Stochastic COWS[⋆]

Davide Prandi and Paola Quaglia

Dipartimento di Informatica e Telecomunicazioni, Università di Trento, Italy

Abstract. A stochastic extension of COWS is presented. First the formalism is given an operational semantics leading to finitely branching transition systems. Then its syntax and semantics are enriched along the lines of Markovian extensions of process calculi. This allows addressing quantitative reasoning about the behaviour of the specified web services. For instance, a simple case study shows that services can be analyzed using the PRISM probabilistic model checker.

1 Introduction

Interacting via web services is becoming a programming paradigm, and a number of languages, mostly based on XML, has been designed for, e.g., coordinating, orchestrating, and querying services. While the design of those languages and of supporting tools is quickly improving, the formal underpinning of the programming paradigm is still uncertain.

This calls for the investigation of models that can ground the development of methodologies, techniques, and tools for the rigorous analysis of service properties. Recent works on the translation of web service primitives into well-understood formal settings (e.g., [2, 3]), as well as on the definition of process calculi for the specification of web service behaviours (e.g., [6, 8]), go in this direction. These approaches, although based on languages still quite far from WS-BPEL, WSFL, WSCI, or WSDL, bring in the advantage of being based on clean semantic models. For instance, process calculi typically come with a structural operational semantics in Plotkin's style: The dynamic behaviour of a term of the language is represented by a connected oriented graph (called *transition system*) whose nodes are the reachable states of the system, and whose paths stay for its possible runs. This feature is indeed one of the main reasons why process calculi have been extensively used over the years for the specification and verification of distributed systems. One can guess that the same feature could also be useful to reason about the dynamic behaviour of web services. The challenge is appropriately tuning calculi and formal techniques to this new interaction paradigm.

In this paper we present a stochastic extension of COWS [8] (Calculus for Orchestration of Web Services), a calculus strongly inspired by WS-BPEL which combines primitives of well-known process calculi (like, e.g., the π-calculus [9, 16]) with constructs meant to model web services orchestration. For instance,

⋆ This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

besides the expected request/invoke communication primitives, COWS has operators to specify protection, delimited receiving activities, and killing activities. A number of other interesting constructs, although not taken as primitives of the language, have been shown to be easily encoded in COWS. This is the case, e.g., for fault and compensation handlers [8].

The operational semantics of COWS provides a full qualitative account on the behaviour of services specified in the language. Quantitative aspects of computation, though, are as crucial to SOC as qualitative ones (think, e.g., of quality of service, resource usage, or service level agreement). In this paper, we first present a version of the operational semantics of COWS that, giving raise to finitely branching transition systems, is suitable to stochastic reasoning (Sec. 2). The syntax and semantics of the calculus is then enriched along the lines of Markovian extensions of process calculi [11, 5] (Sec. 3). Basic actions are associated with a random duration governed by a negative exponential distribution. In this way the semantic models associated to services result to be Continuous Time Markov Chains, popular models for automated verification. To give a flavour of our approach, we show how the stochastic model checker PRISM [14] can be used to check a few properties of a simple case study (Sec. 4).

2 Operational semantics of monadic COWS

We consider a monadic (vs polyadic) version of the calculus, i.e., it is assumed that request/invoke interactions can carry one single parameter at a time (vs multiple parameters). This simplifies the presentation without impacting on the sort of primitives the calculus is based on, and indeed our setting could be generalized to the case of polyadic communications. Some other differences between the operational approach used in [8] and the one provided here are due to the fact that, for the effective application of Markovian techniques, we need to guarantee that the generated transition system is finitely branching. In order to ensure this main property we chose to express recursive behaviours by means of service identifiers rather than by replication. Syntactically, this is the single deviation from the language as presented in [8]. From the semantic point of view, though, some modifications of the operational setting are also needed. They will be fully commented upon below.

The syntax of COWS is based on three countable and pairwise disjoint sets: the set of *names* $\mathcal{N}$ (ranged over by $m, n, o, p, m', n', o', p'$), the set of *variables* $\mathcal{V}$ (ranged over by x, y, x', y'), and the set of *killer labels* $\mathcal{K}$ (ranged over by k, k'). Services are expressed as structured activities built from basic activities that involve elements of the above sets. In particular, request and invoke activities occur at endpoints, which in [8] are identified by both a *partner* and an *operation* name. Here, for ease of notation, we let endpoints be denoted by single identifiers. In what follows, u, v, w, u', v', w' are used to range over $\mathcal{N} \cup \mathcal{V}$, and d, d' to range over $\mathcal{N} \cup \mathcal{V} \cup \mathcal{K}$. Names, variables, and killer labels are collectively referred to as *entities*.

The terms of the COWS language are generated by the following grammar.

$$\begin{aligned} s &::= u\,!\,w \mid g \mid s \mid s \mid \{\!|s|\!\} \mid \mathbf{kill}(k) \mid [\,d\,]s \mid S(n_1,\ldots,n_j) \\ g &::= \mathbf{0} \mid p\,?\,w.\,s \mid g+g \end{aligned}$$

where, for some service s, a defining equation $S(n_1,\ldots,n_j) = s$ is given.

A service s can consist in an asynchronous *invoke* activity over the endpoint u with parameter w ($u\,!\,w$), or it can be generated by a guarded choice. In this case it can either be the empty activity $\mathbf{0}$, or a choice between two guarded commands ($g+g$), or an input-guarded service $p\,?\,w.\,s$ that waits for a communication over the endpoint p and then proceeds as s after the (possible) instantiation of the input parameter w. Besides service identifiers like $S(n_1,\ldots,n_j)$, which are used to model recursive behaviours, the language offers a few other primitive operators: parallel composition ($s \mid s$), protection ($\{\!|s|\!\}$), kill activity ($\mathbf{kill}(k)$), and delimitation of the entity d within s ($[\,d\,]s$).

