A Symbolic Semantics for a Calculus for Service-Oriented Computing [1]

Rosario Pugliese[2] Francesco Tiezzi[3]

Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze

Nobuko Yoshida[4]

Department of Computing, Imperial College London

Abstract

We introduce a symbolic characterisation of the operational semantics of COWS, a formal language for specifying and combining service-oriented applications, while modelling their dynamic behaviour. This alternative semantics avoids infinite representations of COWS terms due to the value-passing nature of communication in COWS and is more amenable for automatic manipulation by analytical tools, such as e.g. equivalence and model checkers. We illustrate our approach through a 'translation service' scenario.

Keywords: Service-oriented computing, service orchestration, process calculi, symbolic semantics.

1 Introduction

In recent years, the increasing success of e-business, e-learning, e-government, and other similar emerging models, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for *service-oriented computing* (SOC) supporting automated use. SOC advocates the use of loosely coupled 'services', to be understood as autonomous, platform-independent, computational entities that can be described, published, discovered, and assembled, as the basic blocks for building interoperable and evolvable systems and applications. While early examples of technologies that are at least partly service-oriented date back to CORBA, DCOM, J2EE and IBM WebSphere, the most successful instantiation of the SOC paradigm are probably the more recent *web services*. These are sets of operations that can be published, located and invoked through the Web

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

[2] Email: pugliese@dsi.unifi.it

[3] Email: tiezzi@dsi.unifi.it

[4] Email: yoshida@doc.ic.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

via XML messages complying with given standard formats. To support the web service approach, several new languages and technologies have been designed and many international companies have invested a lot of efforts.

Current software engineering technologies for SOC, however, remain at the descriptive level and lack rigorous formal foundations. We are still experiencing a gap between practice (programming) and theory (formal methods and analysis techniques) in the design of SOC applications. The challenges come from the necessity of dealing at once with such issues as asynchronous interactions, concurrent activities, workflow coordination, business transactions, failures, resource usage, and security, in a setting where demands and guarantees can be very different for the many different components. Many researchers have hence put forward the idea of using *process calculi*, a cornerstone of current foundational research on specification and analysis of concurrent, distributed and mobile systems through mathematical — mainly algebraic and logical — tools. Indeed, due to their algebraic nature, process calculi convey in a distilled form the compositional programming style of SOC. Thus, many process calculi have been designed (e.g. [8,7,15,12,9,14,3,5,26]), addressing one aspect or another of SOC and aiming at assessing the adequacy of diverse sets of primitives w.r.t. modelling, combining and analysing service-oriented applications.

By taking inspiration from well-known process calculi and from the standard language for orchestration of web services WS-BPEL [21], in [16] we have designed COWS (*Calculus for Orchestration of Web Services*), a process calculus for specifying and combining service-oriented applications, while modelling their dynamic behaviour. We have shown that COWS can model and handle distinctive features of (web) services, such as, e.g., correlation-based communication, compensation activities, service instances and interactions among them, race conditions among service instances and service definitions.

A major benefit of using process calculi is that they enjoy a rich repertoire of elegant meta-theories, proof techniques and analytical tools that can be likely tailored to the needs of SOC. Concerning this, in [11] we have developed a logic and a model checker to express and check functional properties of services specified in COWS, while in [23] we have studied observational semantic theories for COWS. However, such tools suffer from a lack of compositionality and efficiency. Indeed, generally speaking, model and equivalence checkers, and other similar verification tools, do not work directly on syntactic specifications but rather on abstract representations of the behaviour of processes. Thus, for value-passing languages, such as COWS, using an inappropriate representation can lead to unfeasible verifications. Indeed, according to the COWS's original operational semantics, if the communicable values range over an infinite value set (e.g. natural numbers and strings), the behaviour of a service that performs a receive activity is modelled by an infinite abstract representation. Such representation is a Labelled Transition System whose initial state has infinite outgoing edges, each labelled with an input label having a different value as argument and leading to a different state.

Hence, by taking inspiration from Hennessy and Lin [13], in this paper we define a *symbolic* operational semantics for COWS. Differently from the symbolic semantics for more standard calculi, such as value-passing CCS or π-calculus, ours deals at once with, besides receive transitions, a number of complex features, such as, e.g., generation and exportation of fresh names, pattern-matching, expressions evaluation, and priorities among conflicting receives. The new semantics avoids infinite representations of COWS terms due to the value-passing nature of communication in COWS and associates a finite representation to

each finite COWS term. It is then more amenable for automatic manipulation by analytical tools, such as e.g. equivalence and model checkers. Our major result is a theorem of 'operational correspondence'. We prove that, under appropriate conditions, any transition of the original semantics can be generated using the symbolic one, and vice versa. In general, however, additional transitions can be derived using the symbolic semantics since it also accounts for services ability to interact with the environment.

The rest of the paper is organised as follows. Section 2 provides some motivations for the symbolic semantics of COWS; this is done by means of an 'Italian-English translation service' scenario that is used also to informally describe in a step-by-step fashion the main features of COWS. Section 3 presents the original syntax and operational semantics of COWS. Section 4 introduces the symbolic variant of the operational semantics of COWS and our major results, together with some clarifying examples. Section 5 shows two extensions of the symbolic semantics. Finally, Section 6 touches upon comparisons with related work and directions for future work.

2 A 'translation service' scenario

In this section, we present COWS main features and syntax in a step-by-step fashion while modelling an Italian-English translation service. By means of this scenario, we discuss some verification problems and present the major intuitions underlying the symbolic operational semantics for COWS. For the time being, we use a *monadic* variant of COWS, i.e. we assume that invoke and receive activities can carry one single parameter at a time. In fact, for the sake of presentation, the symbolic semantics is introduced for the monadic variant in Section 4, and is then extended to polyadic communication in Section 5.2.

Let us consider a service that provides to its customers an Italian-English translation service. Specifically, when the service is invoked by a customer, that communicates first her partner name and then an Italian word, it replies to the request with either the corresponding English word or the string "*unknown word*". A high-level specification of the service can be rendered in COWS as follows:

$$[x]\, t \bullet req?x \,.\, [y]\, t \bullet word?y \,.\, x \bullet resp!\mathsf{trans}(y) \qquad (1)$$

where t is the translation service partner name, *req*, *word* and *resp* are operation names, x and y are variables that store the customer partner name and the Italian word to be translated respectively, and $\mathsf{trans}(\_)$ is a total function that maps a large subset of Italian words to the corresponding English ones and returns the string "*unknown word*" for all words that do not appear in the Italian words set. The service simply performs a sequence of two *receive* activities $t \bullet req?x$ and $t \bullet word?y$, corresponding to reception of a request and of an Italian word sent by a customer, and replies with the translated word, by invoking the operation *resp* of the customer by means of the *invoke* activity $x \bullet resp!\mathsf{trans}(y)$. Receives and invokes are the basic communication activities provided by COWS. Besides input parameters and sent values, they indicate the endpoint, i.e. a pair $p \bullet o$ made of a partner name p and an operation name o, through which the communication should occur. Differently from most process calculi, receive activities in COWS bind neither names nor variables. The only binding construct is *delimitation*: $[d]\, s$ binds the delimited element d in the scope s (the notions of bound and free occurrences of a delimited element are defined accordingly). For example, the service (1) uses the delimitation operator to declare the scope of variables x

and y. An inter-service communication takes place when the arguments of a receive and of a concurrent invoke along the same endpoint do match, and causes replacement of the variables arguments of the receive with the corresponding values arguments of the invoke (within the scope of variables declarations). For example, variable x will be initialised by the first receive activity with data provided by a customer.

At a lower level, the service could be described in terms of three entities composed by using the *parallel composition* operator $\_ \mid \_$ that allows them to be concurrently executed and to interact with each other. A low-level COWS specification of the translation service can be

$$[reqDB1, reqDB2, respDB1, respDB2]\,(\,Translator \mid DB1 \mid DB2\,) \qquad (2)$$

The delimitation operator is used here to declare that *reqDB1*, *reqDB2*, *respDB1* and *respDB2* are private operation names known to the three components *Translator*, *DB1* and *DB2*, and only to them (at least initially, since during a computation private names can be exported exactly as in π-calculus). The three subservices are defined as follows:

$$\begin{aligned}
Translator \triangleq\ & [x]\,t \bullet req?x\,.\,[y]\,t \bullet word?y\,. \\
& [k]\,(\,t \bullet reqDB1!y \mid [x_1]\,t \bullet respDB1?x_1\,.\,(\,\mathbf{kill}(k) \mid \{\!|x \bullet resp!x_1|\!\}\,) \\
& \quad \mid t \bullet reqDB2!y \mid [x_2]\,t \bullet respDB2?x_2\,.\,(\,\mathbf{kill}(k) \mid \{\!|x \bullet resp!x_2|\!\}\,)\,)
\end{aligned}$$

$$\begin{aligned}
DB1 \triangleq\ & t \bullet reqDB1?\text{“}a\text{”}.\,t \bullet respDB1!\text{“}to\text{”} \\
& + t \bullet reqDB1?\text{“}albero\text{”}.\,t \bullet respDB1!\text{“}tree\text{”} \\
& + \ldots + t \bullet reqDB1?\text{“}zucca\text{”}.\,t \bullet respDB1!\text{“}pumpkin\text{”}
\end{aligned}$$

$$\begin{aligned}
DB2 \triangleq\ & [z]\,(\,t \bullet reqDB2?z.\,t \bullet respDB2!\text{“}unknown\ word\text{”} \\
& \quad + t \bullet reqDB2?\text{“}a\text{”}.\,t \bullet respDB2!\text{“}to\text{”} \\
& \quad + t \bullet reqDB2?\text{“}abate\text{”}.\,t \bullet respDB2!\text{“}abbot\text{”} \\
& \quad + \ldots + t \bullet reqDB2?\text{“}zuppo\text{”}.\,t \bullet respDB2!\text{“}soaked\text{”}\,)
\end{aligned}$$

Service *Translator* is publicly invocable and can interact with customers other than with the ‘internal’ services *DB1* and *DB2*. These latter two services, instead, can only be invoked by *Translator* (indeed, all the operations used by them are restricted) and have the task of looking up in databases the English word corresponding to a given Italian one and replying accordingly. In particular, *DB1* performs a quick search in a small database of commonly used words, while *DB2* performs a slower search in a bigger database (that exactly corresponds to that modelled by the function $\mathsf{trans}(\_)$). After the two initial receives, for e.g. performance or fault tolerance purposes, *Translator* invokes services *DB1* and *DB2* concurrently. When one of them replies, *Translator* immediately stops the other search. This is done by executing the *kill* activity $\mathbf{kill}(k)$, that forces termination of all unprotected parallel terms inside the enclosing $[k]$, that stops the killing effect. Then, *Translator* forwards the response to the customer and terminates. Kill activities are executed eagerly with respect to the other parallel activities but critical code can be protected from the effect of a forced termination by using the *protection* operator $\{\!|\_|\!\}$; this is indeed the case of the response $x \bullet resp!x_1$ in our example. Services *DB1* and *DB2* use the *choice* operator $\_ + \_$ to offer alternative behaviours: one of them can be selected by executing an invoke matching the receive leading the behaviour. In case the word to be translated is unknown, *DB1* does not reply, while *DB2* returns the string “*unknown word*”. Indeed, the semantics of parallel

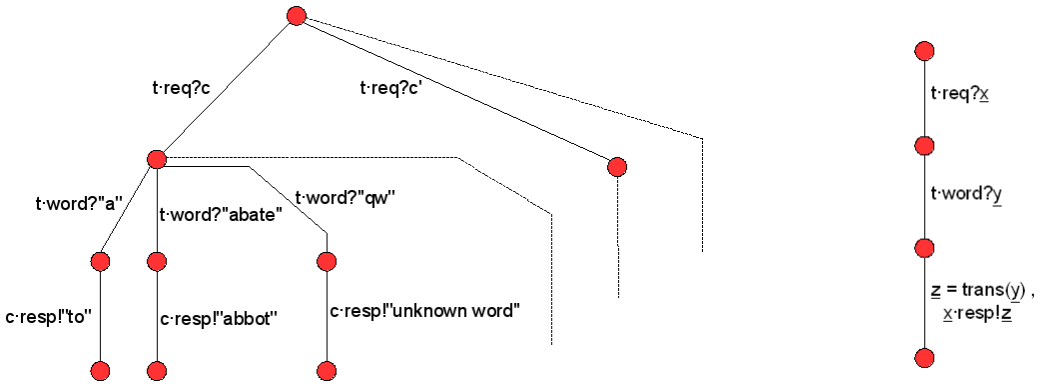

Fig. 1. LTS and symbolic LTS for the translation service (high-level specification)

composition avoids that *DB2* returns "*unknown word*" in case of known words. This is done by assigning the receive $t \bullet reqDB2?z$ less priority than the other receive activities, so that it is only executed when none of the other receives matches the word to be translated (see Section 3 for further details about the prioritised semantics of COWS).

Equivalence and model checkers, and other similar verification tools, do not work directly on syntactic specifications such as those above, but rather on more abstract representations of the behaviour of processes. Thus, using an inappropriate representation can lead to unfeasible verifications. In the rest of the section, we discuss verification problems and how to cope with them by exploiting a symbolic approach.

Verification problems. When the considered specification language is a value-passing process algebra and the value-space is infinite, using standard Labelled Transition Systems (LTSs) for the semantics can lead to infinite representations. For example, the operational behaviour of service (1) can be represented by the infinite LTS in the left-hand side of Figure 1, where nodes denote states and edges denote transitions between states implicitly oriented from top to bottom. Notably, for the sake of presentation, the LTSs shown in the figures rely on an operational semantics in *early* style, where substitutions are applied when receive actions are inferred. However, the problem of infinite representations remains also in case of *late* semantics, due to the fact that the continuation of a receive action with argument a variable x has to be considered under all possible substitutions for x.

The symbolic approach. To tackle the problems above, in [13] Hennessy and Lin have introduced the so-called *symbolic LTSs* and used them to define finite semantical representations of terms of the value-passing CCS. For example, the symbolic LTSs corresponding to the COWS service (1) is shown in the right-hand side of Figure 1. The symbolic actions $t \bullet req?\underline{x}$ and $t \bullet word?\underline{y}$ denote reception of unknown values $\underline{x}$ and $\underline{y}$ along endpoints $t \bullet req$ and $t \bullet word$, respectively; the condition-guarded symbolic action $(\underline{z} = \mathsf{trans}(\underline{y})\,,\ \underline{x} \bullet resp!\underline{z})$ denotes sending of an unknown value $\underline{z}$ such that $\underline{z} = \mathsf{trans}(\underline{y})$. Of course, for the same reasons, also the LTS representing the behaviour of service (2) is infinite, while the corresponding symbolic LTS is finite. Indeed, if for the sake of presentation we assume that database *DB1* contains only the association for word "*a*" and database *DB2* contains only the associations for "*a*" and "*abate*", the symbolic LTS representing (2) is that shown in Figure 2.

