A WSDL-based type system for WS-BPEL*

Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi
Dipartimento di Sistemi e Informatica, Università di Firenze

March 28, 2006

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 a pair of illustrative examples. Finally, we extend the ws-calculus to define an operational semantics for whole WS-BPEL.

Contents

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

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 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.

Finally, we extend the ws-calculus to incorporate such other important constructs of WS-BPEL as flow graphs, timed activities, scopes and compensation handling, that should be considered for modeling the semantics of full-blown orchestration languages. We will show that flow graphs can be easily encoded in ws-calculus, while the semantics of the remaining constructs is given by extending the operational semantics of ws-calculus. This way, as a third contribution, we define an operational semantics for WS-BPEL *executable* processes.

A case study. As a reference case study, in this paper we consider the shipping service described in the official specification of WS-BPEL (Section 16.1). This example covers most of the language features we are interested in, including correlation sets, variables, and flow control structures. We assume that the reader is already familiar with the main features of WS-BPEL. This service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The shipping service offers two types of shipment: shipments where the items are held and shipped together and shipment where the items are shipped piecemeal until all of the order is accounted for.

The process definition of the shipping service exploits the following basic activities. The receive activity allows the service to receive orders from customers, the assign activity permits to handle the received data, and the invoke activity permits to send shipping notices to customers. An order request contains an order identifier that is used to correlate the ship notice(s) with the ship order. The flow control of the process is defined by means of conditional and iterative constructs. In our calculus, the latter is modelled by recursive definition. The formal description of the service will be presented in Section 4.2.

Overview of the paper. 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 two applications of our framework to modelling examples of web services composition. Section 5 extends ws-calculus and defines an operational semantics for the whole WS-BPEL. In Section 6 we touch upon directions for future work and comparisons with related work. The appendix contains a revisited version of the type system described in Section 3.

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 5)

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. ws-calculus can directly model the semantics of an expressive subset of WS-BPEL, we refer the interested reader to Section 5 for a complete account of whole WS-BPEL.

2.1 Syntax

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 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 $\mathbf{ass}(\cdot,\cdot)$, service invocation $\mathbf{inv}(\cdot,\cdot,\cdot)$ and service request processing $\mathbf{rec}(\cdot,\cdot,\cdot)$, by exploiting operators for conditional choice $\mathbf{if}\ (\cdot)\ \mathbf{then}\ \{\cdot\}\ \mathbf{else}\ \{\cdot\}$ (*switch*), sequential composition $\cdot;\cdot$ (*sequence*), parallel composition $\cdot \mid \cdot$ (*flow*), external choice[1] $\sum_{i\in I} \mathbf{rec}(\cdot,\cdot,\cdot);\cdot$ (*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 $\mathbf{ass}(\bar{w}, \bar{e})$ and $\mathbf{rec}(r, o, \bar{w})$ 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:

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

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 will be described in a little

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

$$
\begin{array}{rcl@{\qquad}rcl}
s; \mathbf{0} & \equiv & s & s + (s' + s'') & \equiv & (s + s') + s'' \\
\mathbf{0}; s & \equiv & s & s + s & \equiv & s \\
(s; s'); s'' & \equiv & s; (s'; s'') & s \mid \mathbf{0} & \equiv & s \\
s + \mathbf{0} & \equiv & s & s \mid s' & \equiv & s' \mid s \\
s + s' & \equiv & s' + s & s \mid (s' \mid s'') & \equiv & (s \mid s') \mid s''
\end{array}
$$

$$
\frac{\text{If } s \text{ and } s' \text{ are } \alpha\text{-equivalent}}{s \equiv s'} \qquad \frac{s \equiv s'}{\_s \mid C \equiv \_s' \mid C}
$$

$$
C \mid m \gg \mathbf{0} \equiv C \qquad C_1 \mid C_2 \equiv C_2 \mid C_1 \qquad (C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3)
$$

$$
\frac{C \equiv C'}{\{a ::^{Op,L} C\} \equiv \{a ::^{Op,L} C'\}} \qquad \frac{N_1 \equiv N_1'}{N_1 \cup N_2 \equiv N_1' \cup N_2}
$$

Table 2: Structural congruence $\equiv$

while), 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.

2.2 Operational semantics

The *operational semantics* of WS-CALCULUS is given in terms of a structural congruence and of a reduction relation over nets. For the sake of simplicity, in this section we model taking place of errors (e.g. when the premises of reduction rules are not satisfied) simply as deadlock. We refer the interested reader to Section 5 for more details on fault throwing and handling exceptions.

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 $\equiv$ and defined by the rules in Table 2, identifies syntactically different terms which intuitively represent the same term. In Table 2, and in the sequel, notation $\_s$ shall denote both service specifications $(*s)$ and service instances $(m \gg s)$. At the level of services, the structural congruence states that: the sequence operator is associative and has $\mathbf{0}$ as identity element (law $\mathbf{0}; s \equiv s$ is exploited to enable a new activity when a 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 remaining rules of Table 2 extend the structural congruence to components and nets, and should be self-explicative. 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 $\rightarrowtail$, 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

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

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

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

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

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

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

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

Table 3: WS-CALCULUS operational semantics: instances

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:

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

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

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

Finally, the last two auxiliary functions we need are:

- $P(\bar{w})$ returns the set of properties contained in $\bar{w}$ and is defined as follows:

$$\begin{aligned} &P(v) = \emptyset \\ &P(p) = \{p\} \\ &P(\langle w_1, \ldots, w_n \rangle) = P(w_1) \ \cup \ \ldots \ \cup \ P(w_n) \end{aligned}$$

- $eR(C)$ returns the set of activities $\mathbf{rec}\,(\cdot,\cdot,\cdot)$ that are start activities of C and is defined inductively on the syntax of C as follows:

$$\begin{aligned} &eR(\_s) = eR(s) \\ &eR(\langle a, o, \bar{u} \rangle) = \emptyset \\ &eR(C_1 \mid C_2) = eR(C_1) \cup eR(C_2) \\ &eR(\mathbf{0}) = \emptyset \\ &eR(\mathbf{exit}) = \emptyset \\ &eR(\mathbf{rec}\,(r, o, \bar{w})) = \{\mathbf{rec}\,(r, o, \bar{w})\} \\ &eR(\mathbf{inv}\,(r, o, \bar{w})) = \emptyset \\ &eR(\mathbf{ass}\,(\bar{v}, \bar{e})) = \emptyset \\ &eR(\mathbf{if}\ (e)\ \mathbf{then}\ \{s\}\ \mathbf{else}\ \{s\}) = \emptyset \\ &eR(s_1; s_2) = eR(s_1) \\ &eR(s_1 \mid s_2) = eR(s_1) \cup eR(s_2) \\ &eR(\textstyle\sum_{i\in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i) = \{\mathbf{rec}\,(r_i, o_i, \bar{w}_i) \mid i \in I\} \\ &eR(A(\bar{w})) = eR(s) \quad \text{if } A(\bar{v} : \bar{\tau}^{\star}) \stackrel{def}{=} s \end{aligned}$$

The labelled transition $\xrightarrow{\alpha}$ is the least relation over service instances induced by the rules in Table 3. 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

[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 replaced 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.