In $[\,d\,]s$ the occurrence of $[\,d\,]$ is a *binding* for d with scope s. An entity is *free* if it is not under the scope of a binder. It is *bound* otherwise. An occurrence of one term in a service is *unguarded* if it is not underneath a request.

Like in [8], the operational semantics of COWS is defined for *closed* services, i.e. for services whose variables and killer labels are all bound. Moreover, to be sure to get finitely branching transition systems, we work under two main assumptions. First, it is assumed that service identifiers do not occur unguarded. Second, we assume that there is no homonymy either among bound entities or among free and bound entities of the service under consideration. This condition can be initially met by appropriately refreshing the term, and is dynamically kept true by a suitable management of the unfolding of recursion.

The labelled transition relation $\xrightarrow{\alpha}$ between services is defined by the rules collected in Tab. 1 and by symmetric rules for the commutative operators of choice and of parallel composition. Labels α are given by the following grammar

$$\alpha ::= \dagger k \mid \dagger \mid p\,?\,w \mid p\,!\,n \mid p\,?\,(x) \mid p\,!\,(n) \mid p\cdot\sigma\cdot\sigma'$$

where, for some n and x, σ ranges over $\varepsilon, \{n/x\}, \{(n)/x\}$, and σ' over $\varepsilon, \{n/x\}$.

Label $\dagger k$ ($\dagger$) denotes that a request for terminating a term s in the delimitation $[\,k\,]s$ is being (was) executed. Label $p\,?\,w$ ($p\,!\,n$) stays for the execution of a request (an invocation) activity over the endpoint p with parameter w (n, respectively). Label $p\cdot\sigma\cdot\sigma'$ denotes a communication over the endpoint p. The two components σ and σ' of label $p\cdot\sigma\cdot\sigma'$ are meant to implement a *best-match* communication mechanism. Among the possibly many receives that could match the same invocation, priority of communication is given to the most defined one. This is achieved by possibly delaying the name substitution induced by the interaction, and also by preventing further moves after a name substitution has been improperly applied. To this end, σ' recalls the name substitution, and σ signals whether is has been already applied ($\sigma = \varepsilon$) or not. We observe that labels like $p\cdot\{(n)/x\}\cdot\sigma'$, just as $p\,?\,(x)$ and $p\,!\,(n)$, have no counterpart in [8]. These labels are used in the rules for scope opening and closure that have no analogue in [8] where scope modification is handled by means of a congruence relation. Their intuitive meaning is analogous to the one of the corresponding

$$\mathbf{kill}(k) \xrightarrow{\dagger k} \mathbf{0}\ (\mathit{kill}) \qquad p\,?\,w.\,s \xrightarrow{p\,?\,w} s\ (\mathit{req}) \qquad p\,!\,n \xrightarrow{p\,!\,n} \mathbf{0}\ (\mathit{inv})$$

$$\frac{g_1 \xrightarrow{\alpha} s}{g_1 + g_2 \xrightarrow{\alpha} s}\ (\mathit{choice}) \qquad \frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}}\ (\mathit{prot}) \qquad \frac{s_1 \xrightarrow{p\,!\,n} s_1' \qquad s_2 \xrightarrow{p\,?\,n} s_2'}{s_1 \mid s_2 \xrightarrow{p\cdot\varepsilon\cdot\varepsilon} s_1' \mid s_2'}\ (\mathit{com\_n})$$

$$\frac{s_1 \xrightarrow{p\,!\,n} s_1' \qquad s_2 \xrightarrow{p\,?\,x} s_2' \qquad (s_1 \mid s_2) \not\downarrow_{p\,?\,n}}{s_1 \mid s_2 \xrightarrow{p\cdot\{n/x\}\cdot\{n/x\}} s_1' \mid s_2'}\ (\mathit{com\_x})$$

$$\frac{s_1 \xrightarrow{p\cdot\sigma\cdot\sigma'} s_1' \quad \sigma' = \{n/x\} \Rightarrow s_2 \not\downarrow_{p\,?\,n}}{s_1 \mid s_2 \xrightarrow{p\cdot\sigma\cdot\sigma'} s_1' \mid s_2}\ (\mathit{par\_conf}) \qquad \frac{s \xrightarrow{p\cdot\{n/x\}\cdot\{n/x\}} s'}{[x]s \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} s'\{n/x\}}\ (\mathit{del\_sub})$$

$$\frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid \mathit{halt}(s_2)}\ (\mathit{par\_kill}) \qquad \frac{s_1 \xrightarrow{\alpha} s_1' \quad \alpha \neq p\cdot\sigma\cdot\sigma' \quad \alpha \neq \dagger k}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2}\ (\mathit{par\_pass})$$

$$\frac{s \xrightarrow{\dagger k} s'}{[k]s \xrightarrow{\dagger} [k]s'}\ (\mathit{del\_kill}) \qquad \frac{s \xrightarrow{\alpha} s' \quad d \notin \mathrm{d}(\alpha) \quad s\downarrow_d \Rightarrow (\alpha = \dagger \text{ or } \alpha = \dagger k)}{[d]s \xrightarrow{\alpha} [d]s'}\ (\mathit{del\_pass})$$

