A Calculus for Orchestration of Web Services*

Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi

Dipartimento di Sistemi e Informatica Università degli Studi di Firenze

Abstract. We introduce COWS (*Calculus for Orchestration of Web Services*), a new foundational language for SOC whose design has been influenced by WS-BPEL, the *de facto* 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.

1 Introduction

Web services are a successful instantiation of service-oriented computing (SOC), an emerging paradigm for developing loosely coupled, interoperable, evolvable systems and applications which exploits the pervasiveness of the Internet and its related technologies. Web services are autonomous, stateless, platform-independent and composable computational entities that can be published, located and invoked through the Web via XML messages. 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, 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'.

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

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

R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 33–47, 2007.

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.

Recently, 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 [2,19,20] and of CSP with compensation [9]).

Here, we follow a different approach and exploit WS-BPEL [1], the *de facto* 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 presents the encoding of the orchestration language Orc [28]. Finally, Section 5 touches upon comparisons with related work and directions for future work.

2 COWS: Calculus for Orchestration of Web Services

The basic elements of COWS are *partners* and *operations*. Alike channels in [10], 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). 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. 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 [25], 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 [29,30]. In COWS, however, inter-service communication give rise to substitutions of variables with values (alike [29]), rather than to fusions of names (as in [30]). 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 [27]) 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]).

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 **0**, the kill activity **kill**(_) , the invoke activity _ · _!_ and the receive activity _ · _?_ , by means of prefixing _ . _, choice _ + _, parallel composition _ | _, protection {|_|}, delimitation [_] _ and replication * _. Notably, as in the Lπ [25], communication endpoints of receive

Table 1. COWS syntax

$s ::=$	$\mathbf{kill}(k)$	$\mid$	$u \cdot u'!\bar{e}$	$\mid$	g	$\mid$	$s \mid s$	$\mid$	$\{\!\mid s \mid\!\}$	$\mid$	$[d]\, s$	$\mid$	$* \, s$	(services)
$g ::=$	$\mathbf{0}$	$\mid$	$p \cdot o?\bar{w}.s$	$\mid$	$g + g$									(input-guarded choice)

Table 2. COWS 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 $d \notin \mathrm{fd}(s_1) \cup \mathrm{fk}(s_2)$	

activities are identified statically because their syntax only allows using names and not variables. The decreasing order of precedence among the operators is as follows: monadic operators, choice and parallel composition.

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. $p \cdot o?\bar{w}$ instead of $p \cdot o?\bar{w}.\mathbf{0}$, 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.

Operational Semantics. COWS operational 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 (see e.g. the premises of rules *(del_)*). 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, 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).

Table 3. Matching rules

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

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.

Finally, we define a predicate, $noc(\_, \_, \_, \_)$, that takes a service s, an endpoint $p \cdot o$, a tuple of receive parameters $\bar{w}$ and a matching tuple of values $\bar{v}$ as arguments and holds true if either there are no conflicting receives within s (namely, s cannot immediately perform a receive activity matching $\bar{v}$ over the endpoint $p \cdot o$), or $p \cdot o?\bar{w}$ is the most defined conflicting receive. The predicate exploits the notion of *active context*, namely a service $\mathbb{A}$ with a 'hole' $[\![\cdot]\!]$ such that, once the hole is filled with a service s, if the resulting term $\mathbb{A}[\![s]\!]$ is a COWS service then it is capable of immediately performing an activity of s. Formally, active contexts are generated by the grammar:

$$\mathbb{A} ::= [\![\cdot]\!] \mid \mathbb{A} + g \mid g + \mathbb{A} \mid \mathbb{A} \mid s \mid s \mid \mathbb{A} \mid \{\!|\mathbb{A}|\!\} \mid [d]\,\mathbb{A} \mid * \mathbb{A}$$

Now, predicate $noc(s, p \cdot o, \bar{w}, \bar{v})$ can be defined as follows:

$$(\, s = \mathbb{A}[\![p \cdot o?\bar{w}'.s']\!] \;\wedge\; \mathcal{M}(\bar{w}', \bar{v}) = \sigma \,) \;\Rightarrow |\mathcal{M}(\bar{w}, \bar{v})| \leqslant |\sigma|$$

where $s = \mathbb{A}[\![p \cdot o?\bar{w}'.s']\!]$ means that s can be written as $p \cdot o?\bar{w}'.s'$ filling the hole of some active context $\mathbb{A}$.

