A WSDL-based type system for WS-BPEL*

Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi

Dipartimento di Sistemi e Informatica Università degli Studi di Firenze

Abstract. We tackle the problem of providing rigorous formal foundations to current software engineering technologies for web services. We focus on two of the most used XML-based languages for web services: WSDL and WS-BPEL. To this aim, first we select an expressive subset of WS-BPEL, with special concern for modeling the interactions among web service instances in a network context, and define its operational semantics. We call ws-calculus the resulting formalism. Then, we put forward a rigorous typing discipline that formalizes the relationship existing between ws-calculus terms and the associated WSDL documents and supports verification of their compliance. We prove that the type system and the operational semantics of ws-calculus are 'sound' and apply our approach to an example application involving three interacting web services.

1 Introduction

Service-Oriented Computing (SOC) has recently put forward as a promising computing paradigm for developing massively distributed, interoperable, evolvable systems and applications that exploit the pervasiveness of the Internet and its related technologies. The SOC paradigm advocates the use of 'services', to be understood as autonomous, platform-independent computational entities that can be described, published, discovered, and dynamically assembled, as the basic blocks for building applications. Web services (WS), along with grid computing, are the present most successful instantiation of the SOC paradigm, as it is demonstrated by the fact that companies like IBM, Microsoft and Sun invested a lot of efforts and resources to promote their deployment.

A *web service* is basically a set of operations that can be invoked through the Web via XML messages complying with given standard formats. To support the WS approach, many new languages, most of which based on XML, have been designed, like e.g. business coordination languages (such as WS-BPEL, WSFL, WSCI, and XLANG), contract languages (such as WSDL and SWS), and query languages (such as XPath and XQuery). However, current software engineering technologies for WS still lack rigorous formal foundations. The challenges come from the necessity of dealing at once with issues like communication, co-operation, resource usage, failures, security, etc. in a setting where demands and guarantees can be very different for the many components.

In this paper we focus on two of the most used XML-based languages for WS: *Web Services Description Language* (WSDL [CCMW01]) and *Web Services Business Process Execution Language* (WS-BPEL [BCG+05]). The former is a W3C standard that

* Supported by EU within the FP6-2004-IST-FET Proactive project SENSORIA proposal contract number 016004.

permits to express the functionalities offered and required by web services by defining, akin object interfaces in Object-Oriented Programming, the signatures of operations and the structure of the documents for invoking them and returned by them. The latter, currently under evaluation to become a standard by OASIS, permits to describe the activities to be executed for completing the service as a reaction to a service invocation. WSDL declarations can be exploited to verify the possibility of connecting different services, while WS-BPEL descriptions can be used to define new services by appropriately composing other existing ones.

We aim at formalizing the relationship existing between WS-BPEL processes and the associated WSDL documents by putting forward a rigorous typing discipline. In general, the WSDL document associated to a WS-BPEL process does not contain the declarations of all the operations provided and required by the process, together with the structure of the messages exchanged. In fact, some of these declarations usually are in the WSDL documents of the orchestrated services. Moreover, WSDL provides four different types of operations, but only two of them are really supported by WS-BPEL: (synchronous) request-response and one-way. There is another interaction pattern that is largely used in WS-BPEL (see, e.g., the example 16.1 in [BCG$^+$05]) but it is not provided by WSDL: asynchronous request-response. This last pattern is implemented through a partner link connecting two one-way operations but no constraint is imposed on which process must declare the type of the operations. Finally, WS-BPEL provides many redundant programming constructs and suggest a quite liberal programming style. For example, it is possible for a programmer to write parallel activities that have strict implicit dependencies so that they are sequentially (rather than concurrently) executed.

To achieve our goal, we first define a semantic model for WS-BPEL because the semantics of the language, as presented in [BCG$^+$05], is informal and, sometimes, ambiguous. Hence, as a first contribution of this paper, we introduce a process language, that we call ws-calculus (*web services calculus*), that formalizes the semantics of an expressive subset of WS-BPEL, with special concern for modeling the interactions among web services, be them WS-BPEL processes or not, in a network context. This allows us to tackle those problems arising when executing WS-BPEL processes, such as multiple start activities, receive conflicts, routing of messages, while avoiding the intricacies of dealing with any, possibly redundant, WS-BPEL construct.

As a second contribution, we define a type system for ws-calculus terms and show that the type system and the operational semantics are 'sound', in the sense that ws-calculus terms reached along any reduction sequence starting from well-typed terms are still well-typed and, thus, do not generate runtime errors. The type system enforces many of the constraints imposed by WSDL/WS-BPEL, e.g. it prevents programs from passing links that have been implicitly initialized and from invoking callback operations that do not have previous triggering receive operations. However, in some cases it is even further restrictive so to enforce a more disciplined programming style. Thus, for example, the type system deems as ill-typed those programs containing flow activities that have strict implicit dependencies. Moreover, in case of asynchronous request-response, it forces the WSDL document associated to the process providing the service to contain the declaration of both the two operations, that for invoking the service and that for sending the reply back to the client. This last choice is dictated by the need to

preserve two important properties of web services, namely compositionality and loose coupling. Indeed, should each client contain the declaration for the reply, then, if the service provider wants to modify such a declaration, all clients should be updated.

The rest of the paper is organized as follows. Syntax and operational semantics of ws-calculus are defined in Section 2, while the type system and the soundness results are presented in Section 3. Section 4 illustrates an application of our framework to modelling an example of web services composition. In Section 5 we touch upon directions for future work and comparisons with related work. We refer the interested reader to the full paper [LPT06] for a complete semantic account of WS-BPEL, for further examples and for the proofs of all results stated in this paper.

2 WS-CALCULUS

ws-calculus (*web services calculus*) permits to express web services in a primitive form with special concern for modeling the interactions among web service instances in a network context. Although ws-calculus can directly model the semantics of an expressive subset of WS-BPEL, we refer the interested reader to [LPT06] for a more complete account of this topic. Indeed, due to lack of space, here we do not deal with many features such as, e.g., flow graphs, timed activities, scopes and compensation handling, that should be considered for modeling the semantics of full-blown orchestration languages.

The *syntax* of ws-calculus, given in Table 1, is parameterized with respect to the following syntactic sets, which we assume to be countable and pairwise disjoint: *properties* (sorts of late bound constants storing some relevant values within service instances, ranged over by p), *basic values* (integers `Int`, strings `Str`, and booleans) and corresponding *variables* (ranged over by b), *addresses* (ranged over by a) and *partner links* (namely variables storing addresses used to identify service partners within an interaction, ranged over by l), and *service identifiers* (ranged over by A) each with a fixed nonnegative arity. The language is also parametric with respect to a set of *operations*, ranged over by o, which we do not specify, and *expressions*, ranged over by e, whose exact syntax is deliberately omitted; we just assume that expressions contain, at least, basic values and variables, partner links, addresses and properties. Notationally, we will use u to range over *values* (i.e. basic values and addresses), v to range over *variables* (i.e. basic variables and partner links), w to range over *operation parameters* (i.e. variables and properties), c to range over *correlation patterns* (i.e. values and properties), and r to range over addresses and partner links. Addresses may be *underlined* to denote that they cannot be transmitted as operation parameters, while partner links may be subject to the operator $\ulcorner \cdot \urcorner$ that forces them to be already initialized.

Notation $\bar{\cdot}$ denotes tuples of objects. E.g. $\bar{v}$ is a tuple of variables; this will sometimes be written as $\bar{v}_{i\in I}$, for an appropriate index-set I, and v_i denotes the i-th element. We assume that variables in the same tuple are pairwise distinct. When convenient, we shall regard a tuple simply as a set writing e.g. $a \in \bar{u}$ to mean that a is an element of $\bar{u}$. All notations shall extend to tuples component-wise.

A ws-calculus *node* can be thought of as a WS-BPEL process web service. Nodes, written as $a ::^{Op,L} C$, are uniquely identified by an address a and have a declarative part

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

Table 1. WS-CALCULUS syntax (The syntax of types Op, L is in Table 4)

Op, L, i.e. its *type*, and a behavioural part C. Finite sets of nodes are called *nets* and are ranged over by $N, N', N_1, \ldots$. The type of a node collects all the information about the format of the messages exchanged by the operations available at the node, Op, and the local declarations, L, like the WSDL document associated to the corresponding WS-BPEL process web service. Since we are interested in describing asynchronous interactions, we model each communication pattern by connecting one or more one-way operations. In the simplest interaction, a single one-way operation suffices; the service provider process, which is the one that performs the receive activity, holds the type definition of the requested operation. The more complex asynchronous request-response interaction pattern is expressed by connecting two one-way operations (request and callback); in this case, the provider holds the type definitions of both operations (the rationale for this choice has been explained in the Introduction). We defer syntax of types and comments on them to Section 3.