- -

$$\frac{s\{m_1\ldots m_j/n_1\ldots n_j\} \xrightarrow{\alpha} s' \qquad S(n_1,\ldots,n_j) = s}{S(m_1,\ldots,m_j) \xrightarrow{\mathrm{l\_dec}(\alpha)} \mathrm{s\_dec}(\alpha, \mathrm{s}')}\ (\mathit{ser\_id})$$

$$\frac{s \xrightarrow{p\,?\,x} s'}{[x]s \xrightarrow{p\,?\,(x)} s'}\ (\mathit{op\_req}) \qquad \frac{s_1 \xrightarrow{p\,!\,(n)} s_1' \qquad s_2 \xrightarrow{p\,?\,(x)} s_2' \qquad (s_1 \mid s_2) \not\downarrow_{p\,?\,n}}{s_1 \mid s_2 \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} [n](s_1' \mid s_2'\{n/x\})}\ (\mathit{cl\_nx})$$

$$\frac{s \xrightarrow{p\,!\,n} s'}{[n]s \xrightarrow{p\,!\,(n)} s'}\ (\mathit{op\_inv}) \qquad \frac{s_1 \xrightarrow{p\,!\,(n)} s_1' \qquad s_2 \xrightarrow{p\,?\,x} s_2' \qquad (s_1 \mid s_2) \not\downarrow_{p\,?\,n}}{s_1 \mid s_2 \xrightarrow{p\cdot\{(n)/x\}\cdot\{n/x\}} s_1' \mid s_2'}\ (\mathit{cl\_n})$$

$$\frac{s \xrightarrow{p\cdot\{(n)/x\}\cdot\{n/x\}} s'}{[x]s \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} [n]s'\{n/x\}}\ (\mathit{del\_cl}) \qquad \frac{s_1 \xrightarrow{p\,!\,n} s_1' \quad s_2 \xrightarrow{p\,?\,(x)} s_2' \quad (s_1 \mid s_2) \not\downarrow_{p\,?\,n}}{s_1 \mid s_2 \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} s_1' \mid s_2'\{n/x\}}\ (\mathit{cl\_x})$$

Table 1. Operational semantics of COWS.

labels $p \cdot \{n/x\} \cdot \sigma'$, $p\,?\,x$, and $p\,!\,n$. The parentheses only record that the scope of the entity is undergoing a modification.

Notation and auxiliary functions. We use $[\,d_1,\ldots,d_2\,]$ as a shorthand for $[\,d_1\,]\ldots[\,d_2\,]$, and adopt the notation $s\{d_1'\cdots d_j'/d_1\ldots d_j\}$ to mean the simultaneous substitution of d_is by d_i's in the term s . We write $s\downarrow_{p\,?\,n}$ if, for some s', an unguarded subterm of s has the shape $p\,?\,n.\,s'$. Analogously, we write $s\downarrow_k$ if some unguarded subterm of s has the shape $\mathbf{kill}(k)$. The predicates $s\not\downarrow_{p\,?\,n}$ and $s\not\downarrow_k$ are used as negations of $s\downarrow_{p\,?\,n}$ and of $s\downarrow_k$, respectively. Function $\mathit{halt}(\_)$, used to define service behaviours correspondingly to the execution of a kill activity, takes a service s and eliminates all of its unprotected subservices. In detail: $\mathit{halt}(u\,!\,w) = \mathit{halt}(g) = \mathit{halt}(\mathbf{kill}(k)) = \mathbf{0}$, and $\mathit{halt}(\{\!|s|\!\}) = \{\!|s|\!\}$. Function $\mathit{halt}(\_)$ is a homomorphism on the other operators, namely: $halt(s_1 \mid s_2) = halt(s_1) \mid halt(s_2)$, $halt([\,d\,]s) = [\,d\,]halt(s)$, and $halt(S(m_1, \ldots, m_j)) = halt(s\{^{m_1 \cdots m_j}/_{n_1 \ldots n_j}\})$ for $S(n_1, \ldots, n_j) = s$. Finally, an auxiliary function $\mathrm{d}(\_)$ on labels is defined. We let $\mathrm{d}(p \cdot \{n/x\} \cdot \sigma') = \mathrm{d}(p \cdot \{(n)/x\} \cdot \sigma') = \{n, x\}$ and $\mathrm{d}(p \cdot \varepsilon \cdot \sigma') = \emptyset$. For the other forms of labels, $\mathrm{d}(\alpha)$ stays for the set of entities occurring in α.

Tab. 1 defines $\xrightarrow{\alpha}$ for a rich class of labels. This is technically necessary to get what is actually taken as an *execution step* of a closed service:

$$s \xrightarrow{\alpha} s' \text{ with either } \alpha = \dagger \text{ or } \alpha = p \cdot \varepsilon \cdot \sigma'.$$

The upper portion of Tab. 1 displays the monadic version of rules which are in common with the operational semantics presented in [8]. We first comment on the most interesting rules of that portion.

The execution of the **kill**(k) primitive (axiom *kill*) results in spreading the killer signal $\dagger k$ that forces the termination of all the parallel services (rule *par_kill*) but the protected ones (rule *prot*). Once $\dagger k$ reaches the delimiter of its scope, the killer signal is turned off to $\dagger$ (rule *del_kill*). Kill activities are executed eagerly: Whenever a kill primitive occurs unguarded within a service s delimited by d, the service $[\,d\,]s$ can only execute actions of the form $\dagger k$ or $\dagger$ (rule *del_pass*).