The labelled transition relation $\xrightarrow{\alpha}$ is the least relation over services induced by the rules in Table 4, where label α is generated by the following grammar:

$$\alpha ::= \dagger k \quad | \quad (p \cdot o) \lhd \bar{v} \quad | \quad (p \cdot o) \rhd \bar{w} \quad | \quad p \cdot o\, \lfloor \sigma \rfloor\, \bar{w}\, \bar{v} \quad | \quad \dagger$$

Table 4. COWS operational semantics

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

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

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

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

$$\frac{s_1 \xrightarrow{(p\cdot o)\rhd \bar{w}} s_1' \qquad s_2 \xrightarrow{(p\cdot o)\lhd \bar{v}} s_2' \qquad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \qquad \mathit{noc}(s_1 \mid s_2, p \cdot o, \bar{w}, \bar{v})}{s_1 \mid s_2 \xrightarrow{p\cdot o\lfloor \sigma\rfloor \bar{w}\,\bar{v}} s_1' \mid s_2'} \quad \textit{(com)}$$

$$\frac{s_1 \xrightarrow{p\cdot o\lfloor \sigma\rfloor \bar{w}\,\bar{v}} s_1' \qquad \mathit{noc}(s_2, p \cdot o, \bar{w}, \bar{v})}{s_1 \mid s_2 \xrightarrow{p\cdot o\lfloor \sigma\rfloor \bar{w}\,\bar{v}} s_1' \mid s_2} \quad \textit{(par}_{conf}\textit{)} \qquad \frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid \mathit{halt}(s_2)} \quad \textit{(par}_{kill}\textit{)}$$

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

In the sequel, we use $\mathrm{d}(\alpha)$ to denote the set of names, variables and killer labels occurring in α, except for $\alpha = p \cdot o\lfloor \sigma\rfloor \bar{w}\,\bar{v}$ for which we let $\mathrm{d}(p \cdot o\lfloor \sigma\rfloor \bar{w}\,\bar{v}) = \mathrm{d}(\sigma)$, where $\mathrm{d}(\{x \mapsto v\}) = \{x, v\}$ and $\mathrm{d}(\sigma_1 \uplus \sigma_2) = \mathrm{d}(\sigma_1) \cup \mathrm{d}(\sigma_2)$. The meaning of labels is as follows: $\dagger k$ denotes execution of a request for terminating a term from within the delimitation $[k]$, $(p \cdot o) \lhd \bar{v}$ and $(p \cdot o) \rhd \bar{w}$ denote execution of invoke and receive activities over the endpoint $p \cdot o$, respectively, $p \cdot o\lfloor \sigma\rfloor \bar{w}\,\bar{v}$ (if $\sigma \neq \emptyset$) denotes execution of a communication over $p \cdot o$ with receive parameters $\bar{w}$ and matching values $\bar{v}$ and with substitution σ to be still applied, $\dagger$ and $p \cdot o\lfloor \emptyset\rfloor \bar{w}\,\bar{v}$ 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 $p \cdot o\lfloor \emptyset\rfloor \bar{w}\,\bar{v}$ (for some $p, o, \bar{w}$ and $\bar{v}$); services s_i, for each i, will be called *reducts* of s_0.

We comment on salient points. Activity $\mathbf{kill}(k)$ 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)*). Receive activities can always proceed (rule *(rec)*) and can resolve choices (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.

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.[1] 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:

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

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 accounts for the global scope of variables and permits to easily model the *delayed input* of fusion calculus [30].

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

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

[1] The variant of π-calculus closest to COWS is localised π-calculus [25] and, indeed, in [21] we define an encoding that enjoys *operational correspondence*.

where, for simplicity, we assume that $halt(s_1) = halt(s_3) = \mathbf{0}$. 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).

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

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

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

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

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

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

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:

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

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 $\bar{v}$. We use $\hat{n}$ to stand for the endpoint $n_p \cdot n_o$. Sometimes, we write $\hat{n}$ for the tuple $\langle n_p, n_o\rangle$ and rely on the context to resolve any ambiguity.

Imperative constructs. Due to lack of space, we only present the encodings of those constructs that will be further exploited in the rest of the section. We refer the interested reader to [21] for deeper explanations and additional encodings.