***Applying the symbolic approach to* COWS.** The main contribution of this work is the

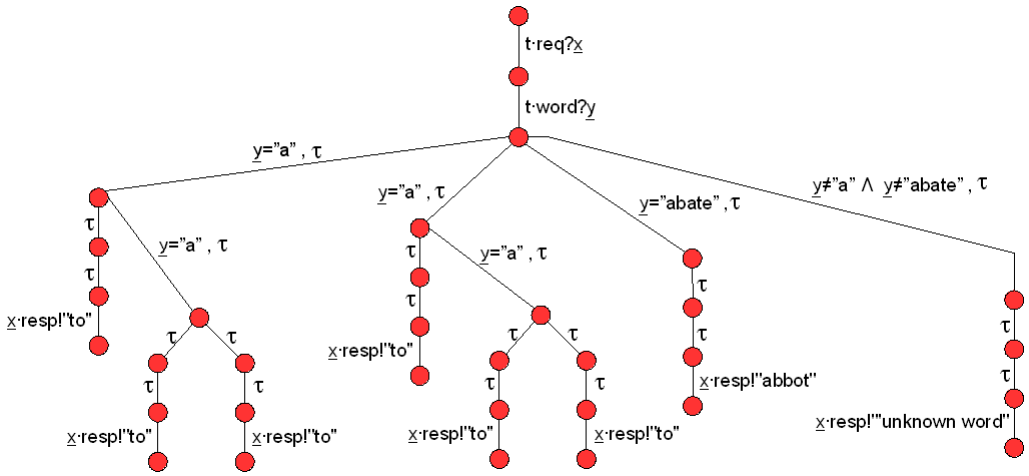

Fig. 2. Symbolic LTS for the (simplified) translation service (low-level specification)

development of a symbolic operational semantics for COWS. To achieve this goal, the main issue is to give receive activities a proper semantics, because variables in their arguments are placeholders for something to be received. For example, let us consider the service $p \bullet o?x.s$. If $p \bullet o?x.s \xrightarrow{p \bullet o?x} s$ then the behaviour of the continuation service s must be considered under all substitutions of the form $\{x \mapsto v\}$ (i.e. the semantics of s can intuitively be thought of as a function $\lambda x.\, s$ from values to services). In case of the standard semantics for π-calculus [20], for example, this problem is not tackled at the operational semantics level, but it is postponed to the observational semantics level. In fact, in the definition of late bisimulation for π-calculus, whenever P is bisimilar to Q, if $P \xrightarrow{a(x)} P'$ then there is Q' such that $Q \xrightarrow{a(x)} Q'$ and $P'\{u/x\}$ is bisimilar to $Q'\{u/x\}$ for every u. Thus, continuations P' and Q' are considered under all substitutions for x. Instead, here we aim at defining an operational semantics for COWS that properly handles input transitions, and allow finite state LTSs to be associated to finite COWS terms.

The basic idea is to allow receive activities to evolve by performing a communication with the 'external world' (i.e. a COWS context), this way they do not need to synchronise with invoke activities within the considered term. To avoid infinite branching (as in the case of early operational semantics), we replace variables with *unknown values* rather than with specific values. We denote by $\underline{x}$ the unknown value for the variable x. This way, the term $[x]\,(\,p \bullet o?x.\, q \bullet o'!x\,)$ can evolve as follows:

$$[x]\,(\,p \bullet o?x.\, q \bullet o'!x\,) \xrightarrow{p \bullet o\,?[x]} q \bullet o'!\underline{x} \xrightarrow{q \bullet o'!\underline{x}} \mathbf{0}$$

Also receive activities having a value as argument (e.g. $p \bullet o?v$) and invoke activities (e.g. $p \bullet o!v$) can evolve by communicating with the external world. Of course, these kinds of communication do not produce substitutions.

When an external communication takes place, the behaviour of the continuation service depends on the *admittable values* for the unknown value. To take care of the real values that the unknown values can assume, we define a *symbolic semantics* for COWS, where the label on each transition has two components: the *condition* that must hold for the transition to be enabled and, as usual, the *action* of the transition. Moreover, to store the conditions that must hold to reach a state and the names exported along the path, we define the semantics

over configurations of the form $\Phi, \Delta \vdash s$, called *constrained services*, where the condition Φ and the set of names Δ are used to determine the actions that s can perform. Thus, the symbolic transitions are of the form $\Phi, \Delta \vdash s_1 \overset{\Phi', \alpha}{\succ\!\!\longrightarrow} \Phi', \Delta' \vdash s_2$, meaning "if the condition Φ' (such that Φ is a subterm of Φ') holds then s_1 can perform the action α leading to s_2 by extending the set of exported private names Δ to the set Δ'".

The symbolic LTS associated to a COWS term conveys in a distilled form all the semantics information on the behaviour of terms. More specifically, besides receive transitions, symbolic representations take into account generation and exportation of fresh names, pattern-matching, expressions evaluation, and priorities among conflicting receives. Dealing at once with all the above features at operational semantics level makes the development of a symbolic semantics for COWS more complex than for more standard calculi, such as value-passing CCS or π-calculus.

3 COWS and its standard operational semantics

COWS (Calculus for Orchestration of Web Services, [16]) is a recently designed process calculus for specifying, combining and analyzing service-oriented applications, while modelling their dynamic behaviour. 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. In this section, we present the standard syntax and operational semantics of COWS. We refer the interested reader to [16] for many examples illustrating COWS peculiarities and expressiveness, and for comparisons with other process-based and orchestration formalisms.

The syntax of COWS is presented in Table 1. It 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 $v, v', \ldots$) and the set of 'write once' *variables* (ranged over by $x, y, \ldots$). The set of values is left unspecified; however, we assume that it includes the set of *names*, ranged over by $n, m, o, p, \ldots$, 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, but do not include killer labels (that, hence, are *non-communicable* values). Partner names and operation names can be combined to designate *communication endpoints*, written $p \bullet o$, and can be communicated, but dynamically received names can only be used for service invocation (as in the Lπ [19]). Indeed, communication endpoints of receive activities are identified statically because their syntax only allows using names and not variables.

We use w to range over values and variables, u to range over names and variables, and d to range over killer labels, names and variables. Notation $\overline{\cdot}$ stands for tuples of objects, e.g. $\overline{x}$ 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. We adopt the following conventions about the operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice. In the sequel, we shall use $\mathfrak{n}$ to range over communication endpoints that do not contain variables (e.g. $p \bullet o$), and $\mathfrak{u}$ to range over communication endpoints that may contain variables (e.g. $u \bullet u'$). We will omit trailing occurrences of $\mathbf{0}$, writing e.g. $p \bullet o?\overline{w}$ instead of $p \bullet o?\overline{w}.\mathbf{0}$, and write $[d_1, \ldots, d_n]\, s$ in place of $[d_1] \ldots [d_n]\, s$. We will write

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

Table 1
COWS syntax

$I \triangleq s$ to assign a name I to the term s.

The only *binding* construct is delimitation: $[d]\, s$ binds d in the scope s. In fact, to enable concurrent threads within each service instance to share (part of) the state, receive activities in COWS bind neither names nor variables, which is different from most process calculi. Instead, the range of application of the substitutions generated by a communication is regulated by the delimitation operator, that additionally permits to generate fresh names (as the restriction operator of the π-calculus) and to delimit the field of action of kill activities. Thus, the occurrence of a name/variable/label is *free* if it is not under the scope of a delimitation for it. We denote by $\mathrm{fk}(t)$ the set of killer labels that occur free in t, and by $\mathrm{fd}(t)$ that of free names/variables/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.

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

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).

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

$$
\begin{array}{lll}
*\,\mathbf{0} \equiv \mathbf{0} & *\,s \equiv s \mid *\,s & \{\!|\mathbf{0}|\!\} \equiv \mathbf{0} \\
\{\!|\{\!|s|\!\}|\!\} \equiv \{\!|s|\!\} & \{\!|[d]\,s|\!\} \equiv [d]\,\{\!|s|\!\} & [d]\,\mathbf{0} \equiv \mathbf{0} \\
[d_1]\,[d_2]\,s \equiv [d_2]\,[d_1]\,s & s_1 \mid [d]\,s_2 \equiv [d]\,(s_1 \mid s_2) \quad \text{if } d \notin \mathrm{fd}(s_1) \cup \mathrm{fk}(s_2) &
\end{array}
$$

Table 2
COWS structural congruence (excerpt of laws)

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

Table 3
Matching rules

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 $\overline{w}$ and $\overline{v}$ do match, $\mathcal{M}(\overline{w}, \overline{v})$ returns a substitution for the variables in $\overline{w}$; otherwise, it is undefined. *Substitutions* (ranged over by σ) are functions mapping variables to values and are written as collections of pairs of the form $x \mapsto v$. Application of substitution σ to s, written $s \cdot \sigma$, has the effect of replacing every free occurrence of x in s with v, for each $x \mapsto v \in \sigma$, by possibly using alpha conversion for avoiding v to be captured by name delimitations within s. We use $|\sigma|$ to denote the number of pairs in σ and $\sigma_1 \uplus \sigma_2$ to denote the union of σ_1 and σ_2 when they have disjoint domains.

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

$$
halt(\mathbf{kill}(k)) = halt(\mathrm{u}!\overline{e}) = halt(g) = \mathbf{0} \qquad halt(\{\!|s|\!\}) = \{\!|s|\!\}
$$

$$
halt(s_1 \mid s_2) = halt(s_1) \mid halt(s_2) \qquad halt([d]\,s) = [d]\,halt(s) \qquad halt(*\,s) = *\,halt(s)
$$

Finally, in Table 4, we inductively define two predicates: $\mathrm{noKill}(s, d)$ holds true if either d is not a killer label or $d = k$ and s cannot immediately perform a free kill activity $\mathbf{kill}(k)$; $\mathrm{noConf}(s, \mathrm{n}, \overline{v}, \ell)$, with ℓ natural number, holds true if s does not produce communication conflicts, i.e. s cannot immediately perform a receive activity over the endpoint n which matches $\overline{v}$ and generates a substitution with fewer pairs than ℓ.

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

$$
\alpha \;::=\; \mathrm{n} \lhd \overline{v} \;\mid\; \mathrm{n} \rhd \overline{w} \;\mid\; \mathrm{n}\,\sigma\,\ell\,\overline{v} \;\mid\; k \;\mid\; \dagger
$$

In the sequel, we use $\mathrm{nvk}(\alpha)$ to denote the set of names, variables and killer labels occurring in α, except for $\alpha = \mathrm{n}\,\sigma\,\ell\,\overline{v}$ for which we let $\mathrm{nvk}(\mathrm{n}\,\sigma\,\ell\,\overline{v}) = \mathrm{nvk}(\sigma)$, where $\mathrm{nvk}(\{x \mapsto v\}) = \{x\} \cup \mathrm{fd}(v)$ and $\mathrm{nvk}(\sigma_1 \uplus \sigma_2) = \mathrm{nvk}(\sigma_1) \cup \mathrm{nvk}(\sigma_2)$.

$$\mathrm{noKill}(s,d) = \mathrm{true} \quad \text{if } \mathrm{fk}(d) = \emptyset \qquad \mathrm{noKill}(s \mid s', k) = \mathrm{noKill}(s,k) \wedge \mathrm{noKill}(s',k)$$

$$\mathrm{noKill}(\mathbf{kill}(k),k) = \mathrm{false} \qquad \mathrm{noKill}([d]\,s,k) = \mathrm{noKill}(s,k) \quad \text{if } d \neq k$$

$$\mathrm{noKill}(\mathbf{kill}(k'),k) = \mathrm{true} \quad \text{if } k \neq k' \qquad \mathrm{noKill}([k]\,s,k) = \mathrm{true}$$

$$\mathrm{noKill}(\mathrm{u}!\overline{e},k) = \mathrm{noKill}(g,k) = \mathrm{true} \qquad \mathrm{noKill}(\{\!|s|\!\},k) = \mathrm{noKill}(*\,s,k) = \mathrm{noKill}(s,k)$$

$$\mathrm{noConf}(\mathbf{kill}(k),\mathrm{n},\overline{v},\ell) = \mathrm{noConf}(\mathrm{u}!\overline{e},\mathrm{n},\overline{v},\ell) = \mathrm{noConf}(\mathbf{0},\mathrm{n},\overline{v},\ell) = \mathrm{true}$$

$$\mathrm{noConf}(\mathrm{n}'?\overline{w}.s,\mathrm{n},\overline{v},\ell) = \begin{cases} \text{false} & \text{if } \mathrm{n}' = \mathrm{n} \wedge\ |\mathcal{M}(\overline{w},\overline{v})| < \ell \\ \text{true} & \text{otherwise} \end{cases}$$

$$\mathrm{noConf}(g+g',\mathrm{n},\overline{v},\ell) = \mathrm{noConf}(g,\mathrm{n},\overline{v},\ell) \wedge \mathrm{noConf}(g',\mathrm{n},\overline{v},\ell)$$

$$\mathrm{noConf}(s \mid s',\mathrm{n},\overline{v},\ell) = \mathrm{noConf}(s,\mathrm{n},\overline{v},\ell) \wedge \mathrm{noConf}(s',\mathrm{n},\overline{v},\ell)$$

$$\mathrm{noConf}([d]\,s,\mathrm{n},\overline{v},\ell) = \begin{cases} \mathrm{noConf}(s,\mathrm{n},\overline{v},\ell) & \text{if } d \notin \mathrm{n} \\ \text{true} & \text{otherwise} \end{cases}$$

$$\mathrm{noConf}(\{\!|s|\!\},\mathrm{n},\overline{v},\ell) = \mathrm{noConf}(*\,s,\mathrm{n},\overline{v},\ell) = \mathrm{noConf}(s,\mathrm{n},\overline{v},\ell)$$

Table 4
There are not active $\mathbf{kill}(k)$ / There are not conflicting receives along n matching $\overline{v}$

$$\mathbf{kill}(k) \xrightarrow{k} \mathbf{0} \ \textit{(kill)} \qquad \mathrm{n}?\overline{w}.s \xrightarrow{\mathrm{n} \rhd \overline{w}} s \ \textit{(rec)} \qquad \frac{[\![\overline{e}]\!] = \overline{v}}{\mathrm{n}!\overline{e} \xrightarrow{\mathrm{n} \lhd \overline{v}} \mathbf{0}} \ \textit{(inv)} \qquad \frac{g \xrightarrow{\alpha} s}{g + g' \xrightarrow{\alpha} s} \ \textit{(choice)}$$

$$\frac{s \xrightarrow{\mathrm{n}\sigma \uplus \{x \mapsto v\}\,\ell\,\overline{v}} s'}{[x]\,s \xrightarrow{\mathrm{n}\sigma\,\ell\,\overline{v}} s' \cdot \{x \mapsto v\}} \ \textit{(del}_{sub}\textit{)} \qquad \frac{s \xrightarrow{k} s'}{[k]\,s \xrightarrow{\dagger} [k]\,s'} \ \textit{(del}_{kill}\textit{)} \qquad \frac{s \xrightarrow{k} s' \quad k \neq d}{[d]\,s \xrightarrow{k} [d]\,s'} \ \textit{(pass}_{k}\textit{)} \qquad \frac{s \xrightarrow{\dagger} s'}{[d]\,s \xrightarrow{\dagger} [d]\,s'} \ \textit{(pass}_{\dagger}\textit{)}$$

