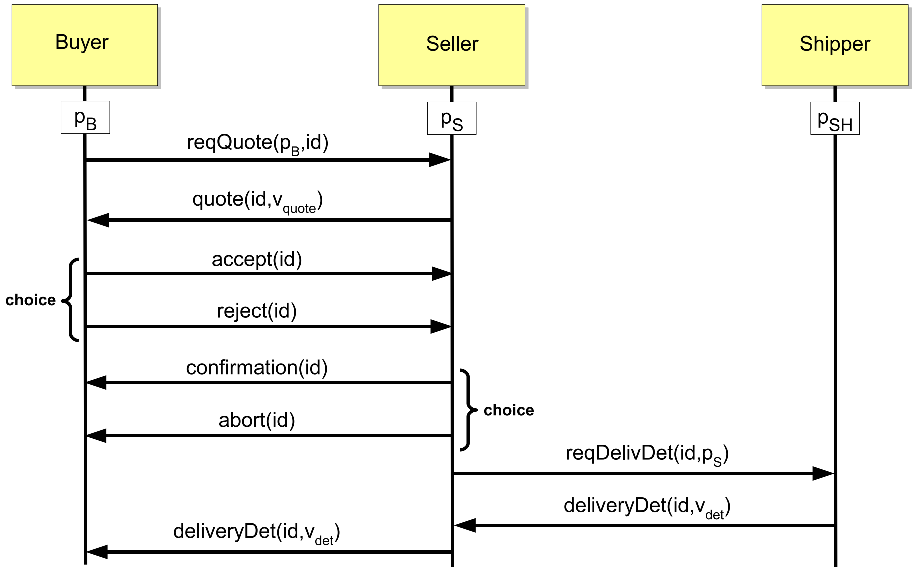

Fig. 2. Graphical representation of the Buyer/Seller/Shipper protocol

that p_{champ} won, the substitutive effects of communication transforms the system as 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:

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

A Buyer/Seller/Shipper protocol. We illustrate a simple business protocol for purchasing a fixed good. The protocol, graphically represented in Figure 2, 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

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

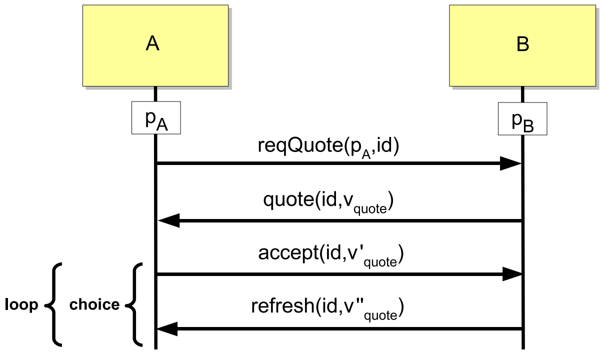

Fig. 3. Graphical representation of the Investment Bank interaction pattern

where

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

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

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

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.

Investment Bank interaction pattern. We describe a typical interaction pattern in Investment Bank and other businesses, graphically represented in Figure 3. 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:

$$
\begin{array}{l}
A \triangleq p_B \bullet o_{reqQuote}!\langle p_A, id\rangle \\
\quad\;\; \mid [x_{quote}]\, p_A \bullet o_{quote}?\langle id, x_{quote}\rangle\,. \\
\qquad\qquad\quad [\hat{n}]\,(\,\hat{n}!\langle x_{quote}\rangle \\
\qquad\qquad\qquad\;\; \mid *\,[x]\,\hat{n}?\langle x\rangle\,. \\
\qquad\qquad\qquad\qquad\quad [x_{new}]\,(\,\text{🕒}_{rand()}\,.\,p_B \bullet o_{accept}!\langle id, x\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad +\;\; p_A \bullet o_{refresh}?\langle id, x_{new}\rangle\,.\,\hat{n}!\langle x_{new}\rangle\,)\,)
\end{array}
$$