We start adding *matching with assignment* $[\bar{w} = \bar{e}]$ to COWS basic activities. If $\bar{w}$ and $\bar{e}$ do match, service $[\bar{w} = \bar{e}].s$ returns a substitution that will eventually assign to the variables in $\bar{w}$ the corresponding values of $\bar{e}$, and service s can proceed. In COWS, this meaning can be rendered through the following encoding (for $\hat{m}$ fresh)

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

Notably, the new construct differs from standard assignment both because values can occur on the left of =, in which case it behaves as a matching mechanism, and because, like the receive activity, it does not bind the variables on the left of =, thus it cannot reassign a value to them if a value has already been assigned (more details are in [21]).

Conditional choice is encoded similarly:

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

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

Sequential composition can be encoded alike in CCS [26, 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 $x_{done} \cdot o_{done}!\langle\rangle$, 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 partner p, $s' \cdot \{x_{done} \mapsto p\} \xrightarrow{(p \cdot o_{done}) \lhd \langle\rangle}$ implies that

- either $s' \xrightarrow{\alpha} s''$ for some $\alpha = \dagger$ or $\alpha = \dagger k$ and s'' is well-terminating
- or $s' \xrightarrow{\alpha} s''$ implies $\alpha = (p \cdot o) \lhd \bar{v}$, for some s'', p, o and $\bar{v}$.

Notably, well-termination does not demand a service to terminate, but only that whenever the service can perform activity $p \cdot o_{done}!\langle\rangle$ and cannot perform any kill activities, then it terminates except for, possibly, some parallel pending invoke activities. As usual, the encoding relies on the assumption that all calculus operators themselves (in particular, parallel composition) can be rendered as to preserve well-termination. Finally, if we only consider well-terminating services, then, for a fresh p, we can let:

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

Fault and compensation handlers. 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 Table 5. The *scope* activity $[s : \textbf{catch}(\phi_1)\{s_1\} : \ldots : \textbf{catch}(\phi_n)\{s_n\} : s_c]_\iota$ 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 **throw**(ϕ) 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 **catch**(ϕ)$\{s'\}$ exists within the same scope. The *compensate* activity **undo**(ι) 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 **undo**(ι) occurring in a service there exists at least an inner scope ι.

In fact, it is not necessary to extend COWS syntax because fault and compensation handling can be easily encoded. The most interesting cases of the encoding are shown in the lower part of Table 5 (in the remaining cases, the encoding acts as an homomorphism), where the killer labels used to identify scopes and the introduced partner

Table 5. Syntax and encoding of fault and compensation handling

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

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

names are taken fresh for s, s_1, …, s_n and s_c. The two distinguished names o_{fault} and o_{comp} denote the operations for receiving fault and compensation signals, respectively. We are assuming that for each scope identifier or fault signal named n, the partner used to activate scope compensation or fault handling, respectively, is p_n.

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 $p_\iota \cdot o_{comp}!\langle\rangle$ occurs. Similarly, if during normal execution a fault ϕ occurs, a signal $p_\phi \cdot o_{fault}!\langle\rangle$ triggers execution of the corresponding fault handler (if any). Installed compensation handlers are protected from killing by means of $\{\!|\_|\!\}$. Notably, both the compensate activity and the fault generator activity can immediately terminate (thus enabling possible sequential compositions); this, of course, does not mean that the corresponding handler is terminated.

Flow graphs. Flow graphs 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 effect is called *Dead-Path Elimination* [1]).

To express the constructs above, we extend the syntax of COWS as illustrated in the upper part of Table 6. A *flow graph activity* $[\overline{fl}]\, ls$ is a delimited *linked service*, where