Components C may be service specifications, instances or requests. The behavioural specification of a service s is written $*s$, while $m \gg s'$ represents a service instance that behaves according to s' and whose properties evaluate according to the (possibly empty) set m of correlation constraints. A *correlation constraint* is a pair, written $p = u$, recording the value u assigned to the property p. Properties are used to store values that are important to identify service instances. For example, one might use a property named *purchase-order-id* to uniquely identify instances of a service that handles purchase orders. A service request $\langle a, o, \bar{u} \rangle$ represents an operation invocation that must still be processed and contains the invoker address a, the operation name o and the data $\bar{u}$ for operation execution. WS-CALCULUS operation names represent WS-BPEL pairs 'partner link – operation' (instead, WS-BPEL partner links are not explicitly modeled), thus pairs 'a – o', that are the first two components of service requests, represent endpoints between two interacting process web services.

Services are structured activities built from *basic activities*, i.e. instance forced termination **exit**, assignment **ass** $(\cdot,\cdot)$, service invocation **inv** $(\cdot,\cdot,\cdot)$ and service request processing **rec** $(\cdot,\cdot,\cdot)$, by exploiting operators for conditional choice **if** $(\cdot)$ **then** $\{\cdot\}$ **else** $\{\cdot\}$ (*switch*), sequential composition $\cdot;\cdot$ (*sequence*), parallel composition $\cdot \mid \cdot$ (*flow*), external choice[1] $\sum_{i\in I}$ **rec** $(\cdot,\cdot,\cdot);\cdot$ (*pick*) and service call $A(w_1,\cdots,w_n)$ where n is the arity of A. Every service identifier A with arity n has a unique definition of the form $A(\bar{v}_{i\in\{1,..,n\}} : \bar{\tau}^{\star}_{i\in\{1,..,n\}}) \stackrel{def}{=} s$, where the v_i must be fresh and pairwise distinct. Notably, parameters of a service definition are typed (see the next section).

The ws-calculus binding constructs are **ass** $(\bar{w},\bar{e})$ and **rec** $(r,o,\bar{w})$ that bind the variables and the properties in $\bar{w}$. The latter also binds r if it is a partner link and is not subject to the operator $\ulcorner\cdot\urcorner$; we will say that r is *implicitly* initialized (conversely, we will say that a partner link is *explicitly* initialized in all other cases). This means that $\ulcorner l \urcorner$ represents a free occurrence of l (e.g. a callback address) that must have been bound previously. The scope of the bindings extends to the whole component where the binder occurs (namely, like in WS-BPEL, variables and properties are global to the instance). A variable occurrence is free if it is not under the scope of a binder. We assume that all bound partner links are pairwise distinct, but for those occurring within alternative branches of switch and pick constructs. Thus, the following fragment of service is well-defined:

$$\ldots \textbf{if } (e) \textbf{ then } \{\ldots \textbf{rec}\,(l,\ldots,\ldots)\ldots\} \textbf{ else } \{\ldots \textbf{rec}\,(l,\ldots,\ldots)\ldots\}; \textbf{inv}\,(l,\ldots,\ldots)\ldots$$

In general, we use $fv(s)$ (resp. $bv(s)$) to denote the set of variables which occur free (resp. bound) in s. In particular, variables of $\bar{w}$ are free in $A(\bar{w})$. In a definition $A(\bar{v} : \bar{\tau}^{\star}) \stackrel{def}{=} s$ we assume $fv(s) \subseteq \bar{v}$.

In the sequel we shall only consider nets that are *well-formed* in the sense that they comply with the following syntactic constraints. First, pairwise distinct nodes must have different addresses. Then, if we call *start activities* of a service s all those activities that are not syntactically preceded by other ones (as formalized by function $eR()$ whose inductive definition can be found in [LPT06]), then at least one start activity of $*s$ must be a **rec** $(\cdot,\cdot,\cdot)$ and, if multiple **rec** $(\cdot,\cdot,\cdot)$ are enabled concurrently, then they must use the same non-empty set of properties.

The *operational semantics* of ws-calculus is given in terms of a structural congruence and of a reduction relation over nets. Due to space limitations, here we only present the major ingredients and refer the interested reader to [LPT06] for the details. For instance, we omit the rules for fault throwing and handling, and model taking place of errors (e.g. when the premises of reduction rules are not satisfied) simply as deadlock.

The semantics of nets will be defined over an enriched set of nets that also includes those auxiliary nets resulting from replacing (free occurrences of) variables with values in nets produced by the syntax of Table 1. Therefore, we will let free occurrences of v (and w) to also denote corresponding values.

The *structural congruence*, denoted by $\equiv$, identifies syntactically different terms which intuitively represent the same term. At the level of services, the structural congruence states that: the sequence operator is associative and has **0** as identity element

[1] Whenever the external choice is between two activities, we shall simply write $s_1 + s_2$.

(thus we have the law $\mathbf{0}; s \equiv s$, which is exploited to enable a new activity when a syntactically preceding one terminates); the flow operator is commutative, associative and has $\mathbf{0}$ as identity element; the pick operator enjoys the same properties and, additionally, is idempotent; services only differing for the bound variables are the same (*alpha-conversion*). The structural congruence is extended to components and nets in the obvious way. In particular, components composition is commutative and associative, and has $m \gg \mathbf{0}$ as identity element (i.e. instances of this form are terminated instances and, thus, can be removed).

The *reduction relation* over nets, written $\succ\!\!\rightarrow$, relies on a labelled transition relation $\xrightarrow{\alpha}$ over service instances, where α is generated by the following production:

$$\alpha \ ::= \ \natural \ \mid \ \bar{w} := \bar{u} \ \mid \ i(a, o, \bar{u}) \ \mid \ r(r, o, \bar{w})$$

The meaning of labels is as follows: $\natural$ denotes forced termination of a service instance, $\bar{w} := \bar{u}$ denotes execution of a multiple assignment, $i(a, o, \bar{u})$ denotes invocation of operation o located at a with data $\bar{u}$ and $r(r, o, \bar{w})$ denotes launching of o with operation parameters $\bar{w}$ on request of a web service instance located at r.

To define the operational semantics, we exploit a few auxiliary functions. First, we define a function for evaluating expressions: it takes an expression and returns a basic value or an address. We write $m \triangleright e$ such a function, but we do not explicitly define it because the exact syntax of expressions is deliberately not specified (recall that ws-calculus is parameterized wrt the syntax of expressions). Expressions to be evaluated can contain properties; thus, evaluation of e takes place wrt a set of correlation constraints m storing the values of the properties that may occur within e. On the contrary, expressions to be evaluated cannot contain (free) variables because these occurrences are replaced with the corresponding values as soon as the variables are bound. Indeed, execution of a binding construct generates a *substitution* (ranged over by σ), i.e. a map from basic variables to basic values and from partner links to addresses, that is applied to the whole instance where the binder occurs. A substitution σ will be sometimes written as $(\bar{v} \mapsto \sigma(\bar{v}))$ for $\bar{v} = dom(\sigma)$. Application of substitution σ to s is written $s \cdot \sigma$. The effect of $s \cdot \sigma$ is that, for each $x \in dom(\sigma)$, every free occurrence of x in s is replaced with $\sigma(x)$.