$$
\begin{array}{l}
B \triangleq *\,[x_A, x_{id}]\;\; p_B \bullet o_{reqQuote}?\langle x_A, x_{id}\rangle\,. \\
\qquad\qquad\quad (\,x_A \bullet o_{quote}!\langle x_{id}, v_{quote}\rangle \\
\qquad\qquad\quad\;\; \mid\; [\hat{n}]\,(\,\hat{n}!\langle v_{quote}\rangle \\
\qquad\qquad\qquad\qquad \mid *\,[x]\,\hat{n}?\langle x\rangle\,. \\
\qquad\qquad\qquad\qquad\quad [x_{quote}]\,(\,p_B \bullet o_{accept}?\langle x_{id}, x_{quote}\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad + \text{🕒}_5\,.\,(\,x_A \bullet o_{refresh}!\langle x_{id}, newQuote(x)\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid\; \hat{n}!\langle newQuote(x)\rangle\,)\,)\,)\,)\,)
\end{array}
$$

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.

6 Concluding remarks

We have introduced C🕒WS, a formalism for specifying and combining services, while modelling their dynamic behaviour. We have first considered a language where all services are implicitly allocated on a same engine. Then, we have presented an extension with explicit notions of service engine and of deployment of services on engines.

We plan to continue our programme to lay rigorous methodological foundations for specification and validation of SOC middlewares and applications. We are currently working on formalizing the semantics of WS-BPEL through labelled transition systems. We intend then to prove that this semantics and that defined by translation in COWS do agree. Of course, the extension presented in this paper will be essential to faithfully capture the semantics of WS-BPEL timed constructs. As a further work, we want to develop type systems and behavioural equivalences capable of dealing also with time aspects. Pragmatically, they could provide a means to express and guarantee time-based QoS properties of services (such as, e.g., time to reply to service requests), that should be published in service contracts.

Acknowledgements. We thank the anonymous referees for their useful comments.

References

1. M. Berger. Basic theory of reduction congruence for two timed asynchronous pi-calculi. In *CONCUR*, volume 3170 of *LNCS*, pages 115–130. Springer, 2004.
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. 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.
5. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
6. 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.
7. F. Corradini, D. D'Ortenzio, and P. Inverardi. On the relationships among four timed process algebras. *Fundam. Inform.*, 38(4):377–395, 1999.
8. P. Geguang, Z. Huibiao, Q. Zongyan, W. Shuling, Z. Xiangpeng, and H. Jifeng. Theoretical foundations of scope-based compensable flow language for web service. In *FMOODS*, pages 251–266, 2006.
9. P. Geguang, Z. Xiangpeng, W. Shuling, and Q. Zongyan. Towards the semantics and verification of bpel4ws. In *WLFM 2005*. Elsevier, 2005.
10. 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.
11. D. Kitchin, W.R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, volume 4137 of *LNCS*, pages 477–491. Springer, 2006.
12. C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.
13. C. Laneve and G. Zavattaro. web-pi at work. In *TGC*, volume 3705 of *LNCS*, pages 182–194. Springer, 2005.
14. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2006. `http://rap.dsi.unifi.it/cows`.
15. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
16. M. Mazzara and R. Lucchi. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2006.
17. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.
18. F. Moller and C. Tofts. A temporal calculus of communicating systems. In *CONCUR*, pages 401–415, 1990.
19. F. Moller and C. Tofts. Relating processes with respect to speed. In *CONCUR*, pages 424–438, 1991.
20. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *CAV*, volume 575 of *LNCS*, pages 376–398. Springer, 1991.
21. OASIS. Web Services Business Process Execution Language Version 2.0. Technical report, WS-BPEL TC OASIS, August 2006. `http://www.oasis-open.org/`.
22. R.J. van Glabbeek. On specifying timeouts. *ENTCS*, 162:173–175, 2006.