Table 6. Syntax and encoding of flow graphs

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

$$
\begin{array}{l}
\langle\!\langle [\overline{fl}]\, ls \rangle\!\rangle = [\overline{fl}]\, \langle\!\langle ls \rangle\!\rangle \qquad \langle\!\langle ls_1 \mid ls_2 \rangle\!\rangle = \langle\!\langle ls_1 \rangle\!\rangle \mid \langle\!\langle ls_2 \rangle\!\rangle \qquad \langle\!\langle s \Rightarrow (\overline{fl}, \bar{e}) \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle; [\overline{fl} = \bar{e}] \\
\\
\langle\!\langle (jc) \overset{yes}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e}) \rangle\!\rangle = \mathbf{if}\ (jc)\ \mathbf{then}\ \{\langle\!\langle s \rangle\!\rangle; [\overline{fl} = \bar{e}]\}\ \mathbf{else}\ \{[outLinkOf(s) = \overline{\mathbf{false}}]\} \\
\\
\langle\!\langle (jc) \overset{no}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e}) \rangle\!\rangle = \mathbf{if}\ (jc)\ \mathbf{then}\ \{\langle\!\langle s \rangle\!\rangle; [\overline{fl} = \bar{e}]\}\ \mathbf{else}\ \{\mathbf{throw}(\phi_{join\_f})\} \\
\\
\langle\!\langle \sum_{i\in\{1..n\}} p_i \cdot o_i?\bar{w}_i.s_i \rangle\!\rangle = p_1 \cdot o_1?\bar{w}_1.[\bigcup_{j\in\{2..n\}} outLinkOf(s_j) = \overline{\mathbf{false}}].\langle\!\langle s_1 \rangle\!\rangle \\
\qquad\qquad + \ldots + p_n \cdot o_n?\bar{w}_n.[\bigcup_{j\in\{1..n-1\}} outLinkOf(s_j) = \overline{\mathbf{false}}].\langle\!\langle s_n \rangle\!\rangle
\end{array}
$$

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 relying on 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 6. The encoding exploits the auxiliary function $outLinkOf(s)$, that returns the tuple of outgoing links in s. 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 encoding of conditional choice can be modified similarly).

4 Encoding the Orchestration Language Orc

We present here the encoding of Orc [28], a recently proposed task orchestration language with applications in workflow, business process management, and web service orchestration. Orc *expressions* are generated by the following grammar:

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

where S ranges over *site names*, E over *expression names*, x over variables, and w over *parameters*, i.e. variables or values (ranged over by v). Each expression name E has a unique declaration of the form $E(x) \triangleq f$. Expressions $f > x > g$ and g **where** $x :\in f$ bind variable x in g.

We now briefly describe the semantics of Orc expressions (and refer the interested reader to [21] for a formal account). Evaluation of expressions may call a number of sites and returns a (possibly empty) stream of values. In [28], this is formalized through a labelled transition relation, where label τ indicates an internal event while label $!v$ indicates publication of the value v resulting from evaluating an expression. A *site call* can progress only when the actual parameter is a value; it elicits one response. While site calls use a call-by-value mechanism, *expression calls* use a call-by-name mechanism, 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. *Sequential composition* $f > x > g$ activates a concurrent copy of g with x replaced by v, for each value v returned by f. *Asymmetric parallel composition* g **where** $x :\in f$ starts in parallel both f and the part of g that does not need x. The first value returned by f is assigned to x and the continuation of f and all its descendants are then terminated.

The encoding of Orc expressions in COWS exploits function $\langle\!\langle \cdot \rangle\!\rangle_{\hat{r}}$ shown in Table 7. 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:

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

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_x^S, an unspecified expression corresponding to S and depending on x.

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

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

Table 7. Orc encoding

$\langle\!\langle \mathbf{0} \rangle\!\rangle_{\hat{r}} = \mathbf{0}$	$\langle\!\langle S(w) \rangle\!\rangle_{\hat{r}} = \hat{S}!\langle w, \hat{r} \rangle$	$\langle\!\langle E(w) \rangle\!\rangle_{\hat{r}} = [\hat{r}'](\hat{E}!\langle \hat{r}, \hat{r}' \rangle \mid [z]\,\hat{r}'?\langle z \rangle.z!\langle w \rangle)$
$\langle\!\langle f > x > g \rangle\!\rangle_{\hat{r}} = [\hat{r}_f](\langle\!\langle f \rangle\!\rangle_{\hat{r}_f} \mid *[x]\,\hat{r}_f?\langle x \rangle.\langle\!\langle g \rangle\!\rangle_{\hat{r}})$		$\langle\!\langle f \mid g \rangle\!\rangle_{\hat{r}} = \langle\!\langle f \rangle\!\rangle_{\hat{r}} \mid \langle\!\langle g \rangle\!\rangle_{\hat{r}}$
$\langle\!\langle g \textbf{ where } x :\in f \rangle\!\rangle_{\hat{r}} = [\hat{r}_f, x]\ (\ \langle\!\langle g \rangle\!\rangle_{\hat{r}} \mid [k](\ \langle\!\langle f \rangle\!\rangle_{\hat{r}_f} \mid \hat{r}_f?\langle x \rangle.\textbf{kill}(k)\ )\ )$		

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.