Another ingredient we need to define the semantics is a mechanism for checking if the assignments of u_i to w_i, for any index i in a given set I, comply with the correlation constraints in m. We will write $m \triangleright (\bar{w}_{i \in I} := \bar{u}_{i \in I})$ such a mechanism. In case the check succeeds, to take care of the effect of the assignments, a pair $\langle m', \sigma \rangle$ is returned where m' is the set of the correlation constraints for the properties in $\bar{w}_{i \in I}$ and σ is the substitution for the variables in $\bar{w}_{i \in I}$. The function is defined inductively on the syntax of $\bar{w}$ as follows:

$$m \triangleright (v := u) = \langle \emptyset, (v \mapsto u) \rangle$$

$$m \triangleright (p := u) = \begin{cases} \langle \emptyset, \emptyset \rangle & \text{if } p = u \in m \\ \langle \{p = u\}, \emptyset \rangle & \text{if it does not exists } u' \text{ s.t. } p = u' \in m \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$m \triangleright (w, \bar{w} := u, \bar{u}) = \langle m' \cup m'', \sigma \circ \sigma' \rangle \quad \begin{cases} \text{if } m \triangleright (w := u) = \langle m', \sigma \rangle \text{ and} \\ m \cup m' \triangleright (\bar{w} := \bar{u}) = \langle m'', \sigma' \rangle \end{cases}$$

$$m \vdash \mathbf{exit} \xrightarrow{\natural} \mathbf{0} \quad (\mathit{Exit}) \qquad m \vdash \mathbf{ass}\,(\bar{w}, \bar{e}) \xrightarrow{\bar{w} := m \rhd \bar{e}} \mathbf{0} \quad (\mathit{Assign})$$

$$m \vdash \mathbf{inv}\,(a, o, \bar{c}) \xrightarrow{i(a,o,m \rhd \bar{c})} \mathbf{0} \quad (\mathit{Invoke}) \qquad \frac{r \neq \ulcorner l \urcorner}{m \vdash \mathbf{rec}\,(r, o, \bar{w}) \xrightarrow{r(r,o,\bar{w})} \mathbf{0}} \quad (\mathit{Receive})$$

$$\frac{m \rhd e = \mathbf{false} \qquad m \vdash s_2 \xrightarrow{\alpha} s_2'}{m \vdash \mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\} \xrightarrow{\alpha} s_2'} \quad (\mathit{If}_{\mathrm{ff}}) \qquad \frac{m \rhd e = \mathbf{true} \qquad m \vdash s_1 \xrightarrow{\alpha} s_1'}{m \vdash \mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\} \xrightarrow{\alpha} s_1'} \quad (\mathit{If}_{\mathrm{tt}})$$

$$\frac{m \vdash s_1 \xrightarrow{\alpha} s_1'}{m \vdash s_1; s_2 \xrightarrow{\alpha} s_1'; s_2} \quad (\mathit{Sequence}) \qquad \frac{m \vdash s_1 \xrightarrow{\alpha} s_1' \quad \alpha \neq r(\cdot,\cdot,\cdot)}{m \vdash s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2} \quad (\mathit{Flow})$$

$$\frac{m \vdash s_1 \xrightarrow{r(r,o,\bar{w})} s_1' \quad \nexists\, \mathbf{rec}\,(r, o, \bar{w}') \in eR(s_2)\ .\ P(\bar{w}) = P(\bar{w}')}{m \vdash s_1 \mid s_2 \xrightarrow{r(r,o,\bar{w})} s_1' \mid s_2} \quad (\mathit{Flow}_{Rec})$$

$$\frac{m \vdash \mathbf{rec}\,(r, o, \bar{w})\,; s \xrightarrow{r(r,o,\bar{w})} s' \quad \nexists\, i \in I\ .\ r_i = r\ \wedge\ o_i = o\ \wedge\ P(\bar{w}_i) = P(\bar{w})}{m \vdash \mathbf{rec}\,(r, o, \bar{w})\,; s + \sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i \xrightarrow{r(r,o,\bar{w})} s'} \quad (\mathit{Pick})$$

$$\frac{m \vdash s \cdot (\bar{v} \mapsto m \rhd \bar{c}) \xrightarrow{\alpha} s'}{m \vdash A(\bar{c}) \xrightarrow{\alpha} s'}\ A(\bar{v} : \bar{\tau}^{\star}) \stackrel{def}{=} s \quad (\mathit{Call})$$

Table 2. WS-CALCULUS operational semantics: instances

Finally, the last two auxiliary functions we need are:

- $P(\bar{w})$ returns the set of properties contained in the set of operation parameters $\bar{w}$.
- $eR(C)$ returns the set of activities $\mathbf{rec}\,(\cdot,\cdot,\cdot)$ that are start activities of instances in C (it is defined inductively on the syntax of C, see [LPT06]).

The labelled transition $\xrightarrow{\alpha}$ is the least relation over service instances induced by the rules in Table 2. For the sake of simplicity, we explicitly write only those entities that are necessary for a transition to occur or are modified by it. For example, since correlation constraints are sometimes necessary but are never modified by a transition, we write the relation as $m \vdash s \xrightarrow{\alpha} s'$ instead of $m \gg s \xrightarrow{\alpha} m \gg s'$. The most of the rules are obvious, we only remark a few points. Rule (*Receive*) states that a $\mathbf{rec}\,(\cdot,\cdot,\cdot)$ cannot be performed when the address of the sender of a request is unknown and cannot be learned (i.e. the first argument is neither an address nor a link). Rules (*Flow*) and (Flow_{Rec}) state that, in case no receive conflict occurs[2] (i.e. there aren't two or more receive activities simul-

[2] Sets of correlation constraints are exploited precisely to deal with receive conflicts: they prevent loss of correlation information (which would be lost if properties are simply re-

$$\frac{m \vdash s \xrightarrow{\bar{w}:=\bar{u}} s' \qquad m \rhd (\bar{w} := \bar{u}) = \langle m', \sigma \rangle}{\{a :: m \gg s \mid C\} \rightarrowtail \{a :: (m \cup m') \gg s' \cdot \sigma \mid C\}} \quad (\mathit{Assign})$$

$$\frac{m \vdash s \xrightarrow{i(a_2,o,\bar{u})} s' \qquad \underline{a}' \notin \bar{u}}{\{a_1 :: m \gg s \mid C_1,\ a_2 :: C_2\} \rightarrowtail \{a_1 :: m \gg s' \mid C_1,\ a_2 :: \langle a_1, o, \bar{u} \rangle \mid C_2\}} \quad (\mathit{Invoke})$$

$$\frac{m \vdash s \xrightarrow{r(a',o,\bar{w})} s' \qquad m \rhd (\bar{w} := \bar{u}) = \langle m', \sigma \rangle \qquad m \neq \emptyset}{\{a :: m \gg s \mid \langle a', o, \bar{u} \rangle \mid C\} \rightarrowtail \{a :: (m \cup m') \gg s' \cdot \sigma \mid C\}} \quad (\mathit{Receive}_{aI})$$

$$\frac{m \vdash s \xrightarrow{r(l,o,\bar{w})} s' \qquad m \rhd (l, \bar{w} := \underline{a}', \bar{u}) = \langle m', \sigma \rangle \qquad m \neq \emptyset}{\{a :: m \gg s \mid \langle a', o, \bar{u} \rangle \mid C\} \rightarrowtail \{a :: (m \cup m') \gg s' \cdot \sigma \mid C\}} \quad (\mathit{Receive}_{lI})$$

$$\frac{\emptyset \vdash s \xrightarrow{r(a',o,\bar{w})} s' \quad \emptyset \rhd (\bar{w} := \bar{u}) = \langle m, \sigma \rangle \quad \mathbf{rec}\,(a', o, \bar{w}) \notin eR(C)}{\{a :: *s \mid \langle a', o, \bar{u} \rangle \mid C\} \rightarrowtail \{a :: *s \mid m \gg s' \cdot \sigma \mid C\}} \quad (\mathit{Receive}_{aS})$$

$$\frac{\emptyset \vdash s \xrightarrow{r(l,o,\bar{w})} s' \quad \emptyset \rhd (l, \bar{w} := \underline{a}', \bar{u}) = \langle m, \sigma \rangle \quad \mathbf{rec}\,(l, o, \bar{w}) \notin eR(C)}{\{a :: *s \mid \langle a', o, \bar{u} \rangle \mid C\} \rightarrowtail \{a :: *s \mid m \gg s' \cdot \sigma \mid C\}} \quad (\mathit{Receive}_{lS})$$

$$\frac{m \vdash s \xrightarrow{\natural} s'}{\{a :: m \gg s \mid C\} \rightarrowtail \{a :: C\}} \quad (\mathit{Terminate}) \qquad \frac{N_1 \rightarrowtail N_1'}{N_1 \cup N_2 \rightarrowtail N_1' \cup N_2} \quad (\mathit{Part})$$

$$\frac{N \equiv N_1 \quad N_1 \rightarrowtail N_2 \quad N_2 \equiv N'}{N \rightarrowtail N'} \quad (\mathit{Cong})$$

Table 3. ws-calculus operational semantics: nets

taneously enabled for the same combination of partner link, operation and correlation set), executions of the two argument services are interleaved. Rule (*Pick*) states that, in case no receive conflict occurs, the pick activity can execute any of its receive activities and then proceed accordingly.