Notice that, by our convention on the use of meta-entities, an invoke activity (axiom *inv*) cannot take place if its parameter is a variable. Variable instantiation can take place, involving the whole scope of variable x, due to a pending communication action of shape $p \cdot \{n/x\} \cdot \{n/x\}$ (rule *del_sub*). Communication allows the pairing of the invoke activity $p\,!\,n$ with either the best-matching activity $p\,?\,n$ (rule *com_n*), or with a less defined $p\,?\,x$ action if a best-match is not offered by the locally available context (rule *com_x*). A best-match for $p\,!\,n$ is looked for in the surrounding parallel services (rule *par_conf*) until either $p\,?\,n$ or the delimiter of the variable scope is found. In the first case the attempt to establish an interaction between $p\,!\,n$ and $p\,?\,x$ is blocked by the non applicability of the rules for parallel composition.

The rules in the lower portion of Tab. 1 are a main novelty w.r.t. [8]. In order to carry out quantitative reasoning on the behaviour of services we need to base our stochastic extension on a finitely branching transition system. This was not the case for the authors of [8] who defined their setting for modelling purposes, and hence were mainly interested in *runs* of services rather than on the complete description of their behaviour in terms of *graphs*. Indeed, in [8] the operational semantics of COWS is presented in the most elegant way by using both the replication operator and structural congruence. The rules described below are meant to get rid of both these two ingredients while retaining the expressive power of the language.

As said, we discarded the replication operator in favour of service identifiers. Their use, just as that of replication, is a typical way to allow recursion in the language. When replication is out of the language, the main issue about simulating the expressivity of structural congruence is relative to the management of scope opening for delimiters.

As an example, the operational semantics in [8] permits the interaction between the parallel components of service $[\,n\,]p\,!\,n \mid [\,x\,]p\,?\,x.\,\mathbf{0}$ because, by structural congruence, that parallel composition is exactly the same as $[\,n\,][\,x\,](p\,!\,n \mid p\,?\,x.\,\mathbf{0})$ and hence the transition $[\,n\,]p\,!\,n \mid [\,x\,]p\,?\,x.\,\mathbf{0} \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} [\,n\,](\mathbf{0} \mid \mathbf{0})$ is allowed.

Except for rule *ser_id*, all the newly introduced rules are meant to manage possible moves of delimiters without relying on a notion of structural congruence. The effect is obtained by using a mechanism for opening and closing the scope of binders that is analogous to the technique adopted in the definition of the labelled transition systems of the π-calculus.

Both rules *op_req* and *op_inv* open the scope of their parameter by removing the delimiter from the residual service and recording the binding in the transition label. The definition of the opening rules is where our assumption on the non-homonymy of entities comes into play. If not working under that assumption, we should care of possible name captures caused when closing the scope of the opened entity. To be sure to avoid this, we should allow the applicability of the opening rules to a countably infinite set of entities, which surely contrasts with our need to get finitely branching transition systems.

The idea underlying the opening/closing technique is the following. Opened activities can pass over parallel compositions till a (possibly best) match is found. When this happens, communication can take place and, if due, the delimiter is put back into the term to bind the whole of the residual service.

The three closing rules in Tab. 1 reflect the possible recombinations of pairs of request and invoke activities when at least one of them carries the information that the scope of its parameter has been opened. In each case the parameter of the request is a variable. (If it is a name then, independently on any assumption on entities, it is surely distinct from the invoke parameter.) Recombinations have to be ruled out in different ways depending on the relative original positions of delimiters and parallel composition.

Rule *cl_nx* takes care of scenarios like the one illustrated above for the service $[\,n\,]p\,!\,n \mid [\,x\,]p\,?\,x.\,\mathbf{0}$. Delimiters are originally distributed over the parallel operator, and their scope can be opened to embrace both parallel components. The single delimiter that reappears in the residual term is the one for n.

Rule *cl_x* regulates the case when only variable x underwent a scope opening. The delimiter for the invoke parameter, if present, is in outermost position w.r.t. both the delimiter for x and the parallel operator. An example of this situation is $p\,!\,n \mid [\,x\,]p\,?\,x.\,\mathbf{0}$. The invoke can still find a best matching, though. Think, e.g., of the service $(p\,!\,n \mid p\,?\,n.\,\mathbf{0}) \mid [\,x\,]p\,?\,x.\,\mathbf{0}$. If such matching is not available, then the closing communication can effectively occur and the variable gets instantiated.

Rule *cl_n* handles those scenarios when the delimiter for the invoke is within the scope of the delimiter for x, like, e.g., in $[\,x\,](p\,?\,x.\,\mathbf{0} \mid [\,n\,]p\,!\,n)$. Communication is left pending by executing $p\cdot\{(n)/x\}\cdot\{n/x\}$ which is passed over possible parallel compositions using the *par_conf* rule. Variable x is instantiated when $p\cdot\{(n)/x\}\cdot\{n/x\}$ reaches the delimiter for x (rule *del_cl*). On the occasion, $[\,x\,]$ becomes a delimiter for n.

```
NS(p1,m1) | NS(p2,m2) | ES(p,p1,p2) | US(p,n)
  where
NS(p,m) = [x] p?x. [k,o]( {|NS(p,m)|} | x!m | o!o | o?o. kill(k) )
ES(p,p1,p2) = [y,n1,n2,z1,z2] p?y.
  ( p1!n1 | p2!n2 | n1?z1.(y!z1|ES(p,p1,p2)) + n2?z2.(y!z2|ES(p,p1,p2)) )
US(p,n) = p!n | [z] n?z.0
```

Fig. 1. COWS specification of a news/e-mail service.