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin \mathrm{nvk}(\alpha) \quad \alpha \neq k, \dagger \quad \mathrm{noKill}(s,d)}{[d]\,s \xrightarrow{\alpha} [d]\,s'} \ \textit{(pass}_{\alpha}\textit{)} \qquad \frac{s \equiv s_1 \quad s_1 \xrightarrow{\alpha} s_2 \quad s_2 \equiv s'}{s \xrightarrow{\alpha} s'} \ \textit{(cong)}$$

$$\frac{s_1 \xrightarrow{\mathrm{n} \rhd \overline{w}} s_1' \quad s_2 \xrightarrow{\mathrm{n} \lhd \overline{v}} s_2' \quad \mathcal{M}(\overline{w},\overline{v}) = \sigma \quad \mathrm{noConf}(s_1 \mid s_2,\mathrm{n},\overline{v},|\sigma|)}{s_1 \mid s_2 \xrightarrow{\mathrm{n}\sigma|\sigma|\overline{v}} s_1' \mid s_2'} \ \textit{(com)} \qquad \frac{s_1 \xrightarrow{\mathrm{n}\sigma\,\ell\,\overline{v}} s_1' \quad \mathrm{noConf}(s_2,\mathrm{n},\overline{v},\ell)}{s_1 \mid s_2 \xrightarrow{\mathrm{n}\sigma\,\ell\,\overline{v}} s_1' \mid s_2} \ \textit{(par}_{conf}\textit{)}$$

$$\frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}} \ \textit{(prot)} \qquad \frac{s_1 \xrightarrow{k} s_1'}{s_1 \mid s_2 \xrightarrow{k} s_1' \mid \mathit{halt}(s_2)} \ \textit{(par}_{kill}\textit{)} \qquad \frac{s_1 \xrightarrow{\alpha} s_1' \quad \alpha \neq k, \mathrm{n}\,\sigma\,\ell\,\overline{v}}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2} \ \textit{(par}_{pass}\textit{)}$$

Table 5
COWS operational semantics

The meaning of labels is as follows: $\mathrm{n} \lhd \overline{v}$ and $\mathrm{n} \rhd \overline{w}$ denote execution of invoke and receive activities over the endpoint n, respectively, $\mathrm{n}\,\sigma\,\ell\,\overline{v}$ (if $\sigma \neq \emptyset$) denotes execution of a communication over n with matching values $\overline{v}$, generated substitution having ℓ pairs, and substitution σ to be still applied, k denotes execution of a request for terminating a term from within the delimitation $[k]$, $\dagger$ and $\mathrm{n}\,\emptyset\,\ell\,\overline{v}$ 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 $\mathtt{n}\,\emptyset\,\ell\,\overline{v}$ (for some $\mathtt{n}$, ℓ and $\overline{v}$); services s_i, for each i, will be called *reducts* of s_0.

We comment on salient points. Activity **kill**(k) forces termination of all unprotected parallel activities (rules *(kill)* and *(par$_{kill}$)*) inside an enclosing $[k]$, that stops the killing effect by turning the transition label 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 free kill activity for d is active in s and α does not correspond to a kill activity (rules *(pass$_k$)*, *(pass$_\dagger$)* and *(pass$_\alpha$)*): in such cases the transition should be derived by using rules *(del$_{sub}$)* or *(del$_{kill}$)*. In other words, kill activities are executed *eagerly* with respect to the other activities inside the corresponding killer label delimitations.

A service invocation can proceed only if the expressions in the argument can be evaluated (rule *(inv)*). A receive activity offers an invocable operation along a given partner name (rule *(rec)*), and the execution of a receive permits to take a decision between alternative behaviours (rule *(choice)*). Communication can take place when two parallel services perform matching receive and invoke activities (rule *(com)*). Communication generates a substitution that is recorded in the transition label (for subsequent application), rather than a silent transition as in most process calculi. If more then one matching is possible, the receive that needs fewer substitutions is selected to progress (rules *(com)* and *(par$_{conf}$)*). This mechanism, based on pattern-matching, permits to correlate different service communications logically forming a same interaction 'session' by means of their same contents 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. For this reason they are called 'write once' variables. Rule *(cong)* is standard and states that structurally congruent services have the same transitions.

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$_{conf}$)*). In practice, COWS parallel operator is equipped with a priority mechanism which allows some actions to take precedence over others. Receive activities are assigned priority values which depend on the messages available so that, in presence of concurrent matching receives, only a receive using a more defined pattern (i.e. having greater priority) can proceed. This way, service definitions and service instances are represented as processes running concurrently, but service instances take precedence over the corresponding service definition when both can process the same message, thus preventing creation of wrong new instances. Additionally, kill activities have assigned greatest priority so that they pre-empt all other activities inside the enclosing killer label's delimitation and cause termination of those unprotected activities. This way, they turn out to be quite useful for handling situations where abnormal termination is required, like in case of fault and exception throw, or compensation invocation.

4 A symbolic semantics for COWS

In this section, we introduce a symbolic operational semantics for COWS. For the sake of simplicity, here we consider a *monadic* version of COWS, i.e. communication activities are of the form $\mathtt{u}!e$ and $\mathtt{n}?w.s$ (we discuss in Section 5.2 how to tailor the symbolic semantics to handle polyadic communication). Many illustrative examples shed light on the technical development.

4.1 Symbolic operational semantics

The symbolic operational semantics of COWS is defined over configurations of the form $\Phi, \Delta \vdash s$, called *constrained services*, where Φ is the *condition* that must hold to reach the current state, Δ is the *set of private names* previously exported, and s is a service whose actions are determined by Φ and Δ. The set Δ will be omitted when empty, writing e.g. $\Phi \vdash s$ instead of $\Phi, \emptyset \vdash s$. We define the semantics over an enriched set of services that also includes those auxiliary terms resulting from replacing (free occurrences of) variables with *unknown values* in terms produced by the syntax introduced in Section 3, where now expressions contain also unknown values. Therefore, in the extended syntax we use $\underline{u} \bullet \underline{u'}!e$ and $p \bullet o?\underline{w}.s$ to denote invoke and receive activities respectively, $\underline{x}$ to denote unknown values, and $\underline{t}$ to denote an unknown value or a term t (where t can be n, v, u, w, $\mathtt{n}$ or $\mathtt{u}$).

As in the standard semantics, the only *binding* construct is delimitation: let $\Phi, \Delta \vdash \mathbb{C}[\![[d]\, s]\!]$ be a constrained service (where $\mathbb{C}$ is a context [1]), $[d]$ binds d in the scope s, in the condition Φ and in the set Δ. We denote by $\mathrm{bn}(t)$ the set of names that occur bound in a term t, and by $\mathrm{uvar}(t)$ the set of variables that have been replaced by corresponding unknown values in t (i.e. if $\underline{x}$ is an unknown value in t, then $x \in \mathrm{uvar}(t)$). For simplicity sake, in the sequel we assume that bound variables in constrained services are pairwise distinct and different from variables corresponding to the unknown values of the constrained services, and bound names are all distinct and different from the free ones (of course, these conditions are not restrictive and can always be fulfilled by possibly using alpha-conversion). This assumption avoids that distinct unknown values are denoted by the same $\underline{x}$ in a condition Φ of a constrained service (see Example "Evaluation function, condition $x \notin \mathbf{uv}$ and assumption on bound variables" in Section 4.2), and permits identifying the name delimitation binding each private name within a condition Φ and a set Δ of a constrained service (see Remark 4.1).

The symbolic operational semantics of COWS is defined only for *closed* services, and is given in terms of a structural congruence and of a (bi-)labelled transition relation. The structural congruence $\equiv$ is the trivial extension of that defined in Section 3 to the enriched syntax of services used here. To define the labelled transition relation, we exploit the trivial extension to the enriched syntax of function $\mathit{halt}(\_)$ and predicate $\mathrm{noKill}(\_,\_)$ defined in Section 3. We also extend function $[\![\_]\!]$ to deal with unknown values. Now, it takes a closed expression and returns a pair $(\Phi, \underline{v})$: the (possibly unknown) value $\underline{v}$ is the result of the evaluation provided that the condition Φ holds. Specifically, let e be an expression, if e does not contain unknown values and can be computed, then $[\![e]\!] = (\mathit{true}, v)$ where v is the result of the evaluation, as in the original COWS semantics. Similarly, if e is an unknown value $\underline{x}$, then $[\![e]\!] = (\mathit{true}, \underline{x})$. If e contains unknown values and is not a single unknown

[1] A context $\mathbb{C}$ is a service with a 'hole' $[\![\cdot]\!]$ such that, once the hole is filled with a service s, the resulting term $\mathbb{C}[\![s]\!]$ is a COWS service.

value (i.e. $e \neq \underline{x}$ for every $\underline{x}$), then $[\![e]\!] = ((\underline{y} \neq \mathbf{bn} \wedge y \notin \mathbf{uv} \wedge \underline{y} = e \wedge \Phi'), \underline{y})$ where $\underline{y}$ is a fresh unknown value that must be different from all private names (i.e. $\underline{y} \neq \mathbf{bn}$) and from all existent unknown values (i.e. $y \notin \mathbf{uv}$)[2], and Φ' is a condition that permits dealing with expression operators partially defined[3]. Function $[\![\_]\!]$, and hence condition Φ', cannot be explicitly defined because the exact syntax of expressions is deliberately not specified. Then, consider as an example the following simple language for expressions:

$$e \;::=\; x \;\mid\; \underline{x} \;\mid\; i \;\mid\; e + e \;\mid\; e - e \;\mid\; e * e \;\mid\; e/e \;\mid\; (e)$$

where i is an integer value. For the above language function $[\![\_]\!]$ is such that:

- $[\![(5-2)*3]\!] = (\mathit{true}, 9)$;
- $[\![5 - x]\!]$ is undefined, because the expression $5 - x$ is not closed;
- $[\![5 - \underline{x}]\!] = ((\underline{y} \neq \mathbf{bn} \wedge y \notin \mathbf{uv} \wedge \underline{y} = 5 - \underline{x}), \underline{y})$;
- $[\![5/0]\!]$ is undefined;
- $[\![5/\underline{x}]\!] = ((\underline{y} \neq \mathbf{bn} \wedge y \notin \mathbf{uv} \wedge \underline{y} = 5/\underline{x} \wedge \underline{x} \neq 0), \underline{y})$, where condition $\underline{x} \neq 0$ is due to the fact that operator / is not defined when its second argument is 0.

We also define a function $\mathit{confRec}(\_,\_)$, that takes a service s and an endpoint $\mathtt{n}$ as an arguments and returns the set of (possibly unknown) values that are parameters of receive activities over the endpoint $\mathtt{n}$ active in s. This function plays the role of predicate $\mathrm{noConf}(\_,\_,\_,\_)$ of the standard semantics and, indeed, is exploited to disable transitions in case of communication conflicts (by setting transition conditions to *false*). The function is inductively defined as follows:

$$\mathit{confRec}(\mathbf{0},\mathtt{n}) = \mathit{confRec}(\mathbf{kill}(k),\mathtt{n}) = \mathit{confRec}(\mathtt{u}!e,\mathtt{n}) = \mathit{confRec}(\mathtt{n}?x.s,\mathtt{n}) = \emptyset$$

$$\mathit{confRec}(g+g',\mathtt{n}) = \mathit{confRec}(g,\mathtt{n}) \cup \mathit{confRec}(g',\mathtt{n}) \qquad \mathit{confRec}(\mathtt{n}?\underline{v}.s,\mathtt{n}) = \{\underline{v}\}$$

$$\mathit{confRec}(\mathtt{n}'?\underline{w}.s,\mathtt{n}) = \emptyset \;\text{ if } \mathtt{n} \neq \mathtt{n}' \qquad \mathit{confRec}(\{\!|s|\!\},\mathtt{n}) = \mathit{confRec}(s,\mathtt{n})$$

$$\mathit{confRec}(s \mid s',\mathtt{n}) = \mathit{confRec}(s,\mathtt{n}) \cup \mathit{confRec}(s',\mathtt{n}) \qquad \mathit{confRec}([d]\,s,\mathtt{n}) = \emptyset \;\text{ if } d \in \mathtt{n}$$

$$\mathit{confRec}([d]\,s,\mathtt{n}) = \mathit{confRec}(s,\mathtt{n})\backslash\{d\} \;\text{ if } d \notin \mathtt{n} \qquad \mathit{confRec}(*\,s,\mathtt{n}) = \mathit{confRec}(s,\mathtt{n})$$

The labelled transition relation over constrained services, written $\overset{\Phi,\alpha}{\rightarrowtail}$, relies on a labelled transition relation $\overset{\Phi,\alpha}{\longrightarrow}$, that is the least relation over services induced by the rules in Table 6. Conditions Φ and actions α are generated by the following grammar:

$$\begin{array}{rcl}\Phi & ::= & \mathit{true} \mid \mathit{false} \mid \underline{v} = \underline{v}' \mid \underline{v} \neq \underline{v}' \mid \underline{x} \neq \mathbf{bn} \mid x \notin \mathbf{uv} \\ & \mid & x \notin \{x_i\}_{i\in I} \mid \underline{x} = e \mid \Phi \wedge \Phi'\end{array}$$

$$\alpha \;::=\; \mathtt{n} \lhd \underline{v} \;\mid\; \mathtt{n} \lhd [n] \;\mid\; \mathtt{n} \rhd \underline{w} \;\mid\; \mathtt{n} \rhd [x] \;\mid\; \mathtt{n}\,\sigma\,\ell\,\underline{v} \;\mid\; k \;\mid\; \dagger$$

where, now, a substitutions σ can be either the empty substitution $\emptyset$ or a substitution $\{x \mapsto \underline{v}\}$ that maps the variable x to the (possibly unknown) value $\underline{v}$.

The meaning of labels is as follows:

[2] Notably, here $\underline{y}$ can be any unknown value, provide that it satisfies conditions $\underline{y} \neq \mathbf{bn}$ and $y \notin \mathbf{uv}$. Notice that condition $y \notin \mathbf{uv}$ is a syntactical condition on the variable name y. Later we shall explain the exact meaning of the above conditions and show how they are evaluated in the last step of the inference of a transition.