The reduction relation $\rightarrowtail$ is the least relation over nets induced by the rules in Table 3. Types of nodes are omitted because they play no role in the operational semantics of ws-calculus. Let us now comment on the rules. Rule (*Assign*) states that the effect of

placed by the corresponding values). For example, if properties are dealt with as basic variables, by applying the substitution $(p \mapsto 5)$ to the service instance $(\ldots \mathbf{rec}\,(r, o, \langle \ulcorner p \urcorner, v \rangle) \mid \mathbf{rec}\,(r, o, \langle \ulcorner p \urcorner, v' \rangle) \ldots)$ we would obtain $(\ldots \mathbf{rec}\,(r, o, \langle 5, v \rangle) \mid \mathbf{rec}\,(r, o, \langle 5, v' \rangle) \ldots)$ that does not permit to establish if the receive activities are in conflict. Indeed, we would obtain the same term by applying the substitution $(p \mapsto 5, v'' \mapsto 5)$ to $(\ldots \mathbf{rec}\,(r, o, \langle \ulcorner p \urcorner, v \rangle) \mid \mathbf{rec}\,(r, o, \langle v'', v' \rangle) \ldots)$, where no conflict occurs.

an assignment is global wrt the instance and consists of replacing the free occurrences of the variables bound by the assignment with the corresponding values and of extending the set of correlation constraints identifying the instance with the pairs resulting from the assignment. Rule (*Invoke*) states that service invocation corresponds to adding a service request to the dataspace of the invoked service provided that no address implicitly received is exported as operation parameter. The request is a tuple, containing the address a_1 of the invoker, the name of the invoked operation o and the message $\bar{u}$ (i.e. the arguments to be passed to o). Hence, the invocation of a remote service is asynchronous because the invoker can proceed before its request is processed. WS-BPEL also provides a synchronous invocation that forces the invoker to wait for an answer by the invoked service, which indeed performs a pair of activities *receive – reply*. In ws-calculus, this behaviour is rendered as execution of a pair of activities *invoke – receive* by the invoker and of a pair of activities *receive – invoke* by the invoked service. Rule ($Receive_{aI}$) states that activity receive cannot progress until a matching request has been received. Thus, differently from activity invoke, it is blocking. Requests are routed to the correct service instance by exploiting the partner link and the operation contained in the request, which must coincide with those in the label of the transition performed by the service instance, and the correlation constraints identifying the instance, which must enable the assignment of the values contained in the request to the parameters contained in the receive. The correlation set identifying the instance must not be empty otherwise it could not be possible to determine the correct instance to which the request must be delivered. When the reduction takes place, the matching request is consumed and the effect on the instance is the same as that of the corresponding assignment. Rule ($Receive_{lI}$) differs only because in this case the address of the invoker is not known in advance. After the reduction, the address contained in the request is marked as not further transmissible and is used to replace the partner link occurring in the receive. The last two rules for the activity receive, ($Receive_{aS}$) and ($Receive_{lS}$), permit to create a new service instance on receipt of a request that cannot be routed to an existing instance. The additional premise prevents interferences with the first two rules for receive in case of multiple start activities, as illustrated by the example

$$\{a :: *(\mathbf{rec}\,(l, o, \langle p\rangle) \mid \mathbf{rec}\,(l', o', \langle p\rangle)) \quad | \quad \{p = 10\} \gg \mathbf{rec}\,(l, o, \langle p\rangle) \quad | \quad \langle a', o, \langle 10\rangle\rangle\}$$

where only the service instance can evolve. Rule (*Terminate*) states that the whole service instance performing a transition labelled $\natural$ immediately terminates. Rule (*Part*) states that if a part of a larger net evolves, the whole net evolves accordingly. Rule (*Cong*) is standard and states that structural congruent nets have the same reductions.

3 Types

The syntax of types is defined in Table 4. An *operation type set* Op is a collection of type definitions of operations $o : \bar{\tau}$, where $\bar{\tau}$ is a tuple of *message types* that characterizes the format of the arguments that an operation requires. We assume that the type definition of a given operation only occurs at a single node within a net. *Local declarations* L consist of type definitions of basic variables and properties, which have *basic types* bt

Op	$::=$	$\emptyset \mid \{o : \bar{\tau}\} \mid Op \cup Op$	(operation type sets)
L	$::=$	$\emptyset \mid \{b : bt\} \mid \{p : bt\} \mid L \cup L$	(local declarations)
bt	$::=$	$\textsc{Int} \mid \textsc{Str} \mid \textsc{Bool}$	(basic types)
τ	$::=$	$bt \mid Op$	(message types)
t	$::=$	$\bar{\tau} \mid \textsc{bnet} \mid \textsc{bserv}$	(generic types)

Table 4. Type syntax

(for simplicity sake, we only consider Int, Str and Bool). Types bnet and bserv are those of (well-typed) nets and services, respectively.

In the sequel, we will use the symbol $\star$ to type partner links that are implicitly initialized (i.e. they are bound as first argument of a **rec**$(,,)$). Notation $\tau^\star$, which is used to type the parameters of service definitions (see the previous section), stands for a message type τ or for $\star$. Typing a parameter with $\star$ means that it is a partner link that should have been bound implicitly by a receive activity that syntactically precedes the service call. Moreover, notation $\_s$ shall denote both service specifications ($*s$) and service instances ($m \gg s$).

Type inference. Type environments, ranged over by Γ, map addresses and partner links to sets of operation types Op or to $\star$, and service identifiers to bserv. If $x \notin dom(\Gamma)$, we write $\Gamma, x : t$ for the type environment obtained by extending Γ with the binding of x to t (the notation generalizes to $\Gamma, \{x_i : t_i\}_{i \in I}$ with the obvious meaning). We write $\emptyset$ to denote the type environment with empty domain. Type environments are ordered by the standard preorder over functions, thus we write $\Gamma \leqslant \Gamma'$ when $dom(\Gamma) \supseteq dom(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for each $x \in dom(\Gamma')$.

Type environments hold the types of nodes and of partner links. This information is exploited to properly deal with address passing (indeed, invoke and receive activities can use partner links as parameters to exchange node addresses). The type of a partner link is a set of operation types Op stating that the partner link can be bound only to addresses holding a type Op' such that $Op \subseteq Op'$. During the type checking wrt Γ, we can easily determine if a partner link l has been implicitly or explicitly initialized according to the fact that $\Gamma(l)$ is $\star$ or Op, respectively. When a partner link is implicitly initialized, the type system checks that the associated address is never transmitted (the example in Section 4 shows that this limitation does not affect the expressive power of the calculus), as required by WSDL / WS-BPEL. When a partner link is explicitly initialized, the type system checks that the link is not reassigned (in fact, this control is done implicitly because if $l \in dom(\Gamma)$ then $\Gamma, l : Op$ is undefined). Type environments also hold service identifiers: this information is exploited when typing recursive service definitions.

The judgment $\Gamma \vdash N : \textsc{bnet}$, defined by the inference rules of Table 5, says that a net N is well-typed under the type environment Γ. The initial type environment used to typecheck a net does not contain type associations for partner links; this kind of associations may be added to the environment during the type checking of services,

$$\frac{\forall\, i \in I \qquad \Gamma, \{a_j : Op_j \mid j \in I\} \vdash a_i ::^{Op_i, L_i} C_i : \textsc{bnet}}{\Gamma \vdash \{a_i ::^{Op_i, L_i} C_i \mid i \in I\} : \textsc{bnet}}\ (net)$$

$$\frac{\Gamma \vdash^L_a C : \textsc{bserv}}{\Gamma \vdash a ::^{Op, L} C : \textsc{bnet}}\ (netToServ) \qquad \frac{\Gamma' \vdash N : \textsc{bnet} \qquad \Gamma \leqslant \Gamma'}{\Gamma \vdash N : \textsc{bnet}}\ (netWeak)$$

Table 5. Inference rules for $\Gamma \vdash N : \textsc{bnet}$

by means of the function $envExt_{\_}(\cdot)$. Rule (*net*) says that a net is well-typed under a type environment, if each node is well-typed under the environment extended with type information extracted from all nodes. Rule (*netToServ*) says that a node is well-typed if its components are well-typed. Rule (*netWeak*) says that a type environment can be replaced with a stronger one (i.e. one making more assumptions).