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

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

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

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

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

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

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

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

Table 4: WS-CALCULUS operational semantics: nets

receive activities simultaneously 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 4. 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 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

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 5: Type syntax

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 5. 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 (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 $\mathbf{rec}\,(,,)$). 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

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

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

Table 6: Inference rules for $\Gamma \vdash N : \textsc{bnet}$

activity that syntactically precedes the service call. Moreover, notation $\_s$ shall denote both service specifications ($*s$) and service instances ($m \gg s$).

3.1 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 $\textsc{bserv}$. 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 4.1 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 6, 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, 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 7 and 8, 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 8, since the rules in Table 7 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 transmitted. 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:

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

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

Condition $\tau_i \cap Op = \emptyset$ avoids that the service executing the activity can receive its same address as an argument. The remaining cases that define $envExt_{\_,\_}(\cdot)$ are the following:

- $envExt_{\Gamma,a}(\mathbf{0}) = \emptyset$
- $envExt_{\Gamma,a}(\mathbf{exit}) = \emptyset$
- $envExt_{\Gamma,a}(\mathbf{inv}\,(r, o, \bar{w})) = \emptyset$
- Given $G_1 = envExt_{\Gamma,a}(s_1)$ and $G_2 = envExt_{\Gamma,a}(s_2)$, we have:

$$envExt_{\Gamma,a}(\mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\}) = \begin{cases} G & \text{if } G_1 = G_2 = G \\ \mathbf{undef} & \text{otherwise} \end{cases}$$

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

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

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

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

Table 7: Inference rules for $\Gamma \vdash^L_a S : t$

- $envExt_{\Gamma,a}(s_1; s_2) = envExt_{(\Gamma, envExt_{\Gamma,a}(s_1)),a}(s_2)$
- $envExt_{\Gamma,a}(s_1 \mid s_2) = envExt_{\Gamma,a}(s_1) \cup envExt_{\Gamma,a}(s_2)$
- Given $G_i = envExt_{\Gamma,a}(\mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i)$ for $i \in I$, we have:

$$envExt_{\Gamma,a}(\sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i) = \begin{cases} G & \text{if } \forall\, i \in I \text{ it holds that } G_i = G \\ \textbf{undef} & \text{otherwise} \end{cases}$$

- $envExt_{\Gamma,a}(A(\bar{w})) = A : \textsc{bserv}$

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).

3.2 Type soundness

The major property of our type system is that if a net typechecks then it does never generate runtime errors (Corollary 3.5). 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 3.3), and then proving *type safety*, namely that well-typed nets do not immediately generate errors (Theorem 3.4).

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 by the following grammar:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 8: Inference rules for $\Gamma \vdash_a^L S : t$ (cont.)

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:

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

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 3.1 *Suppose* $\Gamma \vdash^L_a s : \textsc{bserv}$. *If* $\Gamma \vdash^L_a v : \tau$, $\Gamma \vdash^L_a u : \tau'$ *and* $\tau \sqsubseteq \tau'$, *then* $\Gamma \vdash^L_a s \cdot (v \mapsto u) : \textsc{bserv}$.

Proof: The proof in case of a substitution for a basic variable is easy, then we consider only the most relevant case of a substitution for a partner link. Thus, we have $(v \mapsto u) = (l \mapsto a')$, $\tau = Op$ and $\tau' = Op'$.

We proceed by induction on the length of the proof of $\Gamma \vdash^L_a s : \textsc{bserv}$, and case analysis on the last step. The following cases

- $s = \mathbf{0}$
- $s = \mathbf{exit}$
- $s = \mathbf{inv}\,(r, o, \bar{w}_{i\in I})$ with $l \notin \{r\} \cup \{w_j \mid j \in pl(\bar{w}_{i\in I})\}$
- $s = \mathbf{rec}\,(r, o, \bar{w}_{i\in I})$ with $r \neq \ulcorner l \urcorner$
- $s = \mathbf{ass}\,(\bar{w}_{i\in I}, \bar{e}_{i\in I})$ with $l \notin \{e_j \mid j \in pl(\bar{e}_{i\in I})\}$

are trivial. Indeed, since s does not contain a free occurrence of l, we have $s \cdot (l \mapsto a') = s$, so we can conclude $\Gamma \vdash^L_a s \cdot (l \mapsto a') : \textsc{bserv}$.

The other base cases are the following:

- $s = \mathbf{inv}\,(l, o, \bar{w}_{i\in I})$ with $l \notin \{w_j \mid j \in pl(\bar{w}_{i\in I})\}$
  If $\Gamma \vdash^L_a s : \textsc{bserv}$ is inferred by rule (*inv*), we have $o : \bar{\tau}'' \in Op$. Since $Op \subseteq Op'$, we have $o : \bar{\tau}'' \in Op'$, then, by (*inv*), $\Gamma \vdash^L_a \mathbf{inv}\,(a', o, \bar{w}_{i\in I}) : \textsc{bserv}$. Otherwise, $\Gamma \vdash^L_a s : \textsc{bserv}$ is inferred by rule (*inv_cb*), hence $o : \bar{\tau}'' \in Op''$, where $\Gamma \vdash^L_a a : Op''$. Then, by (*inv_cb*), $\Gamma \vdash^L_a \mathbf{inv}\,(a', o, \bar{w}_{i\in I}) : \textsc{bserv}$.
- $s = \mathbf{inv}\,(r, o, \langle \ldots l \ldots \rangle)$ with $l \neq r$
  By both (*inv*) and (*inv_cb*), we have that the operation o has type $\bar{\tau}'' = \langle \ldots Op'' \ldots \rangle$ such that $Op'' \subseteq Op$. Since $Op \subseteq Op'$, we have $Op'' \subseteq Op'$ and, by (*inv*) or (*inv_cb*), $\Gamma \vdash^L_a \mathbf{inv}\,(r, o, \langle \ldots a' \ldots \rangle) : \textsc{bserv}$.