Rule *ser_id* states that the behaviour of an identifier depends on the behaviour of its defining service after the substitution of actual parameters for formal parameters. The rule is engineered in such a way that the non-homonymy condition on bound entities is preserved by the unfoldings of the identifier. This is obtained by using decorated versions of transition label and of derived service in the conclusion of the *ser_id* rule. Function $\mathrm{l\_dec}(\alpha)$ decorates the bound name of α, if any. Function $\mathrm{s\_dec}(\alpha, \mathrm{s})$ returns a copy of s where all of the occurrences of both the bound names of s and of the bound name possibly occurring in α have been decorated. The decoration mechanism is an instance of a technique typically used in the implementation of the abstract machines for calculi with naming and α-conversion (see, e.g., [12, 15]). Here the idea is to enrich entities by superscripts consisting in finite strings of zeros, with d staying for the entity decorated by the empty string. Each time an entity is decorated, an extra zero is appended to the string. Entities decorated by distinct strings are different, and this ensures that the non-homonymy condition is dynamically preserved.

Fig. 1 displays the COWS specification of a simple service adapted from the CNN/BBC example in [10]. The global system, which will be used later on to carry on simple quantitative analysis, consists of two news services (`NS(p1,m1)` and `NS(p2,m2)`), the e-mail service `ES(p,p1,p2)`, and a user `US(p,n)`. The user invokes the e-mail service asking to receive a message with the latest news. On its side, `ES(p,p1,p2)` asks them to both `NS(p1,m1)` and `NS(p2,m2)` and sends back to the user the news it receives first. The sub-component `o!o|o?o.kill(k)` of the news service will be used to simulate (via a delay associated to the invoke and to the request over `o`) a time-out for replying to `ES(p,p1,p2)`.

3 Stochastic semantics

The stochastic extension of COWS is presented below. The syntax of the basic calculus is enriched in such a way that kill, invoke, and request actions are associated with a random variable with exponential distribution. Since exponential distribution is uniquely determined by a single parameter, called *rate*, the above mentioned atomic activities become pairs (μ,r), where μ represents the basic action, and $r \in \mathbb{R}^+$ is the rate of μ. In the enriched syntax, kill activities, invoke activities, and input-guarded services are written:

$$(\mathbf{kill}(k), \lambda) \qquad (u\,!\,w, \delta) \qquad (p\,?\,w, \gamma).\,s$$

$$\mathtt{req}(p;(\mathbf{kill}(k),\lambda)) = \mathtt{req}(p;(u\,!\,w,\delta)) = \mathtt{req}(p;\mathbf{0}) = 0$$

$$\mathtt{req}(p;(p'\,?\,w,\gamma).\,s') = \begin{cases} \gamma & \text{if } p = p' \\ 0 & \text{oth.} \end{cases} \qquad \mathtt{req}(p;s_1 \mid s_2) = \mathtt{req}(p;s_1) + \mathtt{req}(p;s_2)$$

$$\mathtt{req}(p;g_1+g_2) = \mathtt{req}(p;g_1) + \mathtt{req}(p;g_2) \qquad \mathtt{req}(p;\{\!|s|\!\}) = \mathtt{req}(p;s)$$

$$\mathtt{req}(p;[\,d\,]s) = \begin{cases} 0 & \text{if } p = d \text{ or } s\downarrow_d \\ \mathtt{req}(p;s) & \text{oth.} \end{cases}$$

$$\mathtt{req}(p;S(m_1,\ldots,m_j)) = \mathtt{req}(p;s\{^{m_1\cdots m_j}/_{n_1\ldots n_j}\}) \qquad \text{if } S(n_1,\ldots,n_j) = s$$

Table 2. Apparent rate of a request.

where the metavariables λ, δ and γ are used to range over kill, invoke and request rates, respectively. The intuitive meaning of $(\mathbf{kill}(k),\lambda)$ is that the activity $\mathbf{kill}(k)$ is completed after a delay Δt drawn from the exponential distribution with parameter λ. I.e., the elapsed time Δt models the use of resources needed to complete $\mathbf{kill}(k)$. The meaning of both $(u\,!\,w,\delta)$ and $(p\,?\,w,\gamma)$ is analogous.

Whenever more than one activity is enabled, the dynamic evolution of a service is driven by a *race condition*: All the enabled activities try to proceed, but only the fastest one succeeds. Race conditions ground the replacement of the non-deterministic choice of COWS by a *probabilistic choice*. The probability of a computational step $s \xrightarrow{\alpha} s'$ is the ratio between its rate and the *exit rate* of s which is defined as the sum of the rates of all the activities enabled in s. For instance, service $S = [\,x\,][\,y\,]((p\,?\,x,\gamma_1).\,s_1 + (p\,?\,y,\gamma_2).\,s_2)$ has exit rate $\gamma_1+\gamma_2$ and the probability that the activity $p\,?\,x$ is completed is $\gamma_1/(\gamma_1+\gamma_2)$.