[3] Of course, if all operators used in the considered expression are total functions, then condition Φ' is *true*.

$$\mathbf{kill}(k) \xrightarrow{true,\,k} \mathbf{0} \quad \textit{(s-kill)} \qquad \mathrm{n}?\underline{w}.s \xrightarrow{true,\,\mathrm{n}\rhd\underline{w}} s \quad \textit{(s-rec)}$$

$$\frac{s \xrightarrow{\Phi,\,\mathrm{n}\rhd x} s'}{[x]\,s \xrightarrow{\Phi\wedge\underline{x}\neq\mathbf{bn}\wedge\underline{x}\neq confRec(\mathrm{s},\mathrm{n}),\,\mathrm{n}\rhd[x]} s'\cdot\{x\mapsto\underline{x}\}} \textit{(s-rec}_{com}\textit{)} \qquad \frac{g \xrightarrow{\Phi,\,\alpha} s}{g+g' \xrightarrow{\Phi,\,\alpha} s} \textit{(s-choice)}$$

$$\frac{[\![e]\!] = (\Phi,\underline{v})}{\underline{\mathrm{n}}!e \xrightarrow{\Phi,\,\underline{\mathrm{n}}\lhd\underline{v}} \mathbf{0}} \textit{(s-inv)} \qquad \frac{s \xrightarrow{\Phi,\,\underline{\mathrm{n}}\lhd n} s' \quad n\notin\underline{\mathrm{n}}}{[n]\,s \xrightarrow{\Phi,\,\underline{\mathrm{n}}\lhd[n]} s'} \textit{(s-open)}$$

$$\frac{s \xrightarrow{\Phi,\,\mathrm{n}\{x\mapsto\underline{v}\}\,1\,\underline{v}} s'}{[x]\,s \xrightarrow{\Phi,\,\mathrm{n}\emptyset 1\,\underline{v}} s'\cdot\{x\mapsto\underline{v}\}} \textit{(s-del}_{sub}\textit{)} \qquad \frac{s \xrightarrow{\Phi,\,k} s'}{[k]\,s \xrightarrow{\Phi,\,\dagger} [k]\,s'} \textit{(s-del}_{kill}\textit{)}$$

$$\frac{s \xrightarrow{\Phi,\,k} s' \quad k\neq d}{[d]\,s \xrightarrow{\Phi,\,k} [d]\,s'} \textit{(s-pass}_{k}\textit{)} \qquad \frac{s \xrightarrow{\Phi,\,\dagger} s'}{[d]\,s \xrightarrow{\Phi,\,\dagger} [d]\,s'} \textit{(s-pass}_{\dagger}\textit{)}$$

$$\frac{s \xrightarrow{\Phi,\,\alpha} s' \quad d\notin \mathrm{nvk}(\alpha) \quad \alpha\neq k,\dagger \quad \mathrm{noKill}(s,d)}{[d]\,s \xrightarrow{\Phi,\,\alpha} [d]\,s'} \textit{(s-pass}_{\alpha}\textit{)} \qquad \frac{s \xrightarrow{\Phi,\,\alpha} s'}{\{\!|s|\!\} \xrightarrow{\Phi,\,\alpha} \{\!|s'|\!\}} \textit{(s-prot)}$$

$$\frac{s_1 \xrightarrow{\Phi_1,\,\mathrm{n}\rhd\underline{v}'} s_1' \qquad s_2 \xrightarrow{\Phi_2,\,\underline{\mathrm{n}}'\lhd\underline{v}} s_2'}{s_1 \mid s_2 \xrightarrow{\Phi_1\wedge\Phi_2\wedge\mathrm{n}=\underline{\mathrm{n}}'\wedge\underline{v}'=\underline{v},\,\mathrm{n}\emptyset 0\,\underline{v}} s_1' \mid s_2'} \textit{(s-match)}$$

$$\frac{s_1 \xrightarrow{\Phi_1,\,\mathrm{n}\rhd x} s_1' \qquad s_2 \xrightarrow{\Phi_2,\,\underline{\mathrm{n}}'\lhd\underline{v}} s_2'}{s_1 \mid s_2 \xrightarrow{\Phi_1\wedge\Phi_2\wedge\mathrm{n}=\underline{\mathrm{n}}'\wedge\underline{v}\neq confRec(s_1\mid s_2,\mathrm{n}),\,\mathrm{n}\{x\mapsto\underline{v}\}\,1\,\underline{v}} s_1' \mid s_2'} \textit{(s-com)}$$

$$\frac{s_1 \xrightarrow{\Phi,\,\mathrm{n}\sigma\,1\,\underline{v}} s_1'}{s_1 \mid s_2 \xrightarrow{\Phi\wedge\underline{v}\neq confRec(s_2,\mathrm{n}),\,\mathrm{n}\sigma\,1\,\underline{v}} s_1' \mid s_2} \textit{(s-par}_{conf1}\textit{)} \qquad \frac{s_1 \xrightarrow{\Phi,\,k} s_1'}{s_1 \mid s_2 \xrightarrow{\Phi,\,k} s_1' \mid halt(s_2)} \textit{(s-par}_{kill}\textit{)}$$

$$\frac{s_1 \xrightarrow{\Phi,\,\mathrm{n}\rhd[x]} s_1'}{s_1 \mid s_2 \xrightarrow{\Phi\wedge\underline{x}\neq confRec(s_2,\mathrm{n}),\,\mathrm{n}\rhd[x]} s_1' \mid s_2} \textit{(s-par}_{conf2}\textit{)} \qquad \frac{s\equiv s_1 \quad s_1 \xrightarrow{\Phi,\,\alpha} s_2 \quad s_2\equiv s'}{s \xrightarrow{\Phi,\,\alpha} s'} \textit{(s-cong)}$$

$$\frac{s_1 \xrightarrow{\Phi,\,\alpha} s_1' \qquad \alpha\neq k,\,\mathrm{n}\sigma\,1\,\underline{v},\,\mathrm{n}\rhd[x]}{s_1 \mid s_2 \xrightarrow{\Phi,\,\alpha} s_1' \mid s_2} \textit{(s-par}_{pass}\textit{)}$$

Table 6
COWS symbolic semantics (rules for $\xrightarrow{\Phi,\,\alpha}$)

- *Conditions*: *true* (resp. *false*) denotes the condition always (resp. never) satisfied, $\underline{v} = \underline{v}'$ (resp. $\underline{v} \neq \underline{v}'$) denotes an equality (resp. inequality) between (possibly unknown) values, $\underline{x} \neq \mathbf{bn}$ means that the unknown value $\underline{x}$ must be different from all bound names of the considered service, $x \notin \mathbf{uv}$ means that the set of variables corresponding to the unknown values of the considered constrained service may not contain the variable x, $x \notin \{x_i\}_{i\in I}$ means that x must not be in the set $\{x_i\}_{i\in I}$, $\underline{x} = e$ states that the unknown value $\underline{x}$ is equal to the evaluation of the closed non-evaluable expression e (conditions of this form are

generated by the evaluation function, e.g. condition $\underline{y} = 5/\underline{x}$ is generated by evaluation of expression $5/\underline{x}$), and as usual $\wedge$ denotes the logic conjunction. In the sequel, we will use notation $\underline{v} \neq \{\underline{v}_1, \ldots, \underline{v}_n\}$ to indicate the condition $\underline{v} \neq \underline{v}_1 \wedge \ldots \wedge \underline{v} \neq \underline{v}_n$ (where $\underline{v} \neq \emptyset$ indicates *true*). Moreover, we will use a function $\mathcal{B}(\_,\_,\_)$ that, given a condition Φ, a service s and a set of variables $\{x_i\}_{i\in I}$, returns a condition obtained by conjuncting Φ with all inequalities between the unknown values of Φ and the bound names of s and with all conditions $x \notin \{x_i\}_{i\in I}$ for each $x \notin \mathbf{uv}$ in Φ. Formally, $\mathcal{B}(\_,\_,\_)$ is defined as follows:

$$\mathcal{B}(\mathit{true}, s, \{x_i\}_{i\in I}) = \mathit{true} \qquad \mathcal{B}(\mathit{false}, s, \{x_i\}_{i\in I}) = \mathit{false}$$
$$\mathcal{B}(\underline{v} = \underline{v}', s, \{x_i\}_{i\in I}) = \underline{v} = \underline{v}' \qquad \mathcal{B}(\underline{v} \neq \underline{v}', s, \{x_i\}_{i\in I}) = \underline{v} \neq \underline{v}'$$
$$\mathcal{B}(\underline{x} \neq \mathbf{bn}, s, \{x_i\}_{i\in I}) = \underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq \mathrm{bn}(s) \qquad \mathcal{B}(x \notin \mathbf{uv}, s, \{x_i\}_{i\in I}) = x \notin \{x_i\}_{i\in I}$$
$$\mathcal{B}(x \notin \{y_j\}_{j\in J}, s, \{x_i\}_{i\in I}) = x \notin \{y_j\}_{j\in J} \qquad \mathcal{B}(\underline{x} = e, s, \{x_i\}_{i\in I}) = \underline{x} = e$$
$$\mathcal{B}(\Phi \wedge \Phi', s, \{x_i\}_{i\in I}) = \mathcal{B}(\Phi, s, \{x_i\}_{i\in I}) \wedge \mathcal{B}(\Phi', s, \{x_i\}_{i\in I})$$

- *Actions*: $\underline{\mathtt{n}} \lhd [n]$ denotes execution of a bound invoke activity over the endpoint $\underline{\mathtt{n}}$, while $\mathtt{n} \rhd [x]$ denotes taking place of external communication over the endpoint $\mathtt{n}$ with receive parameter x (that will be replaced by the unknown value $\underline{x}$). The remaining labels have the usual meaning. Notably, due to the restraint on monadic communication, here the natural number ℓ can only be either 0 or 1.

We comment on the aspects of the symbolic semantics rules that mainly differ from the standard ones. Bound invocations, that transmit private names, can be generated by rule *(s-open)*. Notably, bound invocation actions do not appear in rules *(s-match)* and *(s-com)*, and therefore cannot directly interact with receive actions. Such interactions are instead inferred by using structural congruence to pull name delimitation outside both interacting activities. Although the bound transitions and rule *(s-open)* can be omitted, we include them both to give a proper semantics to terms $[n]\mathtt{n}!n$ and to support the development of behavioural equivalences for COWS. Communication can be *internal* or *external* to the service. Internal communication can take place when two matching receive and invoke activities (rules *(s-match)* and *(s-com)*) are simultaneously executed. External communication can take place when a value is transmitted to the environment (rules *(s-inv)* and *(s-open)*) or when a receive activity matches an unknown value provided by the environment (rules *(s-rec)* and *(s-rec$_{com}$)*). Differently from the standard semantics, conflicting receives cannot be dealt with by using a predicate in the premises of rules for communication and interleaving, because unknown values can be involved. Here, the check for conflicting receives is simply a condition of the form $\underline{v} \neq \mathit{confRec}(s, \mathtt{n})$ (rules *(s-rec$_{com}$)*, *(s-com)*, *(s-par$_{conf1}$)* and *(s-par$_{conf2}$)*).

The labelled transition relation $\overset{\Phi,\alpha}{\succ\!\!\longrightarrow}$ is the least relation over constrained services induced by the rules reported in Table 7, where notation $\underline{\mathtt{n}} \notin \Delta$ means that set Δ does not contain the names of endpoint $\underline{\mathtt{n}}$. Rule *(constServ)* states that a constrained service $\Phi, \Delta \vdash s$ can perform all the 'non-invoke' transitions performed by s with an enriched condition Φ'' obtained by composing Φ and the condition on the label Φ'. Condition Φ'' takes care of the relationship between unknown values and private names. Indeed, by private names definition, each unknown value, that is a value coming from the environment, must be different from all bound (private) names of the considered service. If the transition $s \xrightarrow{\Phi',\alpha}$

$$\frac{s \xrightarrow{\Phi',\alpha} s' \qquad \alpha \neq \underline{\mathrm{n}} \lhd [n]\ ,\ \underline{\mathrm{n}} \lhd \underline{v} \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'',\alpha} \Phi'', \Delta \vdash s'} \textit{(constServ)}$$

$$\frac{s \xrightarrow{\Phi',\, \underline{\mathrm{n}} \lhd [n]} s' \qquad \underline{\mathrm{n}} \notin \Delta \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'',\, \underline{\mathrm{n}} \lhd [n]} \Phi'', \Delta \cup \{n\} \vdash s'} \textit{(constServ}_{exp}\textit{)}$$

$$\frac{s \xrightarrow{\Phi',\, \underline{\mathrm{n}} \lhd \underline{v}} s' \qquad \underline{\mathrm{n}} \notin \Delta \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'',\, \underline{\mathrm{n}} \lhd \underline{v}} \Phi'', \Delta \vdash s'} \textit{(constServ}_{inv}\textit{)}$$

Table 7
COWS symbolic semantics (rules for $\succ\!\!\xrightarrow{\Phi,\alpha}$)

s' introduces a new unknown value $\underline{x}$ (rules *(s-inv)* and *(s-rec$_{com}$)*), it is not sufficient to add the condition $\underline{x} \neq \mathrm{bn}(s')$ (i.e. the unknown value is different from all bound names of the current service), but we need also to consider bound names that could be subsequently generated. For example, let us consider the following transition:

$$\mathit{true} \vdash [x]\,\mathrm{n}?x.s \mid *[n]\,\mathrm{n}'!n \succ\!\!\xrightarrow{\underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq n\,,\,\mathrm{n} \rhd [x]} \underline{x} \neq \mathbf{bn} \ \wedge\ \underline{x} \neq n \vdash s \cdot \{x \mapsto \underline{x}\} \mid *[n]\,\mathrm{n}'!n$$

Now, if the obtained service performs the transition:

$$s \cdot \{x \mapsto \underline{x}\} \mid *[n]\,\mathrm{n}'!n \xrightarrow{\Phi,\alpha} s' \cdot \{x \mapsto \underline{x}\} \mid *[n]\,\mathrm{n}'!n \mid [n']\,\mathrm{n}'!n' \mid [n'']\,\mathrm{n}'!n''$$

then, let $\Phi' \vdash s''$ be the obtained constrained service, the condition Φ' must contain $\underline{x} \neq n'$ and $\underline{x} \neq n''$. To update after any transition the condition of a constrained service with inequalities between unknown values and private names, we use the condition $\underline{x} \neq \mathbf{bn}$, that simply states that $\underline{x}$ has been introduced in the considered term (rules *(s-inv)* and *(s-rec$_{com}$)*), and function $\mathcal{B}(\_,\_,\_)$, that adds the inequalities for each unknown value (rules *(constServ)*, *(constServ$_{exp}$)* and *(constServ$_{inv}$)*). Moreover, function $\mathcal{B}(\_,\_,\_)$ adds conditions of the form $x \notin \{x_i\}_{i \in I}$ to guarantee that unknown values introduced by rule *(s-inv)* because expression evaluation differ from those of the considered constrained service (i.e. uvar(Φ) if the constrained service is $\Phi, \Delta \vdash s$; for further details see Example "Evaluation function, condition $x \notin \mathbf{uv}$ and assumption on bound variables" in Section 4.2).

Rules *(constServ$_{exp}$)* and *(constServ$_{inv}$)* deal with the localized receiving feature of COWS. Indeed, if a COWS term communicates a private (partner or operation) name to the environment, then the latter (that is a COWS context) can use the name to define a sending endpoint, but not a receiving one. For example, consider the following constrained service:

$$\mathit{true} \vdash [p]\,(q \bullet o!p \mid p \bullet o'!v)$$

It can perform the activity $q \bullet o!p$ (rule *(s-open)*) and become the term $\mathit{true}, \{p\} \vdash p \bullet o'!v$ which is stuck. In fact, to further evolve it needs the environment to be able to perform first a receive $q \bullet o?x$ and then a receive along the endpoint $x \bullet o'$, that is disallowed by the syntax. Therefore, to block invoke activities performed along endpoints using previously

exported private names, we record all exported private names in the set Δ of the constrained service and perform the check $\underline{\mathrm{n}} \notin \Delta$ when an invoke activity along $\underline{\mathrm{n}}$ communicating with the environment is executed.