The judgment $\Gamma \vdash^L_a S : t$, defined by the set of inference rules shown in Tables 6 and 7, says that S has type t, where S is a metavariable denoting values, variables, properties, requests and services, wrt a type environment Γ and a pair a–L made of the address of a node and a set of local declarations. The symbol $\sqsubseteq$ denotes the subtyping preorder over τ induced by letting $Op \sqsubseteq Op'$ if $Op \subseteq Op'$. The preorder extends to tuples of message types by letting $\langle \tau_1, \ldots, \tau_n \rangle \sqsubseteq \langle \tau'_1, \ldots, \tau'_n \rangle$ if $\tau_i \sqsubseteq \tau'_i$ for $i = 1..n$. To distinguish partner links within a tuple of variables and properties, we exploit the auxiliary function $pl(\cdot)$ that, given a tuple $\bar{w}_{i \in I}$, returns the set of indexes of the partner links therein. The function is defined inductively on the syntax of $\bar{w}_{i \in I}$ as follows:

$$pl(b_i) = \emptyset \qquad pl(p_i) = \emptyset \qquad pl(l_i) = \{i\} \qquad pl(\bar{w}_{i \in I}) = \bigcup_{i \in I} pl(w_i)$$

We comment on the most significant rules in Table 7, since the rules in Table 6 are standard. Rule (*inv*) is applied when an invoke activity is performed by a client in a one-way interaction or to start a request-response interaction. In these cases, indeed, the address of the provider (holding the type definition of the operation) is given by r. Of course, the parameters of the invoked operation must conform to the corresponding operation type. In particular, when an invoke transmits an address, e.g. $\bar{w} = \bar{w}_{i \in I}$, $w_k = r'$ and $\tau_k = Op''$, then it must be that $Op'' \subseteq Op'$ where Op' is the operation type set associated to r' in Γ (i.e. $\Gamma \vdash^L_a r' : Op'$). This is indeed what the condition $\bar{\tau} \sqsubseteq \bar{\tau}'$ checks. Rule (*inv_cb*) is applied to an invoke activity performed as a callback in a request-response interaction. The local node is the operation provider. The only difference with the previous rule is that, in case the first argument of the activity is a partner link l, it is additionally checked that a triggering receive activity which initializes l logically precedes the invoke (this is expressed by the premise $\Gamma \vdash^L_a l : \star$). Rule (*rec_cb*) is applied when a client performs a receive activity to obtain a callback in a request-response interaction. Similarly to rule (*inv*), the type of the operation must be retrieved from the provider node whose address is given by r. In case the first argument of the operation is a free occurrence of a link it is checked that the link is not transmit-

$$\frac{u \in \mathtt{Int}}{\emptyset \vdash^L_a u : \textsc{Int}}\ (int) \qquad \frac{u \in \mathtt{Str}}{\emptyset \vdash^L_a u : \textsc{Str}}\ (str) \qquad \frac{u \in \{\mathbf{true}, \mathbf{false}\}}{\emptyset \vdash^L_a u : \textsc{Bool}}\ (bool)$$

$$r : Op \vdash^L_a r : Op \quad (ref) \qquad \frac{b : \tau \in L}{\emptyset \vdash^L_a b : \tau}\ (var) \qquad \frac{p : \tau \in L}{\emptyset \vdash^L_a p : \tau}\ (prop)$$

$$l : Op \vdash^L_a \ulcorner l \urcorner : Op \quad (ref_2) \qquad \frac{\Gamma \vdash^L_a w_1 : \tau_1 \quad \ldots \quad \Gamma \vdash^L_a w_n : \tau_n}{\Gamma \vdash^L_a \langle w_1, \ldots, w_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}\ (\bar{w})$$

$$\frac{\Gamma' \vdash^L_a S : t \qquad \Gamma \leqslant \Gamma'}{\Gamma \vdash^L_a S : t}\ (weak) \qquad \frac{\Gamma \vdash^L_a e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash^L_a e_n : \tau_n}{\Gamma \vdash^L_a \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}\ (\bar{e})$$

Table 6. Inference rules for $\Gamma \vdash^L_a S : t$

ted. Rule (*rec*) is similar but is applied when the local node is the provider of the receive activity. Rule (*seq*) says that a sequence of services $s_1; s_2$ is well-typed under Γ if s_1 is well-typed under Γ and s_2 is well-typed under Γ extended with the type associations for the partner links bound by s_1 (notably, the extension is possible only if such partner links are not reassigned). The set of new associations is returned by the auxiliary function $envExt_{\_,\_}(\cdot)$, that can be defined inductively on the syntax of services. The most significant cases, i.e. those for the binding constructs, are defined as follows:

$$envExt_{\Gamma,a}(\mathbf{rec}\,(r, o, \bar{w})) = \begin{cases} \{l : \tau_i \mid \exists i \,.\, w_i = l \ \wedge\ \tau_i \cap Op = \emptyset\} \cup \{l' : \star \mid r = l'\} & \text{if } \Gamma \vdash^{\emptyset}_a a : Op \ \wedge\ o : \bar{\tau} \in Op \\ \{l : \tau_i \mid \exists i \,.\, w_i = l \ \wedge\ \Gamma \vdash^{\emptyset}_a r : Op' \ \wedge\ o : \bar{\tau} \in Op' \wedge \\ \qquad \Gamma \vdash^{\emptyset}_a a : Op \ \wedge\ \tau_i \cap Op = \emptyset\} & \text{otherwise} \end{cases}$$

$$envExt_{\Gamma,a}(\mathbf{ass}\,(\bar{w}, \bar{e})) = \{l : \tau_i \mid \exists i \,.\, w_i = l \wedge\ \Gamma \vdash^{\emptyset}_a \bar{e} : \bar{\tau} \ \wedge\ \Gamma \vdash^{\emptyset}_a a : Op \ \wedge\ \tau_i \cap Op = \emptyset\}$$

Condition $\tau_i \cap Op = \emptyset$ avoids that the service executing the activity can receive its same address as an argument. Finally, rule (*flow*) forces the two component services to type check in the same environment. This control prevents implicit flow of addresses from one component to the other (recall that partner link declarations are global to the instance) which would force a strict execution ordering of the components (thus, e.g., the service instance $\mathbf{rec}\,(l, o, w) \mid \mathbf{inv}\,(l, o', 5)$ does not type check).

Type soundness. The major property of our type system is that if a net typechecks then it does never generate runtime errors (Corollary 1). The proof proceeds in the style of [WF94] by first proving *subject reduction*, namely that nets well-typedness is an invariant of the reduction relation (Theorem 1), and then proving *type safety*, namely that well-typed nets do not immediately generate errors (Theorem 2). Due to lack of

$$\frac{\Gamma \vdash_a^L C_1 : \text{BSERV} \qquad \Gamma \vdash_a^L C_2 : \text{BSERV}}{\Gamma \vdash_a^L C_1 \mid C_2 : \text{BSERV}} \ (par) \qquad \frac{\Gamma \vdash_a^L s : \text{BSERV}}{\Gamma \vdash_a^L \_s : \text{BSERV}} \ (serv)$$

$$\frac{\Gamma \vdash_a^L a : Op \qquad \Gamma \vdash_a^L a' : Op' \qquad o : \bar{\tau} \in Op \cup Op' \qquad \Gamma \vdash_a^L \bar{u} : \bar{\tau}' \qquad \bar{\tau} \sqsubseteq \bar{\tau}'}{\Gamma \vdash_a^L \langle a', o, \bar{u} \rangle : \text{BSERV}} \ (req)$$

$$\emptyset \vdash_a^L \mathbf{0} : \text{BSERV} \ (nil) \qquad \emptyset \vdash_a^L \mathbf{exit} : \text{BSERV} \ (exit) \qquad A : \text{BSERV} \vdash_a^L A : \text{BSERV} \ (def_2)$$

$$\frac{\Gamma \vdash_a^L \bar{e}_{i\in I} : \bar{\tau}_{i\in I} \qquad \Gamma \vdash_a^L \bar{w}_{i\in(I\setminus J)} : \bar{\tau}_{i\in(I\setminus J)}}{\Gamma \vdash_a^L \mathbf{ass}\,(\bar{w}_{i\in I}, \bar{e}_{i\in I}) : \text{BSERV}} \ J = pl(\bar{w}_{i\in I}) \ (ass)$$

$$\frac{\Gamma \vdash_a^L r : Op \qquad o : \bar{\tau} \in Op \qquad \Gamma \vdash_a^L \bar{w} : \bar{\tau}' \qquad \bar{\tau} \sqsubseteq \bar{\tau}'}{\Gamma \vdash_a^L \mathbf{inv}\,(r, o, \bar{w}) : \text{BSERV}} \ (inv)$$

$$\frac{\Gamma \vdash_a^L a : Op \qquad o : \bar{\tau} \in Op \qquad \Gamma \vdash_a^L \bar{w} : \bar{\tau}' \qquad \bar{\tau} \sqsubseteq \bar{\tau}' \qquad r = l \Rightarrow \Gamma \vdash_a^L l : \star}{\Gamma \vdash_a^L \mathbf{inv}\,(r, o, \bar{w}) : \text{BSERV}} \ (inv\_cb)$$