The exit rate of a service is computed on the basis of the so-called *communication rate*, which is turn is defined in terms of the *apparent rate* of request and invoke activities [13, 7]. The apparent rate of a request over the endpoint p in a service s, written $\mathtt{req}(p;s)$, is the sum of the rates of all the requests over the endpoint p which are enabled in s. Function $\mathtt{req}(p;s)$ is defined in Tab. 2 by induction on the structure of s. It just sums up the rates of all the requests that can be executed in s at endpoint p. As an example, we show in the following the computation of the apparent rate of a request over p for the above service S.

$$\begin{aligned}\mathtt{req}(p;S) &= \mathtt{req}(p;(p\,?\,x,\gamma_1).\,s_1 + (p\,?\,y,\gamma_2).\,s_2)\\ &= \mathtt{req}(p;(p\,?\,x,\gamma_1).\,s_1) + \mathtt{req}(p;(p\,?\,y,\gamma_2).\,s_2) = \gamma_1+\gamma_2\end{aligned}$$

The apparent rate of an invoke over p in a service s, written $\mathtt{inv}(p;s)$, is defined analogously to $\mathtt{req}(p;s)$. It computes the sum of the rates of all the invoke activities at p which are enabled in s. Its formal definition is omitted for the sake of space. The apparent communication rate of a synchronization at endpoint p in service s is taken to be the slower value between $\mathtt{req}(p;s)$ and $\mathtt{inv}(p;s)$, i.e. $\mathtt{min}(\mathtt{req}(p;s),\mathtt{inv}(p;s))$.

All the requests over a certain endpoint p in s compete to take a communication over p. Therefore, given that a request at p is enabled in s, the probability that a request $(p\,?\,x,\gamma)$ completes, is $\gamma/\mathtt{req}(p;s)$. Likewise, when an invoke at p is enabled in s, the probability that the invoke $(p\,!\,n,\delta)$ completes is $\delta/\mathtt{inv}(p;s)$. Hence, if a communication at p occurs in s, the probability that $(p\,?\,x,\gamma)$ and $(p\,!\,n,\delta)$ are involved is $\gamma/\mathtt{req}(p;s) \times \delta/\mathtt{inv}(p;s)$.

$$\sharp(\alpha;s) = \begin{cases} \mathtt{req}(p;s) & \text{if } \alpha = p\,?\,w, p\,?\,(x) \\ \mathtt{inv}(p;s) & \text{if } \alpha = p\,!\,n, p\,!\,(n) \\ [\mathtt{req}(p;s), \mathtt{inv}(p;s)] & \text{if } \alpha = p \cdot \sigma \cdot \sigma' \\ 0 & \text{oth.} \end{cases}$$

Table 3. Apparent rate of α in service s.

The rate of the communication between $(p\,?\,x, \gamma)$ and $(p\,!\,n, \delta)$ in s is given by the following formula:

$$\frac{\gamma}{\mathtt{req}(p;s)} \frac{\delta}{\mathtt{inv}(p;s)} \mathtt{min}(\mathtt{req}(p;s), \mathtt{inv}(p;s)) \tag{1}$$

namely, it is given by the product of the apparent rate of the communication and of the probability, given that a communication at p occurs in s, that this is just a communication between $(p\,?\,x, \gamma)$ and $(p\,!\,n, \delta)$.

The stochastic semantics of COWS uses *enhanced labels* in the style of [4]. An enhanced label θ is a triple (α, ρ, ρ') prefixed by a *choice-address* ϑ. The α component of the triple is a label of the transition system in Tab. 1. The two components ρ and ρ' can both be either a rate (λ, γ, or δ) or a two dimensional vector of request-invoke rates $[\gamma, \delta]$. We will comment later on upon the usefulness of the choice-address component ϑ.

The enhanced label $\vartheta(\alpha, \rho, \rho')$ records in ρ the rate of the fired action. Axioms *kill*, *req*, and *inv* become respectively:

$$(\mathbf{kill}(k), \lambda) \xrightarrow{(\dagger k, \lambda, \lambda)} \mathbf{0} \qquad (p\,?\,w, \gamma)\,.\,s \xrightarrow{(p\,?\,w, \gamma, \gamma)} s \qquad (p\,!\,n, \delta) \xrightarrow{(p\,!\,n, \delta, \delta)} \mathbf{0}\,.$$

The apparent rate of an activity labelled by α is computed inductively and saved in the ρ' component of the enhanced label $\vartheta(\alpha, \rho, \rho')$. Accordingly, rule *par_pass* takes the shape shown below.

$$\frac{s_1 \xrightarrow{\vartheta(\alpha,\rho,\rho')} s_1' \qquad \alpha \neq p \cdot \sigma \cdot \sigma' \qquad \alpha \neq \dagger k}{s_1 \mid s_2 \xrightarrow{\vartheta(\alpha,\rho,\rho' + \sharp(\alpha;s_2))} s_1' \mid s_2} \ (\mathit{par\_pass})$$

Function $\sharp(\alpha;s)$, defined in Tab. 3, computes the apparent rate of the activity α in the service s. If α is a request (an invoke) at endpoint p, then function $\sharp(\alpha;s)$ returns $\mathtt{req}(p;s)$ ($\mathtt{inv}(p;s)$). In case of a communication at p, function $\sharp(\alpha;s)$ returns the vector $[\mathtt{req}(p;s), \mathtt{inv}(p;s)]$ of the request apparent rate and of the invoke apparent rate. Rules *par_conf* and *par_kill* are modified in a similar way.

An example of application of the *par_pass* rule follows.

$$\frac{(p\,!\,n, \delta_1) \xrightarrow{(p\,!\,n, \delta_1, \delta_1)} \mathbf{0} \ (\mathit{inv})}{(p\,!\,n, \delta_1) \mid (p\,!\,m, \delta_2) \xrightarrow{(p\,!\,n, \delta_1, \delta_1 + \sharp(p\,!\,n; (p\,!\,m, \delta_2)))} \mathbf{0} \mid (p\,!\,m, \delta_2)} \ (\mathit{par\_pass})$$

The enhanced label $(p\,!\,n, \delta_1, \delta_1 + \sharp(p\,!\,n; (p\,!\,m, \delta_2)))$ records that the activity $p\,!\,n$ is taking place with rate δ_1, and with apparent rate $\delta_1 + \sharp(p\,!\,n; (p\,!\,m, \delta_2)) = \delta_1 + \delta_2$.