- $s = \mathbf{inv}\,(l, o, \langle \ldots l \ldots \rangle)$
  We obtain the thesis combining the proofs of the two previous cases.

- $s = \mathbf{rec}\,(\ulcorner l \urcorner, o, \bar{w})$
  If $\Gamma \vdash^L_a s$: BSERV is inferred by rule (*rec*), we can directly infer $\Gamma \vdash^L_a \mathbf{rec}\,(a', o, \bar{w})$: BSERV. Otherwise, $\Gamma \vdash^L_a s$: BSERV is inferred by (*rec_cb*), then we have $o : \bar{\tau}'' \in Op$. Since $Op \subseteq Op'$, we have $o : \bar{\tau}'' \in Op'$. Thus, we can infer $\Gamma \vdash^L_a \mathbf{rec}\,(a', o, \bar{w})$: BSERV.

- $s = \mathbf{ass}\,(\bar{w}, \langle \ldots l \ldots \rangle)$
  By rule (*ass*), we can directly infer $\Gamma \vdash^L_a \mathbf{ass}\,(\bar{w}, \langle \ldots a' \ldots \rangle)$: BSERV.

Finally, the remaining cases follow by induction. For instance, if $s = s_1 \mid s_2$, from $\Gamma \vdash^L_a s_1 \mid s_2$: BSERV by the first premise of (*flow*) we have $\Gamma \vdash^L_a s_1$: BSERV and by inductive hypothesis we have $\Gamma \vdash^L_a s_1 \cdot (l \mapsto a')$: BSERV. By the second premise of (*flow*), we have $\Gamma \vdash^L_a s_2$: BSERV and by inductive hypothesis we have $\Gamma \vdash^L_a s_2 \cdot (l \mapsto a')$: BSERV. Then, by (*flow*), $\Gamma \vdash^L_a s_1 \cdot (l \mapsto a') \mid s_2 \cdot (l \mapsto a')$: BSERV, i.e. $\Gamma \vdash^L_a (s_1 \mid s_2) \cdot (l \mapsto a')$: BSERV.

□

Lemma 3.2 *If* $\Gamma \vdash^L_a s$: BSERV *and* $m \vdash s \xrightarrow{\alpha} s'$ *then* $\Gamma' \vdash^L_a s'$: 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* $s \equiv C[\mathbf{ass}\,(\bar{w}, \bar{e})]$.

Proof: The proof is by induction on the depth of the shortest inference of $\xrightarrow{\alpha}$. The base cases (of depth 1) are trivial; when the applied rules are (*Exit*), (*Assign*), (*Invoke*) and (*Receive*), we have $s' = \mathbf{0}$; thus, we can easily conclude $\Gamma' \vdash^L_a \mathbf{0}$: BSERV by rules (*nil*) and (*weak*).
For the inductive step, we reason by case analysis on the last rule applied in the inference:

- (*Sequence*): In this case $s \equiv s_1; s_2$, $m \vdash s_1 \xrightarrow{\alpha} s_1'$ and $s' \equiv s_1'; s_2$. Since s is well-typed, by rule (*seq*) we have $\Gamma \vdash^L_a s_1$: BSERV and $\Gamma'' \vdash^L_a s_2$: BSERV for $\Gamma'' = \Gamma, envExt_{\Gamma,a}(s_1)$. By inductive hypothesis, we have $\Gamma' \vdash^L_a s_1'$: BSERV for some $\Gamma' \leqslant \Gamma$. In particular, we can choose Γ' such that $\Gamma'' = \Gamma', envExt_{\Gamma,a}(s_1')$. Then, by rule (*seq*), we have the thesis.

- (If_{ff}) and (If_{tt}): In case of $m \rhd e = \mathbf{true}$ we have $s \equiv \mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\}$, $m \vdash s_1 \xrightarrow{\alpha} s'$. Since s is well-typed, by rule (*flow*) we have $\Gamma \vdash^L_a s_1$: BSERV. So, by inductive hypothesis, we have straightaway the thesis. In case of $m \rhd e = \mathbf{false}$, we proceed in similar way.

- (*Pick*): In this case $s \equiv \mathbf{rec}\,(r, o, \bar{w})\,; s'' + \sum_{i \in I} \mathbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i$, $m \vdash \mathbf{rec}\,(r, o, \bar{w})\,; s'' \xrightarrow{\alpha} s'$. Since s is well-typed, by rule (*pick*) we have $\Gamma \vdash^L_a \mathbf{rec}\,(r, o, \bar{w})\,; s''$: BSERV. So, by inductive hypothesis, we straightaway have the thesis.

- (*Flow*) and ($Flow_{Rec}$): In this case $s \equiv s_1 \mid s_2$, $m \vdash s_1 \xrightarrow{\alpha} s_1'$ and $s' \equiv s_1' \mid s_2$. Since s is well-typed, by rule (*flow*) we have $\Gamma \vdash^L_a s_1$: BSERV. By inductive hypothesis, we have $\Gamma' \vdash^L_a s_1'$: BSERV for some $\Gamma' \leqslant \Gamma$. By rules (*flow*) and (*weak*), $\Gamma' \vdash^L_a s_2$: BSERV holds and then, by (*flow*), we have the thesis.

[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 $=$.

- (*Call*): In this case $s \equiv A(\bar{c})$ with $A(\bar{v}_{i\in I} : \bar{\tau}^{\star}_{i\in I}) \stackrel{def}{=} s''$ and $m \vdash s'' \cdot (\bar{v} \mapsto m \rhd \bar{c}) \xrightarrow{\alpha} s'$. Since s is well-typed, by rules (*call*) and (def_1) we have $\Gamma'' \vdash^{L'}_{a} s''$: BSERV holds for $\Gamma'' = (\Gamma, A :$ BSERV, $\bar{v}_{j\in J} : \bar{\tau}^{\star}_{j\in J})$ and $L' = L \cup \{\bar{v}_{i\in(I\setminus J)} : \bar{\tau}^{\star}_{i\in(I\setminus J)}\}$ with $J = pl(\bar{v}_{i\in I})$. By Lemma 3.1 and by the fact that if we substitute in a well-typed service a partner link having type $\star$ with any address then the service is still well-typed (because the partner link is used only to perform a callback), $\Gamma'' \vdash^{L'}_{a} s'' \cdot (\bar{v} \mapsto m \rhd \bar{c})$: BSERV holds and, since the occurences of $\bar{v}$ in s'' are replaced by $m \rhd \bar{c}$, we have that $\Gamma'' \vdash^{L}_{a} s'' \cdot (\bar{v} \mapsto m \rhd \bar{c})$: BSERV holds. Thus, by inductive hypothesis, we directly have the thesis.