$$\frac{\Gamma \vdash_a^L r : Op \qquad o : \bar{\tau}_{i\in I} \in Op \qquad \Gamma \vdash_a^L \bar{w}_{j\in(I\setminus J)} : \bar{\tau}_{j\in(I\setminus J)} \qquad r \neq l \qquad r = \ulcorner l \urcorner \Rightarrow \forall\, i \in J \ \ w_i \neq l}{\Gamma \vdash_a^L \mathbf{rec}\,(r, o, \bar{w}_{i\in I}) : \text{BSERV}} \ J = pl(\bar{w}_{i\in I}) \ (rec\_cb)$$

$$\frac{\Gamma \vdash_a^L a : Op \qquad o : \bar{\tau}_{i\in I} \in Op \qquad \Gamma \vdash_a^L \bar{w}_{j\in(I\setminus J)} : \bar{\tau}_{j\in(I\setminus J)} \qquad r = \ulcorner l \urcorner \Rightarrow \forall\, i \in J \ \ w_i \neq l \ \wedge \ \Gamma \vdash_a^L l : \star}{\Gamma \vdash_a^L \mathbf{rec}\,(r, o, \bar{w}_{i\in I}) : \text{BSERV}} \ J = pl(\bar{w}_{i\in I}) \ (rec)$$

$$\frac{\Gamma \vdash_a^L e : \text{BOOL} \qquad \Gamma \vdash_a^L s_1 : \text{BSERV} \qquad \Gamma \vdash_a^L s_2 : \text{BSERV}}{\Gamma \vdash_a^L \mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\} : \text{BSERV}} \ (if)$$

$$\frac{\Gamma \vdash_a^L s_1 : \text{BSERV} \qquad \Gamma,\, envExt_{\Gamma,a}(s_1) \vdash_a^L s_2 : \text{BSERV}}{\Gamma \vdash_a^L s_1 ; s_2 : \text{BSERV}} \ (seq)$$

$$\frac{\Gamma \vdash_a^L s_1 : \text{BSERV} \qquad \Gamma \vdash_a^L s_2 : \text{BSERV}}{\Gamma \vdash_a^L s_1 \mid s_2 : \text{BSERV}} \ (flow) \qquad \frac{\forall\, i \in I \quad \Gamma \vdash_a^L \mathbf{rec}\,(r_i, o_i, \bar{w}_i) ; s_i : \text{BSERV}}{\Gamma \vdash_a^L \sum_{i\in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i) ; s_i : \text{BSERV}} \ (pick)$$

$$\frac{\Gamma \vdash_a^L A : \text{BSERV} \qquad \Gamma \vdash_a^L \bar{w} : \bar{\tau}_1^\star \qquad \bar{\tau}^\star \sqsubseteq \bar{\tau}_1^\star}{\Gamma \vdash_a^L A(\bar{w}) : \text{BSERV}} \ A(\bar{v} : \bar{\tau}^\star) \stackrel{def}{=} s \ \ (call)$$

$$\frac{\Gamma,\, A : \text{BSERV},\, \bar{v}_{j\in J} : \bar{\tau}^\star_{j\in J} \vdash_a^{L \,\cup\, \{\bar{v}_{i\in(I\setminus J)} : \bar{\tau}^\star_{i\in(I\setminus J)}\}} s : \text{BSERV}}{\Gamma \vdash_a^L A : \text{BSERV}} \ A(\bar{v}_{i\in I} : \bar{\tau}^\star_{i\in I}) \stackrel{def}{=} s \ ,\ J = pl(\bar{v}_{i\in I}) \ \ (def_1)$$

Table 7. Inference rules for $\Gamma \vdash_a^L S : t$ (cont.)

space, we omit the proofs of the results presented in this section (they are quite standard and can be found in [LPT06]).

First, we introduce some auxiliary definitions. A (generic) *context* C_g is a service with one subterm replaced by a hole, denoted $[\cdot]$. Formally, C_g is defined as follows:

$$\begin{array}{rcl} C_g & ::= & [\cdot] \mid C_g \mid s \mid C_g; s \mid s; C_g \mid \mathbf{rec}\,(r, o, \bar{w})\,; C_g \;+\; \sum_{i\in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i \\ & \mid & \mathbf{if}\ (e)\ \mathbf{then}\ \{s\}\ \mathbf{else}\ \{C_g\} \mid \mathbf{if}\ (e)\ \mathbf{then}\ \{C_g\}\ \mathbf{else}\ \{s\} \mid A(\bar{c}) \end{array}$$

where the body of the service definition is a context, i.e. $A(\bar{v} : \bar{\tau}^{\star}) \stackrel{def}{=} C_g$. Notably, terms of the form $[\cdot]\,; s + \sum_{i\in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i$ are not considered (for the moment). Notation $C_g[s]$ denotes the service resulting from filling the hole of C_g with service s.

An *execution context* C is a context such that, once the hole is filled with a service s, the resulting service $C[s]$ is capable of immediately performing an activity of s. Formally, C is defined by the following grammar:

$$\begin{array}{rcl} C & ::= & [\cdot] \mid C \mid s \mid C; s \mid \mathbf{if}\ (e)\ \mathbf{then}\ \{C\}\ \mathbf{else}\ \{s\} \mid \\ & & \mathbf{if}\ (e)\ \mathbf{then}\ \{s\}\ \mathbf{else}\ \{C\} \mid A(\bar{c}) \end{array}$$

where $A(\bar{v} : \bar{\tau}^{\star}) \stackrel{def}{=} C$. Whenever, we only consider basic receive activities, we can extend the set of possible execution contexts as follows:

$$C_r \;::=\; C \;\mid\; [\cdot]\,; s \;+\; \sum_{i\in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i$$

The subject reduction theorem exploits the following two auxiliary lemmas. The former is the key for showing type preservation for reductions involving substitutions, the latter states that if a service is well-typed then its continuation after a transition is well-typed too.

Lemma 1. *Suppose* $\Gamma \vdash_a^L s : \text{BSERV}$. *If* $\Gamma \vdash_a^L v : \tau$, $\Gamma \vdash_a^L u : \tau'$ *and* $\tau \sqsubseteq \tau'$, *then* $\Gamma \vdash_a^L s \cdot (v \mapsto u) : \text{BSERV}$.

Lemma 2. *If* $\Gamma \vdash_a^L s : \text{BSERV}$ *and* $m \vdash s \xrightarrow{\alpha} s'$ *then* $\Gamma' \vdash_a^L s' : \text{BSERV}$ *with* Γ' *such that*[3]*:*

- $\Gamma' \leqslant \Gamma$ *in case of* $\alpha = \natural,\ i(a', o, \bar{u})$;
- $\Gamma' \leqslant \Gamma, envExt_{\Gamma,a}(\mathbf{rec}\,(r, o, \bar{w}))$ *in case of* $\alpha = r(r, o, \bar{w})$;
- $\Gamma' \leqslant \Gamma, envExt_{\Gamma,a}(\mathbf{ass}\,(\bar{w}, \bar{e}))$ *in case of* $\alpha = (\bar{w} := \bar{u})$ *and* $s \equiv C[\mathbf{ass}\,(\bar{w}, \bar{e})]$.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash N : \text{BNET}$ *and* $N \rightarrowtail N'$ *then* $\Gamma' \vdash N' : \text{BNET}$ *for some* Γ'.

The errors that our type system can capture, are characterized by predicate $\rightarrowtail^{err}$ that holds true when a net can immediately generate a runtime error. The most significant rules defining $\rightarrowtail^{err}$ are in Table 8 (the remaining of rules can be found in [LPT06]).