Remark 4.1 The assumption "bound names are all distinct and different from the free ones" is used to guarantee the correlation between conditions and services. For example, if we do not rely on this assumption, for the constrained service $\underline{x} \neq n \vdash [n]\, s \mid [n]\, s'$ we are not able to understand what are the occurrences of n referred by the condition $\underline{x} \neq n$. Moreover, the definition of bound names permits maintaining this correlation. For example, the constrained service $\underline{x} \neq n \vdash [n]\, \underline{x} \bullet o!n$ is not not alpha-equivalent to $\underline{x} \neq n \vdash [m]\, \underline{x} \bullet o!m$ but is equivalent to $\underline{x} \neq m \vdash [m]\, \underline{x} \bullet o!m$.

Remark 4.2 It is worth noticing that, in the definition of relation $\overset{\Phi,\alpha}{\succ\!\!\longrightarrow}$, the conditions are never evaluated. Thus, at operational semantics level, we do not distinguish unfeasible transitions (whose condition holds *false*) from feasible ones. For example, transitions having the following conditions are unfeasible: $(o_{req} = o_{resp})$, $(\underline{x} \neq \underline{x})$ and $(\underline{x} = \underline{y} \wedge \underline{x} \neq \underline{y})$. Of course, to identify unfeasible transitions, we can replace the condition Φ'' in the conclusion of rules *(constServ)*, $(constServ_{exp})$ and $(constServ_{inv})$ with $\mathcal{E}(\Phi'')$, where $\mathcal{E}(\_)$ is a function for evaluating conditions.

Remark 4.3 Since the transition relation $\overset{\Phi,\alpha}{\succ\!\!\longrightarrow}$ is defined over constrained services, i.e. configuration of the form $\Phi, \Delta \vdash s$, the operational semantics can be naturally interpreted on $\mathrm{L}^2\mathrm{TS}$ [10]. Indeed, each *edge label* (of the form Φ, α) indicates the condition which must hold for the transition to be enabled and the performed action, while each *state label* (of the form Φ, Δ) indicates the condition which must hold to reach the considered state from the initial one and the set of previously exported private names.

We can now formalize the correspondence between the original semantics introduced in Section 3 and the symbolic semantics. We exploit here a function $\mathcal{E}(\_)$ for evaluating conditions: it takes a condition Φ and returns *false* if certainly Φ does not hold; otherwise, it returns Φ. For example, $\mathcal{E}(\Phi' \wedge (5 = 3))$ is *false* whatever Φ' may be. Since a condition Φ can be of the form $\underline{x} = e$ and the syntax of expressions e is not specified, function $\mathcal{E}(\_)$ cannot be explicitly defined (as function $[\![\_]\!]$). For the proof of semantics correspondence, we use the following lemma concerning function $\mathcal{B}(\_,\_,\_)$. For the sake of simplicity, a condition Φ is deemed *favourable* if $\mathrm{uvar}(\Phi) = \emptyset$ and $\mathcal{E}(\Phi) \neq$ *false*, i.e. it does not contain unknown values and can be positively evaluated.

Lemma 4.4 *Let Φ be a favourable condition, then $\mathcal{E}(\mathcal{B}(\Phi, s, \emptyset)) \neq$ false for any s.*

Proof. Function $\mathcal{B}(\_,\_,\_)$ acts as an homomorphism on the first argument, except when the argument is $\underline{x} \neq \mathbf{bn}$ and $x \notin \mathbf{uv}$. We do not need to consider the former case because, by the hypothesis $\mathrm{uvar}(\Phi) = \emptyset$, we have that Φ does not contain unknown values. For the latter case, we get that $\mathcal{B}(x \notin \mathbf{uv}, s, \emptyset) = x \notin \mathbf{uv}$ since the third argument of $\mathcal{B}(\_,\_,\_)$ is $\emptyset$. Thus, the thesis trivially follows by the fact that, under the considered hypotheses, $\mathcal{B}(\_,\_,\_)$ acts always as an homomorphism on the first argument. □

Our major result is a theorem of 'operational correspondence'. It is quite standard and states that for each transition of the original LTS associated to a COWS term there exists a corresponding symbolic transition of the symbolic LTS that does not involve unknown

values and bound names, and vice versa. Notice that, since the original semantics does not take bound invocations into account, only constrained services of the form $\Phi \vdash s$ are considered in the theorem.

Theorem 4.5 *Let* $\mathrm{uvar}(\alpha) = \emptyset$ *and* $\alpha \neq \mathtt{n} \lhd [n]$. $s \xrightarrow{\alpha} s'$ *if and only if, for any favourable condition* Φ, $\Phi \vdash s \overset{\Phi', \alpha}{\succ\!\!\longrightarrow} \Phi' \vdash s'$ *for some favourable condition* Φ'.

Proof. The proof of the "if" part proceeds by induction on the length of the inference of $s \xrightarrow{\alpha} s'$. For the base case, we reason by case analysis on the axioms of the original operational semantics.

(kill) In this case, $\alpha = k$, $s = \mathbf{kill}(k)$ and $s' = \mathbf{0}$. By rule *(s-kill)*, $\mathbf{kill}(k) \xrightarrow{true, k} \mathbf{0}$. Then, by rule *(constServ)*, we get that $\Phi \vdash \mathbf{kill}(k) \overset{\Phi', k}{\succ\!\!\longrightarrow} \Phi' \vdash \mathbf{0}$, where $\Phi' = \mathcal{B}(\Phi \wedge true, \mathbf{0}, \emptyset)$ (since $\mathrm{uvar}(\Phi) = \emptyset$). By definition, $\mathcal{B}(\Phi \wedge true, \mathbf{0}, \emptyset) = \mathcal{B}(\Phi, \mathbf{0}, \emptyset) \wedge \mathcal{B}(true, \mathbf{0}, \emptyset)$. Since Φ is favourable, by Lemma 4.4, we have that $\mathcal{E}(\mathcal{B}(\Phi, \mathbf{0}, \emptyset)) \neq false$. Since $\mathcal{B}(true, \mathbf{0}, \emptyset) = true \neq false$, we can conclude that $\mathcal{E}(\Phi') \neq false$.

(rec) In this case, $\alpha = \mathtt{n} \rhd w$ and $s = \mathtt{n}?w.s'$. By rule *(s-rec)*, $\mathtt{n}?w.s' \xrightarrow{true, \mathtt{n} \rhd w} s'$. Then, by rule *(constServ)*, we get that $\Phi \vdash \mathtt{n}?w.s' \overset{\Phi', \mathtt{n} \rhd w}{\succ\!\!\longrightarrow} \Phi' \vdash s'$, where $\Phi' = \mathcal{B}(\Phi \wedge true, \mathbf{0}, \emptyset)$. As before, we can conclude that $\mathcal{E}(\Phi') \neq false$.

(inv) In this case, $\alpha = \mathtt{n} \lhd \overline{v}$, $s = \mathtt{n}!e$ where $[\![e]\!] = v$, and $s' = \mathbf{0}$. By rule *(s-inv)*, $\mathtt{n}!\overline{e} \xrightarrow{true, \mathtt{n} \lhd v} \mathbf{0}$. Then, by rule *(constServ$_{inv}$)*, we get that $\Phi \vdash \mathtt{n}!e \overset{\Phi', \mathtt{n} \lhd v}{\succ\!\!\longrightarrow} \Phi' \vdash \mathbf{0}$, where $\Phi' = \mathcal{B}(\Phi \wedge true, \mathbf{0}, \emptyset)$. As before, we can conclude that $\mathcal{E}(\Phi') \neq false$.

For the inductive step, we reason by case analysis on the last applied inference rule of the original operational semantics.

(choice) In this case, $s = g + g'$. By the premise of the rule *(choice)*, $g \xrightarrow{\alpha} s'$. By induction, $\Phi \vdash g \overset{\Phi', \alpha}{\succ\!\!\longrightarrow} \Phi' \vdash s'$ for some favourable conditions Φ and Φ'. By the premise of the rule *(constServ)*, we get that $g \xrightarrow{\Phi'', \alpha} s'$ where Φ'' is such that $\Phi' = \mathcal{B}(\Phi \wedge \Phi'', s', \emptyset)$. By rule *(s-choice)*, $g + g' \xrightarrow{\Phi'', \alpha} s'$. Finally, by rule *(constServ)*, we can conclude $\Phi \vdash g + g' \overset{\Phi', \alpha}{\succ\!\!\longrightarrow} \Phi' \vdash s'$.

(del$_{sub}$) In this case, $s = [x]\, s_1$ and $s' = s_2 \cdot \{x \mapsto v\}$. By the premise of the rule *(del$_{sub}$)*, $s_1 \xrightarrow{\mathtt{n}\{x \mapsto v\}\, 1\, v} s_2$. By induction, we get that $\Phi \vdash s_1 \overset{\Phi', \mathtt{n}\{x \mapsto v\}\, 1\, v}{\succ\!\!\longrightarrow} \Phi' \vdash s_2$ for some favourable conditions Φ and Φ'. By the premise of rule *(constServ)*, we get that $s_1 \xrightarrow{\Phi'', \mathtt{n}\{x \mapsto v\}\, 1\, v} s_2$ and, by rule *(del$_{sub}$)*, $[x]\, s_1 \xrightarrow{\Phi'', \mathtt{n}\emptyset\, 1\, v} s_2 \cdot \{x \mapsto v\}$. Finally, by rule *(constServ)*, we can conclude.

(del$_{kill}$), (pass$_k$), (pass$_\dagger$), (pass$_\alpha$), (cong), (prot), (par$_{kill}$), (par$_{pass}$), (par$_{conf}$) These cases are similar to the previous one; the latter case relies on the fact that $\mathrm{noConf}(s_2, \mathtt{n}, v, 1) = true$ implies that $confRec(s_2, \mathtt{n}) = \{v_i\}_{i \in I}$ such that $v \neq v_i$ for all $i \in I$.

(com) In this case, $s = (s_1 \mid s_2)$ and $s' = (s_1' \mid s_2')$. First, we consider the case $\alpha = \mathtt{n}\emptyset\, 0\, v$. By the premises of rule *(com)*, $s_1 \xrightarrow{\mathtt{n} \rhd v} s_1'$ and $s_2 \xrightarrow{\mathtt{n} \lhd v} s_2'$. By induction, we get that $\Phi_1 \vdash s_1 \overset{\Phi_1', \mathtt{n} \rhd v}{\succ\!\!\longrightarrow} \Phi_1' \vdash s_1'$ and $\Phi_2 \vdash s_2 \overset{\Phi_2', \mathtt{n} \lhd v}{\succ\!\!\longrightarrow} \Phi_2' \vdash s_2'$, for some favourable conditions Φ_1, Φ_1', Φ_2 and Φ_2'. By the premises of rules *(constServ)* and *(constServ$_{inv}$)*, we get

that $s_1 \xrightarrow{\Phi_1'', \mathtt{n} \rhd v} s_1'$ and $s_2 \xrightarrow{\Phi_2'', \mathtt{n} \lhd v} s_2'$, where conditions Φ_1'' and Φ_2'' are such that $\Phi_1' = \mathcal{B}(\Phi_1 \wedge \Phi_1'', s_1', \emptyset)$ and $\Phi_2' = \mathcal{B}(\Phi_2 \wedge \Phi_2'', s_2', \emptyset)$. By rule *(s-com)*, $s_1 \mid s_2 \xrightarrow{\Phi', \mathtt{n}\emptyset 0 v} s_1' \mid s_2'$, where $\Phi' = \Phi_1'' \wedge \Phi_2'' \wedge \mathtt{n} = \mathtt{n} \wedge v = v$. Finally, by rule *(constServ)*, we can conclude that $\Phi \vdash s_1 \mid s_2 \succ\!\xrightarrow{\Phi'', \mathtt{n}\emptyset 0 v} \Phi'' \vdash s_1' \mid s_2'$, where $\Phi'' = \mathcal{B}(\Phi \wedge \Phi', s_1' \mid s_2', \emptyset)$. The case $\alpha = \mathtt{n}\,\sigma\,1\,v$ proceeds as above, by also relying on the fact that $\mathrm{noConf}(s_1 \mid s_2, \mathtt{n}, v, 1) = \mathit{true}$ implies that $\mathit{confRec}(s_1 \mid s_2, \mathtt{n}) = \{v_i\}_{i \in I}$ with $v \neq v_i$ for all $i \in I$.

Consider now the "only if" part of the theorem. By the premises of rules *(constServ)* and *(constServ$_{inv}$)*, we get that $s \xrightarrow{\Phi'', \alpha} s'$ where $\Phi' = \mathcal{B}(\Phi \wedge \Phi'', s', \emptyset)$. By hypothesis $\mathcal{E}(\Phi') \neq \mathit{false}$, hence $\mathcal{E}(\Phi'') \neq \mathit{false}$ too. The proof proceeds by induction on the length of the inference of $s \xrightarrow{\Phi'', \alpha} s'$. We omit the details because the proof proceeds as that of the "if" part, but the steps are executed in the reverse order. For the base case, we reason by case analysis on the axioms of the symbolic operational semantics. We take a look at one base case:

(s-rec) In this case, $\Phi'' = \mathit{true}$, $\alpha = \mathtt{n} \rhd w$ and $s = \mathtt{n}?w.s'$. Trivially, by rule *(rec)*, $\mathtt{n}?w.s' \xrightarrow{\mathtt{n} \rhd w} s'$.

For the inductive step, we reason by case analysis on the last applied inference rule of the symbolic operational semantics. We take a look at two cases:

(s-choice) In this case, $s = g + g'$. By the premise of the rule *(s-choice)*, $g \xrightarrow{\Phi'', \alpha} s'$. By induction, we get that $g \xrightarrow{\alpha} s'$. Finally, by rule *(choice)*, we can conclude $g + g' \xrightarrow{\alpha} s'$.

(s-com) In this case, $s = (s_1 \mid s_2)$, $\Phi'' = (\Phi_1 \wedge \Phi_2 \wedge \mathtt{n} = \mathtt{n} \wedge v \neq \mathit{confRec}(s_1 \mid s_2, \mathtt{n}))$, $\alpha = \mathtt{n}\{x \mapsto v\}\,1\,v$ and $s' = (s_1' \mid s_2')$. Since $\mathcal{E}(\Phi'') \neq \mathit{false}$, we get that $\mathcal{E}(\Phi_1) \neq \mathit{false}$, $\mathcal{E}(\Phi_2) \neq \mathit{false}$ and $\mathit{confRec}(s_1 \mid s_2, \mathtt{n}) = \{v_i\}_{i \in I}$ such that $v \neq v_i$ for all $i \in I$. This means that $\mathrm{noConf}(s_1 \mid s_2, \mathtt{n}, v, 1)$ holds true. By induction and since $\mathcal{E}(\Phi_1) \neq \mathit{false}$ and $\mathcal{E}(\Phi_2) \neq \mathit{false}$, we have that $s_1 \xrightarrow{\mathtt{n} \rhd x} s_1'$ and $s_2 \xrightarrow{\mathtt{n} \lhd v} s_2'$. Thus, by rule *(com)*, we can conclude that $s_1 \mid s_2 \xrightarrow{\mathtt{n}\{x \mapsto v\}\,1\,v} s_1' \mid s_2'$.