In [21], we prove that there is a formal correspondence, based on the operational semantics, between Orc expressions and the COWS services resulting from their encoding. This is another sign of COWS expressiveness because it is known that Orc can express the most common workflow patterns identified in [31]. 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 above property can be stated as follows

Theorem 1. *Given an Orc expression f and an 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 $\alpha = \hat{r} \lhd \langle v \rangle$ if $l = !v$, and $\alpha = (p \cdot o \lfloor \emptyset \rfloor \bar{w}\,\bar{v})$ if $l = \tau$.*

The proof (see [21]) proceeds by induction on the length of the inference of $f \stackrel{l}{\hookrightarrow} f'$.

5 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.

The correlation mechanism was first exploited in [32], 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, the calculus introduced in [3], and called SCC, relies on 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 [14]). 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 [19,20,23,24] that present timed and untimed extensions of the π-calculus, called $\mathtt{web}\pi$ and $\mathtt{web}\pi_\infty$, 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 [13] 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 [11] and Petri nets [15]. In [18] 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 [22], 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, [21]).

This paper has focussed on showing the descriptive power of COWS. We leave as a future work the task of developing a formal account of its expressiveness. We also plan to develop analytical tools, such as e.g. behavioural equivalences and type systems, supporting services verification. Behavioural equivalences 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. Type systems, possibly based on behavioural types (see e.g. [12,16,17]), could permit to express and enforce policies of interest for (web) services 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.

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

References

1. A. Alves et al. 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*, *LNCS* 2884, pp. 124–138, 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*, *LNCS* 4184 , pp. 38–57, 2006.
4. R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *CONCUR*, *LNCS* 3653, pp. 383–397, 2005.
5. R. Bruni, H.C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL*, pp. 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*, *LNCS* 3826, pp. 228–240, 2005.
7. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION*, *LNCS* 4038, pp. 63–81, 2006.
8. M.J. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *COORDINATION*, *LNCS* 2949, pp. 87–104, 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*, *LNCS* 3525, pp. 133–150, 2005.
10. M. Carbone and S. Maffeis. On the expressive power of polyadic synchronisation in π- calculus. *Nordic J. of Computing*, 10(2):70–98, 2003.
11. S. Carpineti and C. Laneve. A basic contract language for web services. In *ESOP*, *LNCS* 3924, pp. 197–213, 2006.
12. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL*, pp. 45–57, 2002.
13. H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, pp. 249–259. ACM Press, 1987.
14. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *ICSOC*, *LNCS* 4294, pp. 327–338, 2006.
15. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to petri nets. In *Business Process Management* 3649, pp. 220–235, 2005.
16. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.
17. N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the π-calculus. In *VMCAI*, *LNCS* 3855, pp. 298–312, 2006.
18. C. Laneve and L. Padovani. Smooth orchestrators. In *FoSSaCS*, *LNCS* 3921, pp. 32–46, 2006.
19. C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, *LNCS* 3441, pp. 282–298, 2005.
20. C. Laneve and G. Zavattaro. web-pi at work. In *TGC*, *LNCS* 3705, pp. 182–194, 2005.
21. 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`
22. A. Lapadula, R. Pugliese, and F. Tiezzi. A WSDL-based type system for WS-BPEL. In *COORDINATION*, *LNCS* 4038, pp. 145–163, 2006.
23. M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *WS-FM*, *LNCS* 4184, pp. 257–272, 2006.
24. M. Mazzara and R. Lucchi. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2006.
25. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.
26. R. Milner. *Communication and concurrency.* Prentice-Hall, 1989.
27. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.
28. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*. Springer, May 2006.
29. J. Parrow and B. Victor. The update calculus. In *AMAST*, *LNCS* 1349, pp. 409–423, 1997.
30. J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Logic in Computer Science*, pp. 176–185, 1998.
31. 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. Springer, 2003.
32. M. Viroli. Towards a formal foundational to orchestration languages. *ENTCS*, 105:51–71. Elsevier, 2004.