□

Theorem 3.3 (Subject Reduction) *If* $\Gamma \vdash N :$ BNET *and* $N \rightarrowtail N'$ *then* $\Gamma' \vdash N' :$ BNET *for some* Γ'.

Proof: Given $N \equiv \{a_i ::^{Op_i,L_i} C_i \mid i \in I\}$, by the rule (*net*) it does exists an environment $\Gamma'' = \Gamma, \{a_j : Op_j \mid j \in I\}$ such that for all $i \in I$ we have $\Gamma'' \vdash a_i ::^{Op_i,L_i} C_i :$ BNET. By rule (*netToServ*), (*par*) and (*serv*), we have that $\forall \_s \in C_i$ the judgment $\Gamma'' \vdash^{L_i}_{a_i} s :$ BSERV holds, i.e. the service of each instance located in some node of the net is well-typed under the environment Γ'' and under its address and local declarations. Moreover, $\forall \langle a, o, \bar{u}\rangle \in C_i$ the judgment $\Gamma'' \vdash^{L_i}_{a_i} \langle a, o, \bar{u}\rangle :$ BSERV holds, i.e. all service requests located in the net nodes are well-typed.
Looking at the definitions in Table 4 we proceed by rule induction. We consider the following base cases:

- (*Invoke*) and (*Terminate*): In these cases we see that for each net reduction $N \rightarrowtail N'$ there is, at service level, a corresponding transition $m \vdash s \xrightarrow{\alpha} s'$, for some component $\_s$ in a node of N. In case of (*Invoke*), after the transition there will be a new service request in the data space of some node of N'. This tuple is well-typed, because rule (*req*) has the same requirements of rules for invoke activities. Thus, since s is well-typed, by Lemma 3.2 and applying in reverse order the rules (*netToServ*), (*par*), (*serv*) and (*net*), we obtain the thesis.

- (*Assign*): We have $N \equiv \{a ::^{Op,L} m \gg s \mid C\}$, $\Gamma'' \vdash^{L}_{a} s :$ BSERV, $m \vdash s \xrightarrow{\bar{w}:=\bar{u}} s'$, $N' \equiv \{a ::^{Op,Loc} (m \cup m') \gg s' \cdot \sigma \mid C\}$ with $m \rhd (\bar{w} := \bar{u}) = \langle m', \sigma\rangle$.
  By Lemma 3.2, we have $\Gamma' \vdash^{L}_{a} s' :$ BSERV for $\Gamma' \leqslant \Gamma''$ such that for each association $(l \mapsto a')$ in σ there is a corresponding $(l : Op)$ in Γ' with $Op \subseteq Op'$, with $\Gamma'' \vdash^{L}_{a} a' : Op'$. Moreover, for each association $(b \mapsto u)$ in σ, the types of b and u match, because (*ass*) checks that basic variables and values within an assign activity have the same type. Then, we can apply Lemma 3.1 and obtain $\Gamma' \vdash^{L}_{a} s' \cdot \sigma :$ BSERV. Now, applying in reverse order the rules (*netToServ*), (*par*), (*serv*) and (*net*), we have the thesis.

- ($Receive_{aI}$), ($Receive_{lI}$), ($Receive_{aS}$) and ($Receive_{lS}$): we proceed similarly to the previous case.

The cases for (*Part*) and (*Cong*) follow by induction.

□

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. Rules defining $\rightarrowtail^{err}$ are shown in Table 9. Here, we linger on the most significant rules. 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*) 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

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

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

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

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

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

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

$$\frac{\begin{array}{c} s \equiv C[\mathbf{inv}\,(a', o, \bar{w})] \quad \bar{w} = \langle w_1, \ldots, w_n \rangle \quad w_1 : \tau_1 \in L \ \ldots\ w_n : \tau_n \in L \\ [(o : \bar{\tau} \in Op \ \wedge\ \bar{\tau} \not\sqsubseteq \langle \tau_1, \ldots, \tau_n \rangle) \vee (o : \bar{\tau}' \in Op' \ \wedge\ \bar{\tau}' \not\sqsubseteq \langle \tau_1, \ldots, \tau_n \rangle)] \end{array}}{\{a ::^{Op,L} \_s \mid C,\ a' ::^{Op',L'} C'\} \rightarrowtail^{err}} \quad (varError1)$$

$$\frac{\begin{array}{c} s \equiv C_r[\mathbf{rec}\,(l, o, \bar{w})] \quad o : \bar{\tau} \in Op \quad \bar{w} = \langle w_1, \ldots, w_n \rangle \\ w_1 : \tau_1 \in L \ \ldots\ w_n : \tau_n \in L \quad \bar{\tau} \not\sqsubseteq \langle \tau_1, \ldots, \tau_n \rangle \end{array}}{\{a ::^{Op,L} \_s \mid C\} \rightarrowtail^{err}} \quad (varError2)$$

$$\frac{\begin{array}{c} s \equiv C_r[\mathbf{rec}\,(a', o, \bar{w})] \quad \bar{w} = \langle w_1, \ldots, w_n \rangle \quad w_1 : \tau_1 \in L \ \ldots\ w_n : \tau_n \in L \\ [(o : \bar{\tau} \in Op \ \wedge\ \bar{\tau} \not\sqsubseteq \langle \tau_1, \ldots, \tau_n \rangle) \vee (o : \bar{\tau}' \in Op' \ \wedge\ \bar{\tau}' \not\sqsubseteq \langle \tau_1, \ldots, \tau_n \rangle)] \end{array}}{\{a ::^{Op,L} \_s \mid C,\ a' ::^{Op',L'} C'\} \rightarrowtail^{err}} \quad (varError3)$$

$$\frac{s \equiv C[\mathbf{ass}\,(\bar{w}, \bar{u})] \quad \exists\, i\,.\, w_i : bt \quad u_i \notin typeSet(bt)}{\{a ::^{Op,L} \_s \mid C\} \rightarrowtail^{err}} \quad (assError)$$

$$\frac{N_1 \rightarrowtail^{err}}{N_1 \cup N_2 \rightarrowtail^{err}} \quad (partError) \qquad \frac{N \equiv N' \quad N \rightarrowtail^{err}}{N' \rightarrowtail^{err}} \quad (congError)$$