□

4.2 Examples

In this section, we show some simple examples aimed at clarifying some peculiarities of COWS symbolic semantics. In the sequel, for the sake of readability, we shall evaluate conditions, writing e.g. $\underline{x} \neq n$ instead of $(p = p \wedge o = o \wedge \mathit{true} \wedge \underline{x} \neq n)$.

External communication

According to the operational semantics introduced in Section 3, the service $[x]\,\mathtt{n}?x.\,\mathtt{m}!x$ can perform the receive activity, but then it is blocked (because variable x is not instantiated by the receive transition). Instead, according to the symbolic semantics defined in this section,

the constrained service $true \vdash [x]\,\mathtt{n}?x.\,\mathtt{m}!x$ can evolve as follows:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{n}?x.\,\mathtt{m}!x \xrightarrow{true\,,\,\mathtt{n}\rhd x} \mathtt{m}!x}\ \textit{(s-rec)}}{[x]\,\mathtt{n}?x.\,\mathtt{m}!x \xrightarrow{\underline{x}\neq \mathit{confRec}((\mathtt{n}?x.\,\mathtt{m}!x),\mathtt{n})\,\wedge\,\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \mathtt{m}!\underline{x}}\ \textit{(s-rec}_{com}\textit{)}}{true \vdash [x]\,\mathtt{n}?x.\,\mathtt{m}!x \succ \xrightarrow{\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \underline{x}\neq\mathbf{bn} \vdash \mathtt{m}!\underline{x}}\ \textit{(constServ)}$$

since $(\underline{x} \neq \mathit{confRec}((\mathtt{n}?x.\,\mathtt{m}!x),\mathtt{n})) = (\underline{x} \neq \emptyset) = true$. Then, the continuation can perform the following transition:

$$\dfrac{\dfrac{\dfrac{[\![\underline{x}]\!] = (true, \underline{x})}{\mathtt{m}!\underline{x} \xrightarrow{true\,,\,\mathtt{m}\lhd\underline{x}} \mathbf{0}}\ \textit{(s-inv)}}{\underline{x}\neq\mathbf{bn} \vdash \mathtt{m}!\underline{x} \succ \xrightarrow{\underline{x}\neq\mathbf{bn}\,,\,\mathtt{m}\lhd\underline{x}} \underline{x}\neq\mathbf{bn} \vdash \mathbf{0}}\ \textit{(constServ}_{inv}\textit{)}$$

Notice that, although the external communication generates the condition $\underline{x} \neq \mathbf{bn}$ (that means that the received unknown value must be different from all delimited names), the condition is never exploited because the term does not contain delimited names.

External communication within name delimitations

Consider the constrained service $true \vdash [n]\,[x]\,\mathtt{n}?x.\,x \bullet o!n$. Differently from the previous example, the above service contains a delimited name (i.e. n). Thus, this time, condition $\underline{x} \neq \mathbf{bn}$ is exploited to generate the specific condition $\underline{x} \neq n$. Indeed, the service evolves as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{n}?x.\,x\bullet o!n \xrightarrow{true\,,\,\mathtt{n}\rhd x} x\bullet o!n}\ \textit{(s-rec)}}{[x]\,\mathtt{n}?x.\,x\bullet o!n \xrightarrow{\underline{x}\neq\mathit{confRec}((\mathtt{n}?x.\,x\bullet o!n),\mathtt{n})\,\wedge\,\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \underline{x}\bullet o!n}\ \textit{(s-rec}_{com}\textit{)}}{[n]\,[x]\,\mathtt{n}?x.\,x\bullet o!n \xrightarrow{\underline{x}\neq\mathit{confRec}((\mathtt{n}?x.\,x\bullet o!n),\mathtt{n})\,\wedge\,\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} [n]\,\underline{x}\bullet o!n}\ \textit{(s-del}_{pass}\textit{)}}{true \vdash [n]\,[x]\,\mathtt{n}?x.\,x\bullet o!n \succ \xrightarrow{\underline{x}\neq n\,\wedge\,\underline{x}\neq\mathbf{bn}\,,\,\mathtt{n}\rhd[x]} \underline{x}\neq n\,\wedge\,\underline{x}\neq\mathbf{bn} \vdash [n]\,\underline{x}\bullet o!n}\ \textit{(constServ)}$$

since $(\underline{x} \neq \mathit{confRec}((\mathtt{n}?x.\,x\bullet o!n),\mathtt{n}) = true$ and $\mathcal{B}(\underline{x} \neq \mathbf{bn}, ([n]\,[x]\,\mathtt{n}?x.\,x\bullet o!n), \emptyset) = \underline{x} \neq n \wedge \underline{x} \neq \mathbf{bn}$. Then, the continuation can evolve only provided that condition $\underline{x} \neq n$ holds.

Internal communication

Consider the constrained service $true \vdash [p]\,[x]\,(\,p \bullet o?x.\,\mathtt{n}!x \mid p \bullet o!v\,)$, where $p \notin \mathtt{n}$. In this case, due to the delimitation $[p]$, the receive activity cannot communicate with the

environment, but can synchronize with the internal invoke:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p\bullet o?x.\,\mathtt{n}!x \xrightarrow{true\,,\,p\bullet o\triangleright x} \mathtt{n}!x}\,(s\text{-}rec) \quad \dfrac{[\![v]\!] = (true, v)}{p\bullet o!v \xrightarrow{true\,,\,p\bullet o\triangleleft v} \mathbf{0}}\,(s\text{-}inv)}{p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v \xrightarrow{\Phi\,,\,p\bullet o\,\{x\mapsto v\}\,1\,v} \mathtt{n}!x}\,(s\text{-}com)}{[x](\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,) \xrightarrow{\Phi\,,\,p\bullet o\,\emptyset\,1\,v} \mathtt{n}!x\cdot\{x\mapsto v\}}\,(s\text{-}del_{sub})}{[p][x](\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,) \xrightarrow{\Phi\,,\,p\bullet o\,\emptyset\,1\,v} [p]\,\mathtt{n}!v \equiv \mathtt{n}!v}\,(s\text{-}del_{pass})}{[p][x](\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,) \xrightarrow{\Phi\,,\,p\bullet o\,\emptyset\,1\,v} \mathtt{n}!v}\,(s\text{-}cong)}{true \vdash [p][x](\,p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v\,) \succ\!\xrightarrow{\Phi\,,\,p\bullet o\,\emptyset\,1\,v} \Phi \vdash \mathtt{n}!v}\,(constServ)$$

where $\Phi = (\,true \wedge true \wedge p = p \wedge o = o \wedge v \neq confRec(p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v, p\bullet o)\,)$. Since $confRec(p\bullet o?x.\,\mathtt{n}!x \mid p\bullet o!v, p\bullet o) = \emptyset$, condition Φ holds *true*.

External and internal communication

Consider the constrained service $true \vdash [x](\,\mathtt{n}?x.\,\mathtt{m}!x \mid \mathtt{n}!v\,)$. In this case, both internal and external communication can take place. Its initial transitions are the following:

$$(ext.\ com.) \qquad true \vdash [x](\,\mathtt{n}?x.\,\mathtt{m}!x \mid \mathtt{n}!v\,) \succ\!\xrightarrow{\Phi\,,\,\mathtt{n}\triangleright[x]} \Phi \vdash \mathtt{m}!\underline{x} \mid \mathtt{n}!v$$

$$(ext.\ com.) \qquad true \vdash [x](\,\mathtt{n}?x.\,\mathtt{m}!x \mid \mathtt{n}!v\,) \succ\!\xrightarrow{\Phi\,,\,\mathtt{n}\triangleleft v} \Phi \vdash [x](\,\mathtt{n}?x.\,\mathtt{m}!x\,)$$

$$(int.\ com.) \qquad true \vdash [x](\,\mathtt{n}?x.\,\mathtt{m}!x \mid \mathtt{n}!v\,) \succ\!\xrightarrow{\Phi\,,\,\mathtt{n}\,\emptyset\,1\,v} \Phi \vdash \mathtt{m}!v$$

Conflicting receive

Consider the constrained service $true \vdash [x](\,\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,)$. Due to the presence of the receive $\mathtt{n}?v$, that has greater priority to synchronize with an invocation $\mathtt{n}!v$, the receive $\mathtt{n}?x$ can communicate with the environment only if the received value is not v (indeed, $confRec((\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v), \mathtt{n}) = \{v\}$):

$$true \vdash [x](\,\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,) \succ\!\xrightarrow{\underline{x}\neq\mathbf{bn}\,\wedge\,\underline{x}\neq v\,,\,\mathtt{n}\triangleright[x]} \underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq v \vdash \mathtt{n}?v \mid \mathtt{n}!v$$

Other possible transitions are as follows:

$$true \vdash [x](\,\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,) \succ\!\xrightarrow{true\,,\,\mathtt{n}\triangleleft v} true \vdash [x](\,\mathtt{n}?v \mid \mathtt{n}?x\,)$$

$$true \vdash [x](\,\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,) \succ\!\xrightarrow{true\,,\,\mathtt{n}\triangleright v} true \vdash [x](\,\mathtt{n}?x \mid \mathtt{n}!v\,)$$

$$true \vdash [x](\,\mathtt{n}?v \mid \mathtt{n}?x \mid \mathtt{n}!v\,) \succ\!\xrightarrow{true\,,\,\mathtt{n}\,\emptyset\,0\,v} true \vdash [x]\,\mathtt{n}?x$$

On constrained services

Consider the (plain) service $[x,y](\mathtt{n}?q \mid \mathtt{n}?x \mid x \bullet o!v \mid q \bullet o?y)$ where $\mathtt{n} \neq q \bullet o$. It can perform the following transition:

$$[x,y](\mathtt{n}?q \mid \mathtt{n}?x \mid x \bullet o!v \mid q \bullet o?y) \xrightarrow{\underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq q\,,\,\mathtt{n} \triangleright [x]} [y](\mathtt{n}?q \mid \underline{x} \bullet o!v \mid q \bullet o?y)$$

The obtained service can further perform the following transition:

$$[y](\mathtt{n}?q \mid \underline{x} \bullet o!v \mid q \bullet o?y) \xrightarrow{\underline{x} = q\,,\,q \bullet o\,\emptyset\,1\,v} \mathtt{n}?q$$

Condition $\underline{x} = q$ of this transition contradicts condition $\underline{x} \neq q$ of the previous one, but the service can however evolve. Instead, by using constrained services, we would have:

$$\begin{aligned} &\mathit{true} \vdash [x,y](\mathtt{n}?q \mid \mathtt{n}?x \mid x \bullet o!v \mid q \bullet o?y) \succ\!\xrightarrow{\underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq q\,,\,\mathtt{n} \triangleright [x]} \\ &\underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq q \vdash [y](\mathtt{n}?q \mid \underline{x} \bullet o!v \mid q \bullet o?y) \succ\!\xrightarrow{\underline{x} = q \wedge \underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq q\,,\,q \bullet o\,\emptyset\,1\,v} \mathit{false} \vdash \mathtt{n}?q \end{aligned}$$

because $\underline{x} = q \wedge \underline{x} \neq q$ holds *false*, and the second transition could not be performed. That's why we use constrained services.

Evaluation function, condition $x \notin \mathbf{uv}$ and assumption on bound variables

Consider the service $s \triangleq [y,z](\mathtt{n}!(5 + \underline{x}) \mid \mathtt{n}?y.s' \mid \mathtt{m}?z.\mathtt{m}'!\underline{z}')$, where $\mathtt{n} \neq \mathtt{m} \neq \mathtt{m}'$. If $[\![5 + \underline{x}]\!] = ((\underline{r} \neq \mathbf{bn} \wedge r \notin \mathbf{uv} \wedge \underline{r} = 5 + \underline{x}), \underline{r})$ then

$$\mathtt{n}!(5 + \underline{x}) \xrightarrow{(\underline{r} \neq \mathbf{bn} \wedge r \notin \mathbf{uv} \wedge \underline{r} = 5 + \underline{x})\,,\,\mathtt{n} \triangleleft \underline{r}} \mathbf{0}$$

Therefore, the constrained service $\underline{x} \neq \mathbf{bn} \wedge \underline{x}' \neq \mathbf{bn} \wedge \underline{z}' \neq \mathbf{bn} \vdash s$ can evolve as follows:

$$\underline{x} \neq \mathbf{bn} \wedge \underline{x}' \neq \mathbf{bn} \wedge \underline{z}' \neq \mathbf{bn} \vdash s \succ\!\xrightarrow{\Phi'\,,\,\mathtt{n}\,\emptyset\,1\,\underline{r}} \Phi' \vdash \underbrace{[z](s' \cdot \{y \mapsto \underline{r}\} \mid \mathtt{m}?z.\mathtt{m}'!\underline{z}')}_{s''}$$

for $\Phi' = \mathcal{B}((\underline{x} \neq \mathbf{bn} \wedge \underline{x}' \neq \mathbf{bn} \wedge \underline{z}' \neq \mathbf{bn} \wedge \underline{r} \neq \mathbf{bn} \wedge r \notin \mathbf{uv} \wedge \underline{r} = 5 + \underline{x}), s'', \{x, x', z'\}) = (\underline{x} \neq \mathbf{bn} \wedge \underline{x}' \neq \mathbf{bn} \wedge \underline{z}' \neq \mathbf{bn} \wedge \underline{r} \neq \mathbf{bn} \wedge r \notin \{x, x', z'\} \wedge \underline{r} = 5 + \underline{x})$. Now, we cannot alpha-convert variable z into r, because we would violate the assumption that bound variables differ from variables corresponding to unknown values (in this case, variable z must be different from r because $\underline{r}$ is an unknown value occurring in the constrained service). Similarly, if $[\![5 + \underline{x}]\!] = ((\underline{z} \neq \mathbf{bn} \wedge z \notin \mathbf{uv} \wedge \underline{z} = 5 + \underline{x}), \underline{z})$, then the constrained service could become

$$\Phi'' \vdash [z](s' \cdot \{y \mapsto \underline{z}\} \mid \mathtt{m}?z.\mathtt{m}'!\underline{z}')$$

for some Φ'', but the assumption would be violated again (because the service contains both z and $\underline{z}$). Finally, if $[\![5 + \underline{x}]\!] = ((\underline{z}' \neq \mathbf{bn} \wedge z' \notin \mathbf{uv} \wedge \underline{z}' = 5 + \underline{x}), \underline{z}')$, i.e. the unknown value returned by the evaluation function is not fresh, then the condition on the symbolic transition holds *false*, because $z' \notin \{x, x', z'\}$ does not hold.

5 Extensions of the symbolic operational semantics

In this section, we present two extensions of COWS symbolic semantics for dealing with open terms and polyadic communication.

5.1 Symbolic semantics for open terms

The symbolic operational semantics presented in Section 4 is defined only for closed terms. Indeed, for a reduction semantics it is reasonable that well-formed services may not contain free variables and labels. However, in order to be able to inspect also the behaviour of a part of a service, we need to define the semantics also for open terms.