To compute the rate of a communication between the request $(p\,?\,n, \gamma)$ in s_1 and the invoke $(p\,!\,n, \delta)$ in s_2 with apparent rates γ'' and δ'', respectively, the enhanced label keeps track of both the rates γ and δ, and of both the apparent rates $\gamma'' + \sharp(p\,?\,n; s_2)$ and $\delta'' + \sharp(p\,!\,n; s_1)$. Rule *com_n* is modified as follows.

$$\frac{s_1 \xrightarrow{\vartheta(p\,?\,n,\gamma,\gamma'')} s_1' \qquad s_2 \xrightarrow{\vartheta'(p\,!\,n,\delta,\delta'')} s_2'}{s_1 \mid s_2 \xrightarrow{(\vartheta,\vartheta')(p\cdot\varepsilon\cdot\varepsilon,[\gamma,\delta],[\gamma''+\sharp(p\,?\,n;s_2),\delta''+\sharp(p\,!\,n;s_1)])} s_1' \mid s_2'} \;(\mathit{com\_n})$$

Notice that the enhanced label in the conclusion of rule *com_n* contains all the data needed to compute the relative communication rate which, following Eq. (1), is given by $(\gamma/\gamma')(\delta/\delta')\,\mathtt{min}(\gamma', \delta')$ where $\gamma' = \gamma'' + \sharp(p\,?\,n; s_2)$ and $\delta' = \delta'' + \sharp(p\,!\,n; s_1)$. From the point of view of stochastic information, rules *com_x*, *cl_nx*, *cl_x*, and *cl_n* behave the same as rule *com_n*. Indeed their stochastic versions are similar to the one of *com_n*.

Rules *prot*, *del_sub*, *del_kill*, *del_pass*, *ser_id*, *op_req*, *op_inv*, and *del_cl* are transparent w.r.t. stochastic information, i.e., their conclusion does not change the values ρ and ρ' occurring in the premise $s \xrightarrow{\vartheta(\alpha,\rho,\rho')} s'$. We report here only the stochastic version of *del_kill*, the other rules are changed in an analogous way.

$$\frac{s \xrightarrow{\vartheta(\dagger k,\rho,\rho')} s'}{[\,k\,]s \xrightarrow{\vartheta(\dagger,\rho,\rho')} [\,k\,]s'} \;(\mathit{del\_kill})$$

Rule *choice* deserves special care. Consider the service $(p\,!\,n, \delta) \mid (p\,?\,n, \gamma).\,\mathbf{0} + (p\,?\,n, \gamma).\,\mathbf{0}$, and suppose that enhanced labels would not comprise a choice-address component. Then the above service could perform two communications at p, both with the same label $(p\cdot\varepsilon\cdot\varepsilon, [\gamma, \delta], [\gamma+\gamma, \delta])$ and with the same residual service $\mathbf{0} \mid \mathbf{0}$. If the semantic setting is not able to discriminate between these two transitions, then the exit rate of the service cannot be consistently computed. This calls for having a way to distinguish between the choice of either the left or the right branch of a choice service. Indeed, the stochastic rules for choice become the following ones.

$$\frac{g_1 \xrightarrow{\vartheta(\alpha,\rho,\rho')} s}{g_1 + g_2 \xrightarrow{+_0\vartheta(\alpha,\rho,\rho'+\sharp(\alpha;g_2))} s} \;(\mathit{choice}_0) \qquad \frac{g_2 \xrightarrow{\vartheta(\alpha,\rho,\rho')} s}{g_1 + g_2 \xrightarrow{+_1\vartheta(\alpha,\rho,\rho'+\sharp(\alpha;g_1))} s} \;(\mathit{choice}_1)$$

By these rules, the above service $(p\,!\,n, \delta) \mid (p\,?\,n, \gamma).\,\mathbf{0} + (p\,?\,n, \gamma).\,\mathbf{0}$ executes two transitions leading to the same residual process but labelled by $+_0(p, [\gamma, \delta], [\gamma + \gamma, \delta])$ and by $+_1(p, [\gamma, \delta], [\gamma + \gamma, \delta])$, respectively.

We conclude the presentation of the stochastic semantics of COWS by providing the definition of *stochastic execution step* of a closed service:

$$s \xrightarrow{\vartheta(\alpha,\rho,\rho')} s' \text{ with either } \alpha = \dagger \text{ or } \alpha = p \cdot \varepsilon \cdot \sigma'.$$

4 Stochastic analysis

The definition of stochastic execution step has two main properties: (i) it can be computed automatically by applying the rules of the operational semantics; (ii) it is completely abstract, i.e., enhanced labels only collect information about rates and apparent rates. For instance, it would be possible to compute the communication rate using a formula different from Eq. (1). This makes the modelling phase independent from the analysis phase, and also allows the application of different analysis techniques to the same model.

In what follows, we show how to apply Continuous Time Markov Chain (CTMC) based analysis to COWS terms. A CTMC is a triple $\mathcal{C} = (Q, \overline{q}, \mathbf{R})$ where Q is a finite set of *states*, $\overline{q}$ is the *initial state*, $\mathbf{R} : Q \times Q \to \mathbb{R}^+$ is the *transition matrix.* We write $\mathbf{R}(q_1, q_2) = r$ to mean that q_1 evolves to q_2 with rate r. Various tools are available to analyze CTMCs. Among them there are probabilistic model checkers: Tools that allow the formal verification of stochastic systems against quantitative properties.