Rule (*opDefError1*) raises an error when an operation is invoked whose type declaration is neither in the type of the caller nor in that of the callee. Rule (*opDefError2*)

[3] Notably, wrt the type environment on the right of $\leqslant$, Γ' can additionally contain further associations due to service calls. This explains the use of $\leqslant$ instead of $=$.

$$\frac{s \equiv C[\mathbf{inv}\,(a', o, \bar{w})] \quad o : \bar{\tau} \notin Op \quad o : \bar{\tau} \notin Op'}{\{a ::^{Op,L} \_s \mid C,\ a' ::^{Op',L'} C'\} \succ\!\!\rightarrow^{err}} \quad (opDefError1)$$

$$\frac{s \equiv C_r[\mathbf{rec}\,(l, o, \bar{w})] \quad o : \bar{\tau} \notin Op}{\{a ::^{Op,L} \_s \mid C\} \succ\!\!\rightarrow^{err}} \quad (opDefError2)$$

$$\frac{s \equiv C_r[\mathbf{rec}\,(a', o, \bar{w})] \quad o : \bar{\tau} \notin Op \quad o : \bar{\tau} \notin Op'}{\{a ::^{Op,L} \_s \mid C,\ a' ::^{Op',L'} C'\} \succ\!\!\rightarrow^{err}} \quad (opDefError3)$$

$$\frac{s \equiv C_r[\mathbf{rec}\,(l, o, \bar{w})\,; C_g[\mathbf{inv}\,(l', o', \bar{w}')]] \quad \exists i\,.\, w'_i = l}{\{a ::^{Op,L} \_s \mid C\} \succ\!\!\rightarrow^{err}} \quad (linkError)$$

$$\frac{s \equiv C[\mathbf{inv}\,(l, o, \bar{w})] \quad o : \bar{\tau} \in Op}{\{a ::^{Op,L} \_s \mid C\} \succ\!\!\rightarrow^{err}} \quad (rrError1)$$

$$\frac{s \equiv C[\mathbf{ass}\,(\bar{w}, \bar{u})\,; C_g[\mathbf{inv}\,(l, o, \bar{w}')]] \quad o : \bar{\tau} \in Op \quad \exists i\,.\, w_i = l}{\{a ::^{Op,L} \_s \mid C\} \succ\!\!\rightarrow^{err}} \quad (rrError2)$$

Table 8. Runtime errors (selected rules)

raises an error if the type declaration of the requested operation is not found in the type of the local node. Indeed, the service must be the provider since the activity first argument is a link. If the first argument of the activity is an address, there is no way to tell if the service is a client or a provider. Therefore, rule (*opDefError3*) raises an error only if the type declaration of the requested operation is neither in the type of the callee nor in that of the caller. Rule (*linkError*) raises an error when a partner link implicitly initialized is going to be passed in a communication. Rule (*rrError1*) raises an error if a callback invoke is going to be executed that does not have a previous triggering receive (indeed, its first argument is uninitialized). Finally, rule (*rrError2*) raises an error if the first argument of a callback invoke is initialized by an assignment rather than by a triggering receive.

Theorem 2 (Type Safety). *$\Gamma \vdash N : \textsc{bnet}$ implies that $N \succ\!\!\rightarrow^{err}$ holds false.*

To conclude, we have ($\succ\!\!\rightarrow^{*}$ denotes the reflexive and transitive closure of $\succ\!\!\rightarrow$)

Corollary 1 (Type Soundness). *Let $\Gamma \vdash N : \textsc{bnet}$. Then $N' \succ\!\!\rightarrow^{err}$ holds false for every net N' such that $N \succ\!\!\rightarrow^{*} N'$.*

4 A brokerage scenario

In this section we show an application of our framework. Suppose a client process invokes a process that acts as a broker for a third process. The latter process, once

received a message with an integer value and the client address, increases the value by one (of course, this can be replaced with any complex operation) and sends the response back to the client by exploiting the received address. This scenario is modelled by the net (we write $Z \triangleq W$ to assign a symbolic name Z to the term W)

$$N \triangleq \{a_c ::^{Op_c, L_c} *s_c \mid \langle a_c, o_{init}, 10\rangle \ , \ a_b ::^{Op_b, L_b} *s_b \ , \ a_r ::^{Op_r, L_r} *s_r\} \tag{1}$$

where a_c, a_b and a_r are the addresses of client, broker and responder, respectively.

The client service is defined as follows:

$$s_c \triangleq \mathbf{rec}\,(l_{init}, o_{init}, p)\,;\mathbf{inv}\,(a_b, o, \langle p, a_c\rangle)\,;\mathbf{rec}\,(l_r, o_{cb}, \langle p, res\rangle)$$

The first receive creates a client instance by consuming the initialization tuple $\langle a_c, o_{init}, 10\rangle$. Since multiple client instances can wait a response along the same partner link and operation, we use a correlation set to route each incoming message to the correct instance. At instantiation time, a correlation set consisting of the property p is initialized. When the client process invokes the broker, it must send an integer value and its address to allow the responder process to send back the reply. After this invocation, the client waits the callback. The client type declarations are $Op_c = \{o_{init} : \langle \textsc{Int}\rangle, o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle\}$ and $L_c = \{p : \textsc{Int}, res : \textsc{Int}\}$. Notice that, in this communication pattern, differently from asynchronous request-response, the client has the provider role for the callback operation.

The broker service is defined as follows:

$$s_b \triangleq \mathbf{rec}\,(l, o, \langle b, l_c\rangle)\,;\mathbf{inv}\,(a_r, o', \langle b, l_c\rangle)$$

When invoked, the broker creates an instance (by using the receive activity) that will forward the client request to the responder and then terminate. Since no session with multiple interactions is started, the broker does not use a correlation mechanism. The broker type declarations are $Op_b = \{o : \bar{\tau}\}$ with $\bar{\tau} = \langle \textsc{Int}, \{o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle\}\rangle$ and $L_b = \{b : \textsc{Int}\}$. Of course, the broker has the provider role for the operation invoked by the client. In the message type of the operation, the second field is an operation type set and identifies the client operations that are visible to the broker.

Finally, the responder service is defined as follows:

$$s_r \triangleq \mathbf{rec}\,(l, o', \langle b, l_{cb}\rangle)\,;\mathbf{ass}\,(b', b+1)\,;\mathbf{inv}\,(l_{cb}, o_{cb}, \langle b, b'\rangle)$$

When invoked, the responder creates an instance that will process the received value and send the response back to the client. Also this process does not need a correlation mechanism. The responder type declarations are $Op_r = \{o' : \bar{\tau}\}$ and $L_r = \{b : \textsc{Int}, b' : \textsc{Int}\}$. Notice that, since the responder receives the client address from the broker, its view of client operations along the partner link l_{cb} agrees with that of the broker.

According to our framework, to ensure that N will never generate errors, it suffices to prove that N is well-typed wrt the empty environment, i.e. $\emptyset \vdash N : \textsc{bnet}$. This, by rule (*net*), means that each node of N must be well-typed wrt the type environment $\Gamma = \{a_c : Op_c,\ a_b : Op_b,\ a_r : Op_r\}$. Now, by the rule (*netToServ*), (*par*) and (*serv*) this holds if all components $\langle a_c, o_{init}, 10\rangle$, s_c, s_b and s_r are well-typed wrt Γ and appropriate local

type declarations. Formally, we must check that judgements $\Gamma \vdash^{L_c}_{a_c} \langle a_c, o_{init}, 10\rangle : \text{BSERV}$, $\Gamma \vdash^{L_c}_{a_c} s_c : \text{BSERV}$, $\Gamma \vdash^{L_b}_{a_b} s_b : \text{BSERV}$ and $\Gamma \vdash^{L_r}_{a_r} s_r : \text{BSERV}$ hold. The second inference is fully shown in Table 9 (the remaining inferences can be found in [LPT06]). For the sake of presentation, the inferences are split in a few parts with references between them.

Notably, for both receive activities we must apply rule (*rec*), because the type environment does not store type information for the partner links l_{init} and l_r. Indeed, the client has provider role for both the operations o_{init} and o_{cb} and we check if their type definitions are in the set of operation types of the client Op_c, which is obtained by inferring $\Gamma \vdash^{L_c}_{a_c} a_c : Op_c$. Instead, to check the invoke activity, we apply rule (*inv*), because in this case the service has client role. Op_b contains the type definition of the invoked operation o and is obtained by the inference of $\Gamma_c \vdash^{L_c}_{a_c} a_b : Op_b$, where a_b is the target of the invoke activity. Notice that the type associated to o is a subtype of the type associated to the operation parameters (i.e. $\bar{\tau} \sqsubseteq \langle \text{INT}, Op_c\rangle$), because $\{o_{cb} : \langle \text{INT}, \text{INT}\rangle\} \subseteq Op_c$.