For example, let us consider the following open term:

$$\mathtt{n}?x \mid \mathtt{n}!x$$

The term can only perform the receive activity $\mathtt{n}?x$ (by communicating with the environment), because activity $\mathtt{n}!x$ is stuck until variable x is not replaced by a value. However, since the scope of the variable is not declared in the term, the environment can substitute the variable with an unknown value in any moment. The resulting term is as follows:

$$\mathtt{n}?\underline{x} \mid \mathtt{n}!\underline{x}$$

Now, the term can perform also the activity $\mathtt{n}!\underline{x}$ (by communicating with the environment) and the internal communication (activities $\mathtt{n}?\underline{x}$ and $\mathtt{n}!\underline{x}$ synchronize).

Formally, the symbolic operational semantics for open terms is defined by the rules in Table 6 and the new rules in Table 8, where the transition label $\underline{x}$ represents execution of a substitution by the environment. We denote by $\mathrm{fv}(t)$ the set of variables in t, and we exploit a predicate noKill(_), a slightly modified variant of that defined in Section 3, whose most significant case is $\mathrm{noKill}(\mathbf{kill}(k)) = \mathrm{false}$ (this way, the predicate holds true if there are not free kill activities that can be immediately performed). Notably, rules *(constServ)*, *(constServ$_{exp}$)* and *(constServ$_{inv}$)* differ from that shown in Table 7 for the addition of the predicate noKill(s) to their premises. The presence of this predicate in the rules of Table 8 guarantees the eager execution of unbounded kill activities. Indeed, for instance, the open term $(\mathbf{kill}(k) \mid \mathtt{n}?v)$ can only evolve as follows (rule *(constServ$_{kill}$)*):

$$\mathit{true} \vdash (\mathbf{kill}(k) \mid \mathtt{n}?v) \xrightarrow{\mathit{true}\,,\,k} \mathit{true} \vdash \mathbf{0}$$

We explain how the remaining rules work by means of some examples. By applying rule *(constServ$_{rec}$)*, the term $(\mathtt{n}?x \mid \mathtt{n}!x)$ can communicate with the environment (by receiving an unknown value) and evolve as follows:

$$\mathit{true} \vdash (\mathtt{n}?x \mid \mathtt{n}!x) \xrightarrow{\underline{x} \neq \mathbf{bn}\,,\,\mathtt{n} \rhd x} \underline{x} \neq \mathbf{bn} \vdash \mathtt{n}!\underline{x}$$

Notably, variable x is replaced by an unknown value, thus now the invoke activity can be performed. By applying rule *(constServ$_{sub}$)*, the same term becomes closed:

$$\mathit{true} \vdash (\mathtt{n}?x \mid \mathtt{n}!x) \xrightarrow{\underline{x} \neq \mathbf{bn}\,,\,\underline{x}} \underline{x} \neq \mathbf{bn} \vdash (\mathtt{n}?\underline{x} \mid \mathtt{n}!\underline{x})$$

$$\frac{s \xrightarrow{\Phi', \alpha} s' \qquad \alpha = \mathtt{n} \rhd \underline{v},\ \mathtt{n} \rhd [x]\ ,\ \mathtt{n}\, \emptyset\, \ell\, \underline{v} \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi)) \qquad \mathrm{noKill}(s)}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \alpha} \Phi'', \Delta \vdash s'} \textit{(constServ)}$$

$$\frac{s \xrightarrow{\Phi', \underline{\mathtt{n}} \lhd [n]} s' \qquad \underline{\mathtt{n}} \notin \Delta \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi)) \qquad \mathrm{noKill}(s)}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \underline{\mathtt{n}} \lhd [n]} \Phi'', \Delta \cup \{n\} \vdash s'} \textit{(constServ}_{exp}\textit{)}$$

$$\frac{s \xrightarrow{\Phi', \underline{\mathtt{n}} \lhd \underline{v}} s' \qquad \underline{\mathtt{n}} \notin \Delta \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi)) \qquad \mathrm{noKill}(s)}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \underline{\mathtt{n}} \lhd \underline{v}} \Phi'', \Delta \vdash s'} \textit{(constServ}_{inv}\textit{)}$$

$$\frac{s \xrightarrow{\Phi', \alpha} s' \qquad \alpha = k, \dagger \qquad \Phi'' = \mathcal{B}(\Phi \wedge \Phi', s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \alpha} \Phi'', \Delta \vdash s'} \textit{(constServ}_{kill}\textit{)}$$

$$\frac{x \in \mathrm{fv}(s) \qquad \Phi' = \mathcal{B}((\Phi \wedge \underline{x} \neq \mathbf{bn}), s, \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi', \underline{x}} \Phi', \Delta \vdash s \cdot \{x \mapsto \underline{x}\}} \textit{(constServ}_{sub}\textit{)}$$

$$\frac{s \xrightarrow{\Phi', \mathtt{n} \rhd x} s' \qquad \mathrm{noKill}(s) \qquad \Phi'' = \mathcal{B}((\Phi \wedge \Phi' \wedge \underline{x} \neq \mathbf{bn} \wedge \underline{x} \neq \mathit{confRec}(s, \mathtt{n})), s', \mathrm{uvar}(\Phi))}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \mathtt{n} \rhd x} \Phi'', \Delta \vdash s' \cdot \{x \mapsto \underline{x}\}} \textit{(constServ}_{rec}\textit{)}$$

$$\frac{s \xrightarrow{\Phi', \mathtt{n}\{x \mapsto \underline{v}\}\, 1\, \underline{v}} s' \qquad \Phi'' = \mathcal{B}((\Phi \wedge \Phi'), s', \mathrm{uvar}(\Phi)) \qquad \mathrm{noKill}(s)}{\Phi, \Delta \vdash s \succ\!\!\xrightarrow{\Phi'', \mathtt{n}\{x \mapsto \underline{v}\}\, 1\, \underline{v}} \Phi'', \Delta \vdash s' \cdot \{x \mapsto \underline{v}\}} \textit{(constServ}_{com}\textit{)}$$

Table 8
Symbolic semantics for COWS open terms

Now, both receive and invoke activities can communicate with the environment and also internal communication can take place. Finally, if we slightly modify the term as $(\mathtt{n}?x \mid \mathtt{n}!v \mid s)$, by applying rule *(constServ$_{com}$)*, we obtain the following transition:

$$\mathit{true} \vdash (\mathtt{n}?x \mid \mathtt{n}!v \mid s) \succ\!\!\xrightarrow{\mathit{true}, \mathtt{n}\{x \mapsto v\}\, 1\, v} \mathit{true} \vdash s \cdot \{x \mapsto v\}$$

Also in this case the substitution for x is applied to the whole term.

5.2 *Symbolic semantics for* COWS *with polyadic communication*

We now tailor COWS syntax and symbolic semantics to deal with polyadic communication. We first extend the syntax of invoke and receive activities as follows: $\underline{u} \bullet \underline{u'}!\overline{e}$ stands for an invoke over the endpoint $\underline{u} \bullet \underline{u'}$ with parameter the tuple of expressions $\overline{e}$, while $p \bullet o?\overline{\underline{w}}.s$ stands for a receive over the endpoint $p \bullet o$ with parameter the tuple of variables/(unknown) values $\overline{\underline{w}}$ and continuation s. Tuples can be constructed using a concatenation operator

$$\mathrm{n}?\overline{\underline{w}}.s \xrightarrow{true\,,\,\mathrm{n}\rhd\overline{\underline{w}}} s \quad \textit{(s-rec)} \qquad \frac{\mathrm{v}(\overline{\underline{w}}) = \overline{x} \qquad |\overline{x}| \geqslant 1}{\mathrm{n}?\overline{\underline{w}}.s \xrightarrow{\overline{\underline{x}}\neq\mathbf{bn}\,,\,\mathrm{n}\rhd[\overline{x}]\overline{\underline{w}}} s} \quad \textit{(s-rec}_{com}\textit{)}$$

$$\frac{s \xrightarrow{\Phi\,,\,\mathrm{n}\rhd[\overline{x}]\overline{\underline{w}}} s' \qquad y \in \overline{x}}{[y]\,s \xrightarrow{\Phi\,,\,\mathrm{n}\rhd[\overline{x}]\overline{\underline{w}}} s' \cdot \{y \mapsto \underline{y}\}} \quad \textit{(s-del}_{sub1}\textit{)} \qquad \frac{[\![e_1]\!] = (\Phi_1, \underline{v}_1) \quad \ldots \quad [\![e_n]\!] = (\Phi_n, \underline{v}_n)}{\underline{\mathrm{n}}!\langle e_1, \ldots, e_n\rangle \xrightarrow{\Phi_1 \wedge \ldots \wedge \Phi_n\,,\,\underline{\mathrm{n}}\lhd\langle \underline{v}_1, \ldots, \underline{v}_n\rangle} \mathbf{0}} \quad \textit{(s-inv)}$$

$$\frac{s \xrightarrow{\Phi\,,\,\underline{\mathrm{n}}\lhd\overline{\underline{v}}} s' \quad n \in \overline{\underline{v}} \quad n \notin \underline{\mathrm{n}}}{[n]\,s \xrightarrow{\Phi\,,\,\underline{\mathrm{n}}\lhd[n]\overline{\underline{v}}} s'} \quad \textit{(s-open}_{1}\textit{)} \qquad \frac{s \xrightarrow{\Phi\,,\,\underline{\mathrm{n}}\lhd[\overline{m}]\overline{\underline{v}}} s' \quad n \in \overline{\underline{v}} \quad n \notin \underline{\mathrm{n}}}{[n]\,s \xrightarrow{\Phi\,,\,\underline{\mathrm{n}}\lhd[\langle n\rangle:\overline{m}]\overline{\underline{v}}} s'} \quad \textit{(s-open}_{2}\textit{)}$$

$$\frac{s_1 \xrightarrow{\Phi_1\,,\,\mathrm{n}\rhd\overline{\underline{w}}} s_1' \quad s_2 \xrightarrow{\Phi_2\,,\,\underline{\mathrm{n}}'\lhd\overline{\underline{v}}} s_2' \quad \mathcal{M}(\overline{\underline{w}}, \overline{\underline{v}}) = (\Phi, \sigma) \quad \mathrm{noConf}(s_1 \mid s_2, \mathrm{n}, \overline{\underline{v}}, |\sigma|) = \Phi'}{s_1 \mid s_2 \xrightarrow{\Phi_1 \wedge \Phi_2 \wedge \mathrm{n}=\underline{\mathrm{n}}' \wedge \Phi \wedge \Phi'\,,\,\mathrm{n}\sigma|\sigma|\overline{\underline{v}}} s_1' \mid s_2'} \quad \textit{(s-com)}$$

$$\frac{s \xrightarrow{\Phi\,,\,\mathrm{n}\sigma\uplus\{x\mapsto\underline{v}\}\,\ell\,\overline{\underline{v}}} s'}{[x]\,s \xrightarrow{\Phi\,,\,\mathrm{n}\sigma\,\ell\,\overline{\underline{v}}} s' \cdot \{x \mapsto \underline{v}\}} \quad \textit{(s-del}_{sub2}\textit{)} \qquad \frac{s_1 \xrightarrow{\Phi\,,\,\alpha} s_1' \quad \alpha \neq k\,,\ \mathrm{n} \rhd [\overline{x}]\overline{\underline{w}}\,,\ \mathrm{n}\sigma\,\ell\,\overline{\underline{v}}}{s_1 \mid s_2 \xrightarrow{\Phi\,,\,\alpha} s_1' \mid s_2} \quad \textit{(s-par}_{pass}\textit{)}$$

$$\frac{s_1 \xrightarrow{\Phi\,,\,\mathrm{n}\sigma\,\ell\,\overline{\underline{v}}} s_1' \qquad \mathrm{noConf}(s_2, \mathrm{n}, \overline{\underline{v}}, \ell) = \Phi'}{s_1 \mid s_2 \xrightarrow{\Phi \wedge \Phi'\,,\,\mathrm{n}\sigma\,\ell\,\overline{\underline{v}}} s_1' \mid s_2} \quad \textit{(s-par}_{conf1}\textit{)}$$

$$\frac{s_1 \xrightarrow{\Phi\,,\,\mathrm{n}\rhd[\overline{x}]\overline{\underline{w}}} s_1' \qquad \mathrm{noConf}(s_2, \mathrm{n}, \overline{\underline{w}} \cdot \{\overline{x} \mapsto \overline{\underline{x}}\}, |\overline{x}|) = \Phi'}{s_1 \mid s_2 \xrightarrow{\Phi \wedge \Phi'\,,\,\mathrm{n}\rhd[\overline{x}]\overline{\underline{w}}} s_1' \mid s_2} \quad \textit{(s-par}_{conf2}\textit{)}$$

Table 9
Operational semantics of COWS with polyadic communication (excerpt of rules)

defined as $\langle a_1, \ldots, a_n\rangle : \langle b_1, \ldots, b_m\rangle = \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$. To single out an element of a tuple, we will write $(\overline{a}, c, \overline{b})$ to denote the tuple $\langle a_1, \ldots, a_n, c, b_1, \ldots, b_m\rangle$, where $\overline{a}$ or $\overline{b}$ might not be present. We will use $\overline{a}_i$ to denote the i-th element of the tuple $\overline{a}$ and, when convenient, we shall regard a tuple simply as a set writing e.g. $a \in \overline{b}$ to mean that a is an element of $\overline{b}$. Finally, we denote by $\mathrm{v}(t)$ the set of variables in t.

The labelled transition relation $\xrightarrow{\Phi\,,\,\alpha}$ over services now is induced by the modified rules shown in Table 9 (the remaining ones are those of Table 6, except for rule *(s-match)* which we do not need anymore), where:

- conditions can also have the form $\Phi \vee \Phi'$; we will use $\overline{\underline{x}} \neq \mathbf{bn}$ to denote condition $\underline{x}_1 \neq \mathbf{bn} \wedge \ldots \wedge \underline{x}_n \neq \mathbf{bn}$ for $\overline{\underline{x}} = \langle \underline{x}_1, \ldots, \underline{x}_n\rangle$;
- action labels are generated by the following grammar:

$$\alpha \;::=\; \underline{\mathrm{n}} \lhd \overline{\underline{v}} \;\mid\; \underline{\mathrm{n}} \lhd [\overline{n}]\overline{\underline{v}} \;\mid\; \mathrm{n} \rhd \overline{\underline{w}} \;\mid\; \mathrm{n} \rhd [\overline{x}]\overline{\underline{w}} \;\mid\; \mathrm{n}\sigma\,\ell\,\overline{\underline{v}} \;\mid\; k \;\mid\; \dagger$$

All the above definitions shall extend to relation $\succ\!\xrightarrow{\Phi\,,\,\alpha}$.