Table 9: Runtime errors

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). Rule (*rrError2*) raises an error if the

first argument of a callback invoke is initialized by an assignment rather than by a triggering receive. Finally, rule (*assError*) raises an error if the type of an evaluated expression is not conform to the type of corresponding basic variable. We use the auxiliary function $typeSet(\cdot)$ that, given a basic type, returns the corresponding value set and is defined as follows:

- $typeSet(\textsc{Int}) = \texttt{Int}$
- $typeSet(\textsc{Bool}) = \{\mathbf{true}, \mathbf{false}\}$
- $typeSet(\textsc{Str}) = \texttt{Str}$

Some obvious facts about deductions that we use in the proof of the following theorem are:

- if $\Gamma \vdash^L_a C_g[s] : \textsc{bserv}$ then there exists Γ' such that $\Gamma' \leqslant \Gamma$ and $\Gamma' \vdash^L_a s : \textsc{bserv}$;
- if there is no Γ' such that $\Gamma' \vdash^L_a s : \textsc{bserv}$, then there are no Γ such that $\Gamma' \leqslant \Gamma$ and $\Gamma \vdash^L_a C_g[s] : \textsc{bserv}$.

These follow from the facts that there is exactly one inference rule for each service form, and each inference rule requires a proof for each subterm of the service in its conclusion. Replacing C_g by C or C_r the facts still hold.

We can now prove the type safety theorem.

Theorem 3.4 (Type Safety) $\Gamma \vdash N : \textsc{bnet}$ *implies that* $N \rightarrowtail^{err}$ *holds false.*

Proof: We prove, by rule induction, that if $N \rightarrowtail^{err}$ then there is no Γ for which $\Gamma \vdash N : \textsc{bnet}$ can be derived.
Looking at the definition in Table 9, we see that N can have one of the following forms:

$$N_1 \equiv \{a ::^{Op,L} \_s \mid C,\ a' ::^{Op',L'} C'\}$$

$$N_2 \equiv \{a ::^{Op,L} \_s \mid C\}$$

Moreover, we have that the following base cases have to be considered.

- (*opDefError1*): N has the form N_1 with $s \equiv C[\mathbf{inv}\,(a', o, \bar{w})]$ and o undefined in Op and Op'. Suppose that for some environment Γ we can derive $\Gamma \vdash N : \textsc{bnet}$. By rule (*net*) it does exists an environment Γ' such that $\Gamma' = (\Gamma, a' : Op')$ and $\Gamma' \vdash (a ::^{Op,L} \_s \mid C) : \textsc{bnet}$. By rule (*netToServ*), (*par*) and (*serv*) we have $\Gamma' \vdash^L_a s : \textsc{bserv}$.
  Since C is an execution context, $\Gamma'' \vdash^L_a \mathbf{inv}\,(a', o, \bar{w}) : \textsc{bserv}$ must hold for some $\Gamma'' \leqslant \Gamma'$. This can be inferred only by rules (*inv_cb*) and (*inv*). But, rule (*inv_cb*) cannot be applied, because $\Gamma'' \vdash^L_a a : Op$, such that $o : \bar{\tau} \in Op$, is not satisfied. Similarly, rule (*inv*) cannot be applied, because $\Gamma'' \vdash^L_a a' : Op'$, such that $o : \bar{\tau} \in Op'$, is not satisfied.

- (*opDefError2*): N has the form N_2 with $s \equiv C_r[\mathbf{rec}\,(l, o, \bar{w})]$ and o undefined in Op. Suppose that for some environment Γ we can derive $\Gamma \vdash N : \textsc{bnet}$. By rule (*net*), we have $\Gamma \vdash (a ::^{Op,L} \_s \mid C) : \textsc{bnet}$ and, by rule (*netToServ*), (*par*) and (*serv*) we have $\Gamma \vdash^L_a s : \textsc{bserv}$.
  Since C_r is an execution context, $\Gamma' \vdash^L_a \mathbf{rec}\,(l, o, \bar{w}) : \textsc{bserv}$ must hold for some $\Gamma' \leqslant \Gamma$. This can be inferred only by rules (*rec*) and (*rec_cb*). But the rule (*rec*) cannot be applied, because $\Gamma' \vdash^L_a a : Op$, such that $o : \bar{\tau} \in Op$, is not satisfied. Moreover, the fact that the first argument of the receive is a partner link means that l is unitialized, i.e. no association for l can be in Γ'. Then, the rule (*rec_cb*) cannot be applied because $\Gamma' \vdash^L_a l : Op$ is not satisfied.

- (*opDefError3*): N has the form N_1 with $s' \equiv C_r[\mathbf{rec}\,(a', o, \bar{w})]$ and o undefined in Op and Op'. Similarly to the previous case, $\Gamma' \vdash_a^L \mathbf{rec}\,(a', o, \bar{w}) : \text{BSERV}$ must hold and this can be inferred only by rules (*rec*) and (*rec_cb*). Similarly to (*opDefError1*), we can prove that these rules cannot be applied.

- (*linkError*): N has the form N_2 with $s \equiv C_r[\mathbf{rec}\,(l, o, \bar{w})\,; C_g[\mathbf{inv}\,(l', o', \bar{w}')]]$ and $\exists\, i\ .\ w_i' = l$. Suppose for some Γ we can infer $\Gamma \vdash N : \text{BNET}$. Thus, $\Gamma' \vdash_a^L \mathbf{rec}\,(l, o, \bar{w}) : \text{BSERV}$ must hold for some $\Gamma' \leqslant \Gamma$. Then, by rule (*seq*), $\Gamma'' \vdash_a^L C_g[\mathbf{inv}\,(l', o, \bar{w}')] : \text{BSERV}$ must hold, for some Γ'' such that $\Gamma'' \leqslant \Gamma'$. By $envExt_{\_}(\cdot)$ definition, we have that $\Gamma'' \vdash_a^L l : \star$ holds. Since $\star$ is not a message type, rules (*inv*) and (*inv_cb*) cannot be applied because the premise $\Gamma'' \vdash_a^L \bar{w}' : \bar{\tau}'$ is not satisfied.