A service s' is a *derivative* of service s if s' can be reached from s by a finite number of stochastic evolution steps. The *derivative set* of a service s, $\mathtt{ds}(s)$, is the set including s and all of its derivatives. A service s is *finite* if $\mathtt{ds}(s)$ is finite. Given a finite service s, the associated CTMC is $\mathcal{C}(s) = (\mathtt{ds}(s), s, \mathbf{R})$, where $\mathbf{R}(s, s') = \sum_{s \xrightarrow{\theta} s'} \mathtt{rate}(\theta)$. Here the rate of label θ, $\mathtt{rate}(\theta)$, is computed accordingly to Eq. (1):

$$\mathtt{rate}(\theta) = \begin{cases} (\gamma/\gamma')(\delta/\delta')\mathtt{min}(\gamma', \delta') & \text{if} \quad \theta = \vartheta(p, [\gamma, \delta], [\gamma', \delta']) \\ \rho & \text{if} \quad \theta = \vartheta(\dagger, \rho, \rho') \end{cases}$$

After the above definition, we can analyse COWS services exploiting available tools on CTMCs. As a very simple example, we show how the news/e-mail service in Fig. 1 can be verified using PRISM [14], a probabilistic model checking tool that offers direct support for CTMCs and can check properties described in Continuous Stochastic Logic [1]. A short selection of example properties that can be verified against the news/e-mail service follows.

– `P` $\geq$ 0.9`[ true U`$\geq$ 60`(NS1 | NS2)]`: “With probability greater than 0.9 either `NS(p1,m1)` or `NS(p2,m2)` are activated in at most 60 units of time”;
– `P` ≥ 1 `[ true U (m1|m2)]`: “The user `US(p,n)` receives either the message `m1` or `m2` with probability 1”;
– `P=?[trueU[T,T](m1|m2)]`: “Which is the probability that the user `US(p,n)` receives either the message `m1` or `m2` within time `T`?” Fig. 2 shows a plot generated by PRISM when checking the news/e-mail service against this property.

5 Concluding remarks

We presented a stochastic extension of COWS, a formal calculus strongly inspired by WS-BPEL, and showed how the obtained semantic model can be used as input to carry on probabilistic verification using PRISM.

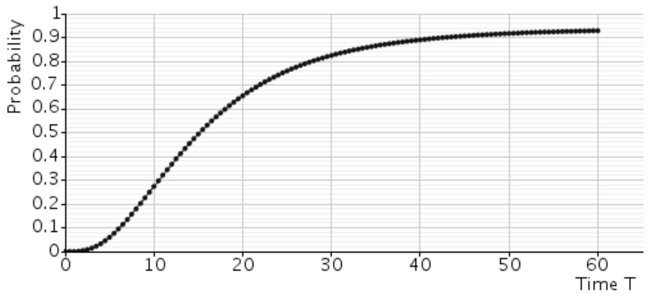

Fig. 2. Probability that `US(p,n)` receives either the message `m1` or `m2` within time `T`.

The technical approach presented in this paper aims at producing an integrated set of tools to quantitatively model, simulate and analyse web service descriptions.

Acknowledgements We thank Rosario Pugliese, Francesco Tiezzi, and an anonymous referee for their useful comments and suggestions on a draft of this work.

References

1. A. Aziz, K. Sanwal, V. Singhal, and R.K. Brayton. Model-checking continuous-time markov chains. *ACM TOCL*, 1(1):162–170, 2000.
2. R. Bruni, H.C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05*, pages 209–220, 2005.
3. R. Bruni, H.C. Melgratti, and E. Tuosto. Translating Orc Features into Petri Nets and the Join Calculus. In *Proc. WS-FM '06*, vol. 4184 of *LNCS*, pages 123–137. Springer, 2006.
4. P. Degano and C. Priami. Enhanced operational semantics. *ACM CS*, 33(2):135–176, 2001.
5. S.T. Gilmore and M. Tribastone. Evaluating the scalability of a web service-based distributed e-learning and course management system. In *Proc. WS-FM '06*, vol. 4184 of *LNCS*, pages 214–226. Springer, 2006.
6. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. A Calculus for Service Oriented Computing. In *Proc. ICSOC'06*, vol. 4294 of *LNCS*. Springer, 2006.
7. J. Hillston. *A Compositional Approach to Performance Modelling.* CUP, 1996.
8. A. Lapadula, R. Pugliese, and F. Tiezzi. Calculus for Orchestration of Web Services. In *Proc. ESOP'07*, vol. 4421 of *LNCS*, pages 33–47, 2007. Full version available at `http://rap.dsi.unifi.it/cows/`.
9. R. Milner. *Communicating and mobile systems: the π-calculus.* CUP, 1999.
10. J. Misra and W.R. Cook. Computation Orchestration: A Basis for Wide-area Computing. *SoSyM*, 6(1):83–110, 2007.
11. PEPA. `http://www.dcs.ed.ac.uk/pepa/`, 2007.
12. F. Pottier. An Overview of Cαml. *ENTCS*, 148(2):27–52, 2006.
13. C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(7):578–589, 1995.
14. PRISM. `http://www.cs.bham.ac.uk/~dxp/prism/`, 2007.
15. P. Quaglia. Explicit substitutions for pi-congruences. *TCS*, 269(1-2):83–134, 2001.
16. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* CUP, 2001.