We have thus proved that the net N defined in 1 behaves correctly. Now, we smoothly modify N so that its execution would eventually generate a runtime error and show that our type system can statically point out this situation. Indeed, suppose that $o_{cb} : \langle \text{INT}, \text{INT}\rangle \notin Op_c$. This could take place, for example, in case the client tries a request-response interaction with the broker (which would be the provider of both operations). The modified net N' would behave as follows (we omit the responder node because it plays no role):

$$
\begin{aligned}
N' &\rightarrowtail \{a_c ::^{Op_c, L_c} *s_c \mid \langle a_c, o_{init}, 10\rangle \ ,\ a_b ::^{Op_b, L_b} *s_b\} \\
&\rightarrowtail \{a_c ::^{Op_c, L_c} *s_c \mid \{p = 10\} \gg s'_c \ ,\ a_b ::^{Op_b, L_b} *s_b\} \\
&\rightarrowtail \{a_c ::^{Op_c, L_c} *s_c \mid \{p = 10\} \gg \mathbf{rec}\,(l_r, o_{cb}, \langle p, res\rangle) \ ,\ a_b ::^{Op_b, L_b} *s_b \mid \langle a_c, o, \langle 10, a_c\rangle\rangle\} \\
&\rightarrowtail^{err}
\end{aligned}
$$

where the runtime error is generated by rule (*opDefError2*). This situation can be captured in advance, since N' is not well-typed because, in the inference for the client service, $\Gamma_c \vdash^{L_c}_{a_c} \mathbf{rec}\,(l_r, o_{cb}, \langle p, res\rangle) : \text{BSERV}$ cannot be inferred.

5 Concluding remarks

We have set a formal semantics framework for web services orchestration languages, and particularly for WS-BPEL. We have introduced ws-CALCULUS, a foundational language specifically designed for modelling interactions among web services, and a type system that permits to formalize the relationship between WS-BPEL processes and the associated WSDL documents. The type system forces a neat programming discipline for communicating processes. We have shown that the type system and the operational semantics of ws-CALCULUS are 'sound' and presented an illustrative example.

We are currently extending the typing system, and the related results, to the enriched language described in [LPT06]. We also plan to enrich the type system to enforce more rigorous type disciplines. For example, partner links could have assigned more sophisticated types that would correspond to complex interaction patterns, such as, e.g., 'one request – multiple responses' or 'one request – one of two possible responses'. Moreover, by exploiting some form of 'behavioural' types, such dynamic aspects of ws-CALCULUS processes could be captured as, e.g., 'an operation parameter may determine whether

Table 9. Type inference for the client service s_c (Γ_c is $(\Gamma, l_{init} : \star)$)

$$\dfrac{\dfrac{\dfrac{}{a_c : Op_c \vdash^{L_c}_{a_c} a_c : Op_c}\,(ref)}{\Gamma_c \vdash^{L_c}_{a_c} a_c : Op_c}\,(weak) \qquad o_{cb} : \langle \textsc{Int}, \textsc{Int} \rangle \in Op_c \qquad \dfrac{\dfrac{\dfrac{p : \textsc{Int} \in L_c}{\emptyset \vdash^{L_c}_{a_c} p : \textsc{Int}}\,(prop)}{\Gamma_c \vdash^{L_c}_{a_c} p : \textsc{Int}}\,(weak) \qquad \dfrac{\dfrac{res : \textsc{Int} \in L_c}{\emptyset \vdash^{L_c}_{a_c} res : \textsc{Int}}\,(var)}{\Gamma_c \vdash^{L_c}_{a_c} res : \textsc{Int}}\,(weak)}{\Gamma_c \vdash^{L_c}_{a_c} \langle p, res \rangle : \langle \textsc{Int}, \textsc{Int} \rangle}\,(\bar{w})}{(2)\ \Gamma_c \vdash^{L_c}_{a_c} \mathbf{rec}\,(l_r, o_{cb}, \langle p, res \rangle) : \textsc{bserv}}\,(rec)$$

$$\dfrac{\dfrac{\dfrac{}{a_b : Op_b \vdash^{L_c}_{a_c} a_b : Op_b}\,(ref)}{\Gamma_c \vdash^{L_c}_{a_c} a_b : Op_b}\,(weak) \qquad o : \bar{\tau} \in Op_b \qquad \dfrac{\dfrac{\dfrac{p : \textsc{Int} \in L_c}{\emptyset \vdash^{L_c}_{a_c} p : \textsc{Int}}\,(prop)}{\Gamma_c \vdash^{L_c}_{a_c} p : \textsc{Int}}\,(weak) \qquad \dfrac{\dfrac{}{a_c : Op_c \vdash^{L_c}_{a_c} a_c : Op_c}\,(ref)}{\Gamma_c \vdash^{L_c}_{a_c} a_c : Op_c}\,(weak)}{\Gamma_c \vdash^{L_c}_{a_c} \langle p, a_c \rangle : \langle \textsc{Int}, Op_c \rangle}\,(\bar{w}) \qquad \bar{\tau} \sqsubseteq \langle \textsc{Int}, Op_c \rangle}{(1)\ \Gamma_c \vdash^{L_c}_{a_c} \mathbf{inv}\,(a_b, o, \langle p, a_c \rangle) : \textsc{bserv}}\,(inv)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a_c : Op_c \vdash^{L_c}_{a_c} a_c : Op_c}\,(ref)}{\Gamma \vdash^{L_c}_{a_c} a_c : Op_c}\,(weak) \qquad o_{init} : \langle \textsc{Int} \rangle \in Op_c \qquad \dfrac{\dfrac{p : \textsc{Int} \in L_c}{\emptyset \vdash^{L_c}_{a_c} p : \textsc{Int}}\,(prop)}{\Gamma \vdash^{L_c}_{a_c} p : \textsc{Int}}\,(weak)}{\Gamma \vdash^{L_c}_{a_c} \mathbf{rec}\,(l_{init}, o_{init}, p) : \textsc{bserv}}\,(rec) \qquad \dfrac{(1)\ \Gamma_c \vdash^{L_c}_{a_c} \mathbf{inv}\,(a_b, o, \langle p, a_c \rangle) : \textsc{bserv} \quad (2)\ \Gamma_c \vdash^{L_c}_{a_c} \mathbf{rec}\,(l_r, o_{cb}, \langle p, res \rangle) : \textsc{bserv}}{\Gamma_c \vdash^{L_c}_{a_c} \mathbf{inv}\,(a_b, o, \langle p, a_c \rangle)\,;\mathbf{rec}\,(l_r, o_{cb}, \langle p, res \rangle) : \textsc{bserv}}\,(seq)}{\Gamma \vdash^{L_c}_{a_c} \mathbf{rec}\,(l_{init}, o_{init}, p)\,;\mathbf{inv}\,(a_b, o, \langle p, a_c \rangle)\,;\mathbf{rec}\,(l_r, o_{cb}, \langle p, res \rangle) : \textsc{bserv}}\,(seq)$$

a callback uses operation A vs. operation B' or 'the invocation of a service of type X must be preceded by the invocation of a service of type Y'.

One major contribution of our work is the formal modelling of different aspects of WS-BPEL, such as multiple start activities, receive conflicts, routing of correlated messages, interactions among different web services, that have not been tackled at once in the literature. The mechanism of correlation sets was first investigated in [Vir04], that however only consider interaction of different instances of a single business process. Other works take the opposite route, and enrich some well-known process calculus with constructs inspired by those of WS-BPEL. The most of them deal with issues of web transactions such as interruptible processes, failure handlers and time. This is, for instance, the case of [LZ05a,LZ05b] that present a timed extension of the π-calculus, called webπ, tailored to study a simplified version of the scope construct of WS-BPEL. We have focused on service orchestration rather than on service choreography (that provides a means to describe service interactions in a top-view way) because we wanted to study those problems arising when executing WS-BPEL processes. In [BGG+05] both aspects are studied. Following [MB03], we have pushed forward the use of a type system to define basic contracts for web services. In [CL06,HSS05], alternative approaches are proposed that are based on the use of schema languages and Petri nets, respectively.

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

References

[BCG+05] B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0. Technical report, WS-BPEL TC OASIS, 2005. http://www.oasis-open.org/.

[BGG+05] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In *ICSOC*, pages 228–240, 2005.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. Technical report, W3C, 2001. http://www.w3.org/TR/wsdl/.

[CL06] S. Carpineti and C. Laneve. A basic contract language for web services. In *Proceedings of ESOP 06, LNCS*, 2006.

[HSS05] S. Hinz, K. Schmidt, and C. Stahl. Transforming bpel to petri nets. In *Business Process Management*, pages 220–235, 2005.

[LPT06] A. Lapadula, R. Pugliese, and F. Tiezzi. A WSDL-based type system for WS-BPEL. Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2006. http://www.dsi.unifi.it/~pugliese/DOWNLOAD/wsc-full.ps.

[LZ05a] C. Laneve and G. Zavattaro. Foundations of web transactions. *Fossacs'05*, LNCS(3441):282–298, 2005.

[LZ05b] C. Laneve and G. Zavattaro. Webπ at work. In *TGC'05*, 2005.

[MB03] L. G. Meredith and S. Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.

[Vir04] M. Viroli. Towards a formal foundational to orchestration languages. *Electronic Notes in Theoretical Computer Science*, 105:51–71, 2004.

[WF94] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.