- (*rrError1*): N has the form N_2 with $s \equiv C[\mathbf{inv}\,(l, o, \bar{w})]$ and $o : \bar{\tau} \in Op$. Similarly to (*opDefError2*), $\Gamma' \vdash_a^L \mathbf{inv}\,(l, o, \bar{w}) : \text{BSERV}$ must hold and, since o is defined in Op, this can be inferred only by rule (*inv_cb*). Since the first argument of the invoke is a partner link, a previous triggering receive doesn't be performed, then $\Gamma' \vdash_a^L l : \star$ is not satisfied.

- (*rrError2*): N has the form N_2 with $s \equiv C[\mathbf{ass}\,(\bar{w}, \bar{u})\,; C_g[\mathbf{inv}\,(l, o, \bar{w}')]]$, $o : \bar{\tau} \in Op$ and $\exists\, i\ .\ w_i = l$. Suppose for some Γ we can infer $\Gamma \vdash_a^L \mathbf{ass}\,(\bar{w}, \bar{u}) : \text{BSERV}$. Then, by rule (*seq*), $\Gamma' \vdash_a^L C_g[\mathbf{inv}\,(l, o, \bar{w}')] : \text{BSERV}$ must hold, for some Γ' and Op' such that $\Gamma' \leqslant (\Gamma, l : Op')$. Similarly to the previous case, (*inv_cb*) cannot be used to infer the type of the invoke activity, because, $\Gamma' \vdash_a^L l : \star$ is not satisfied.

- (*varError1*): N has the form N_1 with $s \equiv C[\mathbf{inv}\,(a', o, \bar{w})]$, $\bar{w} = \langle w_1, \ldots, w_n \rangle$, $w_1 : \tau_1 \in Op, \ldots, w_n : \tau_n \in Op$. Similarly to (*opDefError1*), $\Gamma'' \vdash_a^L \mathbf{inv}\,(a', o, \bar{w}) : \text{BSERV}$ must hold for some environment Γ''. The predicate $\rightarrowtail^{err}$ holds if one of these cases holds:

  - $(o : \bar{\tau} \in Op \ \wedge\ \bar{\tau} \not\sqsubseteq \langle \tau_1, \ldots, \tau_n \rangle)$
    Here we have to apply the rule (*inv_cb*). But it isn't possible, because $\Gamma'' \vdash_a^L \bar{w}) : \langle \tau_1, \ldots, \tau_n \rangle$ and $\bar{\tau} \sqsubseteq \langle \tau_1, \ldots, \tau_n \rangle$ cannot hold both.
  - $(o : \bar{\tau}' \in Op' \ \wedge\ \bar{\tau}' \not\sqsubseteq \langle \tau_1, \ldots, \tau_n \rangle)$
    Here we have to apply the rule (*inv*). But it isn't possible, for the same reason of the previous case.

- (*varError2*): N has the form N_2 with $s' \equiv C_r[\mathbf{rec}\,(l, o, \bar{w})]$, $o : \bar{\tau} \in Op$, $\bar{w} = \langle w_1, \ldots, w_n \rangle$, $w_1 : \tau_1 \in Op, \ldots, w_n : \tau_n \in Op$. Similarly to (*opDefError2*), $\Gamma' \vdash_a^L \mathbf{rec}\,(l, o, \bar{w}) : \text{BSERV}$ must hold. Since o is defined in Op, this judgement can be inferred only by rule (*rec*). But the rule cannot be applied, because the premise $\Gamma \vdash_a^L \bar{w}_{j \in (\{1..n\} \setminus J)} : \bar{\tau}_{j \in (\{1..n\} \setminus J)}$, with $J = pl(\bar{w}_{i \in \{1..n\}})$, is not satisfied.

- (*varError3*): this case is similar to (*varError1*), with the difference that here the concerned activity is a receive and the corresponding rules for its type inference are (*rec*) and (*rec_cb*).

- (*assError*): N has the form N_2 with $s' \equiv C[\mathbf{ass}\,(\bar{w}, \bar{u})]$ and $\exists\, i \in \{1..n\}\ .\ w_i : bt$ such that $u_i \notin typeSet(bt)$. Similarly to the previous cases, $\Gamma' \vdash_a^L \mathbf{ass}\,(\bar{w}, \bar{u}) : \text{BSERV}$ must hold. The only rule that can be applied is (*ass*). But it is not possible, because the premise $\Gamma \vdash_a^L \bar{w}_{j \in (\{1..n\} \setminus J)} : \bar{\tau}_{j \in (\{1..n\} \setminus J)}$, where $\bar{\tau}_{i \in \{1..n\}}$ is the type of $\bar{u}$ and $J = pl(\bar{w}_{i \in \{1..n\}})$, is not satisfied.

The cases of (*partError*) and (*congError*) follow by induction. Suppose, for example, that $N_1 \cup N_2 \rightarrowtail^{err}$ because $N_1 \rightarrowtail^{err}$. By induction N_1 is not typeable and therefore we cannot use (*net*), the only possible rule, to infer that $N_1 \cup N_2$ is well-typed.

□

To conclude, we have ($\rightarrowtail^*$ denotes the reflexive and transitive closure of $\rightarrowtail$)

Corollary 3.5 (Type Soundness) *Let* $\Gamma \vdash N : \textsc{bnet}$. *Then* $N' \rightarrowtail^{err}$ *holds false for every net* N' *such that* $N \rightarrowtail^* N'$.

Proof: Subject Reduction implies that $\Gamma \vdash N' : \textsc{bnet}$ and therefore we can apply Type Safety to obtain $N' \rightarrowtail^{err}$ holds false.

□

4 Examples

In this section we show two application of our framework. The former is an example of a well-known interaction pattern, where a process receives a request and delegates another process to reply, the latter is an example from WS-BPEL specification, where the interaction pattern is the traditional request-response.

4.1 A brokerage scenario

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 $\Gamma \vdash^{L_c}_{a_c} \langle a_c, o_{init}, 10\rangle : \textsc{bserv}$, $\Gamma \vdash^{L_c}_{a_c} s_c : \textsc{bserv}$, $\Gamma \vdash^{L_b}_{a_b} s_b : \textsc{bserv}$ and $\Gamma \vdash^{L_r}_{a_r} s_r : \textsc{bserv}$ hold. This is what the inferences in Tables 10, 11, 12 and 13, respectively, show. For the sake of presentation, the inferences are split in a few parts with references between them.