The new rules exploit a modified version of functions $\mathcal{M}(\_,\_)$ and $\mathrm{noConf}(\_,\_,\_,\_)$ defined in Tables 3 and 4, now redefined by the rules in Table 10. The rules in the upper part of the table state that variables match any value, and two values $\underline{v}$ and $\underline{v}'$ do match only

$$\mathcal{M}(x, \underline{v}) = (true, \{x \mapsto \underline{v}\}) \qquad \mathcal{M}(\underline{v}, \underline{v}') = (\underline{v} = \underline{v}', \emptyset) \qquad \mathcal{M}(\langle\rangle, \langle\rangle) = (true, \emptyset)$$

$$\frac{\mathcal{M}(a_1, b_1) = (\Phi_1, \sigma_1) \quad \mathcal{M}(\overline{a}_2, \overline{b}_2) = (\Phi_2, \sigma_2)}{\mathcal{M}((a_1, \overline{a}_2), (b_1, \overline{b}_2)) = (\Phi_1 \wedge \Phi_2, \sigma_1 \uplus \sigma_2)}$$

$$\mathrm{noConf}(s, \mathrm{n}, \underline{\overline{v}}, \ell) = \bigwedge_{\underline{\overline{w}} \in rec(s,\mathrm{n},\underline{\overline{v}},\ell)} \Big(\bigvee_{(\underline{x},i) \in gval(\underline{\overline{w}})} \underline{x} \neq \underline{\overline{v}}_i \ \wedge \ (gval(\underline{\overline{w}}) = \emptyset \Rightarrow false) \Big)$$

$$rec(\mathrm{n}?\underline{\overline{w}}.s, \mathrm{n}, \underline{\overline{v}}, \ell) = \begin{cases} \{\underline{\overline{w}}\} & \text{if } \mathcal{M}(\underline{\overline{w}}, \underline{\overline{v}}) = (\Phi, \sigma) \wedge |\sigma| < \ell \\ \emptyset & \text{otherwise} \end{cases}$$

$$rec(\mathbf{0}, \mathrm{n}, \underline{\overline{v}}, \ell) = rec(\mathbf{kill}(k), \mathrm{n}, \underline{\overline{v}}, \ell) = rec(\mathrm{u}!\overline{e}, \mathrm{n}, \underline{\overline{v}}, \ell) = \emptyset \qquad rec(\mathrm{n}'?\underline{\overline{w}}.s, \mathrm{n}, \underline{\overline{v}}, \ell) = \emptyset \ \text{ if } \mathrm{n} \neq \mathrm{n}'$$

$$rec([d]\, s, \mathrm{n}, \underline{\overline{v}}, \ell) = rec(s, \mathrm{n}, \underline{\overline{v}}, \ell) \ \text{ if } d \notin \mathrm{n} \qquad rec([d]\, s, \mathrm{n}, \underline{\overline{v}}, \ell) = \emptyset \ \text{ if } d \in \mathrm{n}$$

$$rec(g + g', \mathrm{n}, \underline{\overline{v}}, \ell) = rec(g, \mathrm{n}, \underline{\overline{v}}, \ell) \cup rec(g', \mathrm{n}, \underline{\overline{v}}, \ell) \qquad rec(\{\!|s|\!\}, \mathrm{n}, \underline{\overline{v}}, \ell) = rec(s, \mathrm{n}, \underline{\overline{v}}, \ell)$$

$$rec(s \mid s', \mathrm{n}, \underline{\overline{v}}, \ell) = rec(s, \mathrm{n}, \underline{\overline{v}}, \ell) \cup rec(s', \mathrm{n}, \underline{\overline{v}}, \ell) \qquad rec(* s, \mathrm{n}, \underline{\overline{v}}, \ell) = rec(s, \mathrm{n}, \underline{\overline{v}}, \ell)$$

Table 10
Modified matching and conflicting receives rules

if condition $\underline{v} = \underline{v}'$ holds. When tuples $\underline{\overline{w}}$ and $\underline{\overline{v}}$ do match, $\mathcal{M}(\underline{\overline{w}}, \underline{\overline{v}})$ returns a pair (Φ, σ), where Φ is the condition so that the matching holds, and σ is a substitution for the variables in $\underline{\overline{w}}$; otherwise, it is undefined. Function $\mathrm{noConf}(s, \mathrm{n}, \underline{\overline{v}}, \ell)$ now returns a condition that guarantees absence of conflicts for the inferred transition. Basically, $\mathrm{noConf}(s, \mathrm{n}, \underline{\overline{v}}, \ell)$ exploits function $rec(s, \mathrm{n}, \underline{\overline{v}}, \ell)$ to identify the conflicting receives of s, then for each arguments $\underline{\overline{w}}$ of these receives it determines a condition (i.e. a logical disjunction of inequalities) that makes the conflicting matching between $\underline{\overline{w}}$ and $\underline{\overline{v}}$ false. Finally, it returns the logical conjunction of the determined conditions. We use the auxiliary function $gval(\_)$ that, given a tuple $\underline{\overline{w}}$, returns a collection of pairs of the form $(\underline{x}, i)$, where $\underline{x}$ is an unknown value such that $\underline{\overline{w}}_i = \underline{x}$. Notably, if $rec(s, \mathrm{n}, \underline{\overline{v}}, \ell) = \emptyset$ then function $\mathrm{noConf}(s, \mathrm{n}, \underline{\overline{v}}, \ell)$ returns the condition *true*, because there are not conflicting receives; while, if there is a $\underline{\overline{w}} \in rec(s, \mathrm{n}, \underline{\overline{v}}, \ell)$ such that $gval(\underline{\overline{w}}) = \emptyset$, then the function returns the condition *false*, because there are not conditions that can make the conflicting matching false.

We end this section with an example aimed at clarifying how pattern-matching and conflict checking functions work. Consider the following term:

$$\mathrm{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\, \mathrm{n}?\langle x, y, z\rangle \mid [x']\, \mathrm{n}?\langle x', \underline{y}', \underline{z}'\rangle \mid [x'', y'']\, \mathrm{n}?\langle x'', y'', \underline{z}''\rangle$$

In this case, the invoke activity $\mathrm{n}!\langle v_1, v_2, v_3\rangle$ can synchronize with each receive activity of the term. Firstly, consider the receive $\mathrm{n}?\langle x, y, z\rangle$: its argument $\langle x, y, z\rangle$ matches the tuple $\langle v_1, v_2, v_3\rangle$ by generating the substitution $\{x \mapsto v_1, y \mapsto v_2, z \mapsto v_3\}$. The other two receive activities are in conflict, because they satisfy the matching with the invoke and generate substitutions with fewer pairs than 3. Thus, function $rec(\_, \_, \_, \_)$ applied to the whole term [4] returns the set $\{\langle x', \underline{y}', \underline{z}'\rangle, \langle x'', y'', \underline{z}''\rangle\}$. Then, function $\mathrm{noConf}(\_, \_, \_, \_)$ returns the

[4] This means that the last rule applied in the inference is *(s-com)*. Of course, the last rule could be also *(s-par$_{conf\,1}$)*; in this case, two or three conflict checks will be performed on subterms of the considered service.

condition $(\underline{y'} \neq v_2 \vee \underline{z'} \neq v_3) \wedge \underline{z''} \neq v_3$. Hence, a transition of the term is

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\,\mathtt{n}?\langle x, y, z\rangle \mid [x']\,\mathtt{n}?\langle x', \underline{y'}, \underline{z'}\rangle \mid [x'', y'']\,\mathtt{n}?\langle x'', y'', \underline{z''}\rangle$$
$$\xrightarrow{(\underline{y'}\neq v_2 \vee \underline{z'}\neq v_3) \wedge \underline{z''}\neq v_3\,,\,\mathtt{n}\,@\,3\,\langle v_1,v_2,v_3\rangle} [x']\,\mathtt{n}?\langle x', \underline{y'}, \underline{z'}\rangle \mid [x'', y'']\,\mathtt{n}?\langle x'', y'', \underline{z''}\rangle$$

Consider now the receive $\mathtt{n}?\langle x'', y'', \underline{z''}\rangle$: in this case the matching function returns condition $\underline{z''} = v_3$ and substitution $\{x'' \mapsto v_1, y'' \mapsto v_2\}$. Function $rec(\_,\_,\_,\_)$ applied to the whole term returns the set $\{\langle x', \underline{y'}, \underline{z'}\rangle\}$, because the only conflicting receive is $\mathtt{n}?\langle x', \underline{y'}, \underline{z'}\rangle$. Thus, the corresponding transition is

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\,\mathtt{n}?\langle x, y, z\rangle \mid [x']\,\mathtt{n}?\langle x', \underline{y'}, \underline{z'}\rangle \mid [x'', y'']\,\mathtt{n}?\langle x'', y'', \underline{z''}\rangle$$
$$\xrightarrow{(\underline{y'}\neq v_2 \vee \underline{z'}\neq v_3) \wedge \underline{z''}= v_3\,,\,\mathtt{n}\,@\,2\,\langle v_1,v_2,v_3\rangle} [x, y, z]\,\mathtt{n}?\langle x, y, z\rangle \mid [x']\,\mathtt{n}?\langle x', v_2, \underline{z'}\rangle$$

Moreover, the receive activities can communicate with the environment; in this case the conflict checks are performed by rule *(s-par$_{conf\,2}$)*. For example, the transition corresponding to the execution of $\mathtt{n}?\langle x, y, z\rangle$ is

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\,\mathtt{n}?\langle x, y, z\rangle \mid [x']\,\mathtt{n}?\langle x', \underline{y'}, \underline{z'}\rangle \mid [x'', y'']\,\mathtt{n}?\langle x'', y'', \underline{z''}\rangle$$
$$\xrightarrow{\underline{x}\neq \mathbf{bn} \wedge \underline{y}\neq \mathbf{bn} \wedge \underline{z}\neq \mathbf{bn} \wedge (\underline{y'}\neq \underline{y} \vee \underline{z'}\neq \underline{z}) \wedge \underline{z''}\neq \underline{z}\,,\,\mathtt{n} \rhd [\langle x,y,z\rangle]\,\langle x,y,z\rangle}$$
$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x']\,\mathtt{n}?\langle x', \underline{y'}, \underline{z'}\rangle \mid [x'', y'']\,\mathtt{n}?\langle x'', y'', \underline{z''}\rangle$$

Finally, as another example consider the following term:

$$\mathtt{n}!\langle v_1, v_2, v_3\rangle \mid [x, y, z]\,\mathtt{n}?\langle x, y, z\rangle \mid [x']\,\mathtt{n}?\langle x', v_2, v_3\rangle$$

If we try to infer the transition corresponding to the communication with $\mathtt{n}?\langle x, y, z\rangle$, we have that the condition on the transition label is *false*, because function $rec(\_,\_,\_,\_)$ returns $\langle x', v_2, v_3\rangle$ and $gval(\langle x', v_2, v_3\rangle) = \emptyset$.

6 Related work and concluding remarks

Symbolic semantics and symbolic bisimulation were first introduced in [13] by Hennessy and Lin on value-passing process algebras. The symbolic approach has been then applied to π-calculus in [24] by Sangiorgi and in [4] by Boreale and De Nicola. Victor has adopted a similar approach in [25] to efficiently characterise hyperequivalence for the fusion calculus. A more recent work on a symbolic semantics for a fusion-based calculus is [6] by Buscemi and Montanari. A revisited symbolic technique for π-calculus has been recently proposed in [2] by Bonchi and Montanari.

COWS is a process calculus introduced in [16] for specifying and combining service-oriented applications, while modelling their dynamic behaviour. Since its definition, a number of methods and tools have been devised to analyse COWS specifications, such as a type system to check confidentiality properties [17], a logic and a model checker to express and check functional properties of services [11], a stochastic extension to enable quantitative reasoning on service behaviours [22], a static analysis to establish properties of the

flow of information between services [1], and bisimulation-based observational semantics to check interchangeability of services and conformance against service specifications [23]. An overview of some of the above tools, with an application to the analysis of a case study, can be found in [18].

We believe that the alternative symbolic operational semantics defined in this paper can pave the way for the development of efficient model and equivalence checkers for COWS. In fact, the model checking approach of [11] does not support a fully compositional verification methodology. It allows to analyse systems of services 'as a whole', but does not enable analysis of services in isolation (e.g. a provider service without a proper client). The symbolic operational semantics should permit to overcome this limitation that is somewhat related to the original semantics of COWS which, although based on an LTS, follows a reduction style. Furthermore, the symbolic operational semantics can be used to improve efficiency of checking the equivalences introduced in [23]. This, of course, requires defining alternative characterizations of the equivalences on top of the symbolic transition system. We plan to pursue these lines of research in the near future, and in particular to implement the operational semantics and equivalence and model checkers on top of it.

References

[1] J. Bauer, F. Nielson, H.R. Nielson, and H. Pilegaard. Relational analysis of correlation. In María Alpuente and Germán Vidal, editors, *The 15th International Static Analysis Symposium (SAS'08)*, volume 5079 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2008.

[2] F. Bonchi and U. Montanari. Symbolic semantics revisited. In *FoSSaCS*, volume 4962 of *LNCS*, pages 395–412. Springer, 2008.

[3] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In G. Barthe and F.S. de Boer, editors, *FMOODS*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.

[4] M. Boreale and R. De Nicola. A symbolic semantics for the pi-calculus. *Inf. Comput.*, 126(1):34–52, 1996.

[5] R. Bruni, I. Lanese, H.C. Melgratti, and E. Tuosto. Multiparty sessions in SOC. In D. Lea and G. Zavattaro, editors, *COORDINATION*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008.

[6] M. Buscemi and U. Montanari. Open Bisimulation for the concurrent Constraint Pi-Calculus. In *ESOP*, volume 4960 of *LNCS*, pages 254–268. Springer, 2008.

[7] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION*, volume 4038 of *LNCS*, pages 63–81. Springer, 2006.

[8] M.J. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, volume 3525 of *LNCS*, pages 133–150. Springer, 2005.

[9] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[10] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.

[11] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *FASE*, volume 4961 of *LNCS*, pages 230–245. Springer, 2008.

[12] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.

[13] M. Hennessy and H. Lin. Symbolic bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995.

[14] I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Int. Conf. on Software Engineering and Formal Methods (SEFM'07)*, pages 305–314. IEEE Computer Society, 2007.

[15] C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

[16] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. Technical report, DSI, Univ. Firenze, 2007. Available at http://rap.dsi.unifi.it/cows/papers/cows-esop07-full.pdf. An extended abstract appeared in *Proc. of ESOP'07*, LNCS 4421, pages 33-47, Springer.

[17] A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN*, volume 4767 of *LNCS*, pages 223–239. Springer, 2007.

[18] A. Lapadula, R. Pugliese, and F. Tiezzi. Specifying and analysing soc applications with cows. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 701–720. Springer, 2008.

[19] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.

[21] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007. Available at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[22] D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC*, volume 4749 of *LNCS*, pages 245–256. Springer, 2007.

[23] R. Pugliese, F. Tiezzi, and N. Yoshida. On observing dynamic prioritised actions in SOC. Technical report, DSI, Univ. Firenze, 2008. Available at http:://rap.dsi.unifi.it/cows/bis4cows-full.pdf.

[24] D. Sangiorgi. A theory of bisimulation for the pi-calculus. *Acta Inf.*, 33(1):69–97, 1996.

[25] B. Victor. Symbolic Characterizations and Algorithms for Hyperequivalence. Technical Report DoCS 98/96, Uppsala University, December 1998.

[26] H.T. Vieira, L. Caires, and J. Costa Seco. The conversation calculus: A model of service-oriented computation. In S. Drossopoulou, editor, *Programming Languages and Systems (ESOP'08)*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.