Comments on inference for the client service are in order. 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. $\bar{\tau} \sqsubseteq \langle \textsc{Int}, Op_c\rangle$), because $\{o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle\} \subseteq Op_c$.

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 $o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle \notin Op_c$. 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, $\Gamma_c \vdash^{L_c}_{a_c} \mathbf{rec}\,(l_r, o_{cb}, \langle p, res\rangle) : \textsc{bserv}$ cannot be inferred.

4.2 Shipping service

This example presents the use of a ws-calculus process to describe a rudimentary shipping service (from Subsection 16.1 of WS-BPEL specification). This service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The shipping service offers two types of shipment: shipments where the items are held and shipped together and shipment where the items are shipped piecemeal until all of the order is accounted for.

$$\dfrac{\dfrac{\dfrac{}{a_c : Op_c \vdash^{L_c}_{a_c} a_c : Op_c}\,(ref)}{\Gamma \vdash^{L_c}_{a_c} a_c : Op_c}\,(weak) \qquad o_{init} : \langle \textsc{Int} \rangle \in Op_c \qquad \dfrac{\dfrac{10 \in \texttt{Int}}{\emptyset \vdash^{L_c}_{a_c} 10 : \textsc{Int}}\,(int)}{\Gamma \vdash^{L_c}_{a_c} 10 : \textsc{Int}}\,(weak) \qquad \textsc{Int} \sqsubseteq \textsc{Int}}{\Gamma \vdash^{L_c}_{a_c} \langle a_c, o_{init}, 10 \rangle : \textsc{bserv}}\,(req)$$

Table 10: Type inference for the service request

This service is modelled by the node $a ::^{Op,L} *s$ where the service specification is defined as follows:

$$\begin{aligned}
s &\triangleq \mathbf{rec}\,(cust, shipReq, \overline{req})\,; \\
&\quad \mathbf{if}\ (req_2)\ \mathbf{then}\ \{s_{then}\}\ \mathbf{else}\ \{s_{else}\} \\
\\
s_{then} &\triangleq \mathbf{ass}\,(res_2, req_3)\,; \mathbf{inv}\,(cust, shipNot, \overline{res}) \\
\\
s_{else} &\triangleq \mathbf{ass}\,(itemsShipped, 0)\,; \\
&\quad B(\langle itemsShipped, req_3, req_1, cust \rangle) \\
\\
B(\overline{par} : \bar{\tau}^{\star}_{B}) &\stackrel{def}{=} \mathbf{ass}\,(itemC, rand())\,; \\
&\quad \mathbf{inv}\,(l_c, shipNot, \langle ordId, itemC \rangle)\,; \\
&\quad \mathbf{ass}\,(itemsShipped', items + itemC)\,; \\
&\quad \mathbf{if}\ (itemsShipped' < tot)\ \mathbf{then} \\
&\quad \{B(\langle itemsShipped', tot, ordId, l_c \rangle)\}\ \mathbf{else}\ \{\mathbf{0}\}
\end{aligned}$$

where:

- *cust* is the partner link used by the service to communicate with its customers;
- *shipReq* and *shipNot* are the operations used to receive the shipping request and to send shipping notices, respectively;
- $\overline{req} = \langle p\_orderId, complete, itemsTot \rangle$ is the tuple of parameters used for the request shipping message; each instance of the shipping service is uniquely identified by the correlation set composed by the property *p_orderId*;
- $\overline{res} = \langle p\_orderId, itemsCount \rangle$ is the tuple of parameters used for the response message; we notice that the property *p_orderId* is used to fill the first field of the response message;
- *itemsShipped* is an integer variable used as counter;
- $\overline{par} \triangleq \langle items, tot, ordId, l_c \rangle$ is the tuple of formal parameters of service definition with identifier B and $\bar{\tau}^{\star}_{B} \triangleq \langle \textsc{Int}, \textsc{Int}, \textsc{Int}, \star \rangle$ is the tuple of their types;
- *rand*() is a function which returns a random integer number and represents an internal interaction with back-end system (for the sake of simplicity, we don't describe this interaction).

The service type declarations are $Op = \{shipReq : \bar{\tau}_r, shipNot : \bar{\tau}_n\}$, where $\bar{\tau}_r = \langle \textsc{Int}, \textsc{Bool}, \textsc{Int} \rangle$ and $\bar{\tau}_n = \langle \textsc{Int}, \textsc{Int} \rangle$, and $L = \{p\_orderId : \textsc{Int}, complete : \textsc{Bool}, itemsTot : \textsc{Int}, itemsCount : \textsc{Int}, itemsShipped : \textsc{Int}, itemC : \textsc{Int}, itemsShipped' : \textsc{Int}\}$. Notice that the

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

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

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

Table 11: Type inference for the client service s_c (Γ_c is (Γ, $l_{init} : \star$))

$$
\dfrac{
\dfrac{\dfrac{}{a_r : Op_r \vdash^{L_b}_{a_b} a_r : Op_r}\,(\mathit{ref})}{\Gamma_b \vdash^{L_b}_{a_b} a_r : Op_r}\,(\mathit{weak})
\quad o' : \bar{\tau} \in Op_r \quad
\dfrac{
\dfrac{\dfrac{b : \textsc{Int} \in L_b}{\emptyset \vdash^{L_b}_{a_b} b : \textsc{Int}}\,(\mathit{var})}{\Gamma_b \vdash^{L_b}_{a_b} b : \textsc{Int}}\,(\mathit{weak})
\quad
\dfrac{\dfrac{}{l_c : \{o_{cb} : \langle \textsc{Int}, \textsc{Int} \rangle\} \vdash^{L_b}_{a_b} l_c : \{o_{cb} : \langle \textsc{Int}, \textsc{Int} \rangle\}}\,(\mathit{ref})}{\Gamma_b \vdash^{L_b}_{a_b} l_c : \{o_{cb} : \langle \textsc{Int}, \textsc{Int} \rangle\}}\,(\mathit{weak})
}{\Gamma_b \vdash^{L_b}_{a_b} \langle b, l_c \rangle : \langle \textsc{Int}, \{o_{cb} : \langle \textsc{Int}, \textsc{Int} \rangle\}\rangle}\,(\bar{w})
\quad \bar{\tau} \sqsubseteq \langle \textsc{Int}, \{o_{cb} : \langle \textsc{Int}, \textsc{Int} \rangle\}\rangle
}{(1)\ \Gamma_b \vdash^{L_b}_{a_b} \mathbf{inv}\,(a_r, o', \langle b, l_c \rangle) : \textsc{bserv}}\,(\mathit{inv})
$$

$$
\dfrac{
\dfrac{
\dfrac{\dfrac{}{a_b : Op_b \vdash^{L_b}_{a_b} a_b : Op_b}\,(\mathit{ref})}{\Gamma \vdash^{L_b}_{a_b} a_b : Op_b}\,(\mathit{weak})
\quad o : \bar{\tau} \in Op_b \quad
\dfrac{\dfrac{b : \textsc{Int} \in L_b}{\emptyset \vdash^{L_b}_{a_b} b : \textsc{Int}}\,(\mathit{var})}{\Gamma \vdash^{L_b}_{a_b} b : \textsc{Int}}\,(\mathit{weak})
\quad l_c \neq l
}{\Gamma \vdash^{L_b}_{a_b} \mathbf{rec}\,(l, o, \langle b, l_c \rangle) : \textsc{bserv}}\,(\mathit{rec})
\quad (1)\ \Gamma_b \vdash^{L_b}_{a_b} \mathbf{inv}\,(a_r, o', \langle b, l_c \rangle) : \textsc{bserv}
}{\Gamma \vdash^{L_b}_{a_b} \mathbf{rec}\,(l, o, \langle b, l_c \rangle)\,;\mathbf{inv}\,(a_r, o', \langle b, l_c \rangle) : \textsc{bserv}}\,(\mathit{seq})
$$

Table 12: Type inference for the broker service s_b (Γ_b is $(\Gamma,\ l : \star,\ l_c : \{o_{cb} : \langle \textsc{Int}, \textsc{Int} \rangle\})$)

$$\dfrac{\dfrac{\dfrac{}{l_{cb} : \{o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle\} \vdash^{L_r}_{a_r} l_{cb} : \{o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle\}}\,(\textit{ref})}{\Gamma_r \vdash^{L_r}_{a_r} l_{cb} : \{o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle\}}\,(\textit{weak}) \quad o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle \in \{o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle\} \quad \dfrac{\dfrac{\dfrac{b : \textsc{Int} \in L_r}{\emptyset \vdash^{L_r}_{a_r} b : \textsc{Int}}\,(\textit{var})}{\Gamma_r \vdash^{L_r}_{a_r} b : \textsc{Int}}\,(\textit{weak}) \quad \dfrac{\dfrac{b' : \textsc{Int} \in L_r}{\emptyset \vdash^{L_r}_{a_r} b' : \textsc{Int}}\,(\textit{var})}{\Gamma_r \vdash^{L_r}_{a_r} b' : \textsc{Int}}\,(\textit{weak})}{\Gamma_r \vdash^{L_r}_{a_r} \langle b, b'\rangle : \langle \textsc{Int}, \textsc{Int}\rangle}\,(\bar{w}) \quad \langle \textsc{Int}, \textsc{Int}\rangle \sqsubseteq \langle \textsc{Int}, \textsc{Int}\rangle}{(2)\ \Gamma_r \vdash^{L_r}_{a_r} \mathbf{inv}\,(l_{cb}, o_{cb}, \langle b, b'\rangle) : \textsc{bserv}}\,(\textit{inv})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{b : \textsc{Int} \in L_r}{\emptyset \vdash^{L_r}_{a_r} b : \textsc{Int}}\,(\textit{var})}{\Gamma_r \vdash^{L_r}_{a_r} b : \textsc{Int}}\,(\textit{weak}) \quad \dfrac{\dfrac{1 \in \texttt{Int}}{\emptyset \vdash^{L_r}_{a_r} 1 : \textsc{Int}}\,(\textit{int})}{\Gamma_r \vdash^{L_r}_{a_r} 1 : \textsc{Int}}\,(\textit{weak})}{\Gamma_r \vdash^{L_r}_{a_r} b + 1 : \textsc{Int}}\,(\textit{exp}_+) \quad \dfrac{\dfrac{b' : \textsc{Int} \in L_r}{\emptyset \vdash^{L_r}_{a_r} b' : \textsc{Int}}\,(\textit{var})}{\Gamma_r \vdash^{L_r}_{a_r} b' : \textsc{Int}}\,(\textit{weak})}{\Gamma_r \vdash^{L_r}_{a_r} \mathbf{ass}\,(b', b + 1) : \textsc{bserv}}\,(\textit{ass}) \quad (2)\ \Gamma_r \vdash^{L_r}_{a_r} \mathbf{inv}\,(l_{cb}, o_{cb}, \langle b, b'\rangle) : \textsc{bserv}}{(1)\ \Gamma_r \vdash^{L_r}_{a_r} \mathbf{ass}\,(b', b + 1)\,;\mathbf{inv}\,(l_{cb}, o_{cb}, \langle b, b'\rangle) : \textsc{bserv}}\,(\textit{seq})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a_r : Op_r \vdash^{L_r}_{a_r} a_r : Op_r}\,(\textit{ref})}{\Gamma \vdash^{L_r}_{a_r} a_r : Op_r}\,(\textit{weak}) \quad o' : \bar{\tau} \in Op_r \quad \dfrac{\dfrac{b : \textsc{Int} \in L_r}{\emptyset \vdash^{L_r}_{a_r} b : \textsc{Int}}\,(\textit{var})}{\Gamma \vdash^{L_r}_{a_r} b : \textsc{Int}}\,(\textit{weak}) \quad l_{cb} \neq l}{\Gamma \vdash^{L_r}_{a_r} \mathbf{rec}\,(l, o', \langle b, l_{cb}\rangle) : \textsc{bserv}}\,(\textit{rec}) \quad (1)\ \Gamma_r \vdash^{L_r}_{a_r} \mathbf{ass}\,(b', b + 1)\,;\mathbf{inv}\,(l_{cb}, o_{cb}, \langle b, b'\rangle) : \textsc{bserv}}{\Gamma \quad \vdash^{L_r}_{a_r} \quad \mathbf{rec}\,(l, o', \langle b, l_{cb}\rangle)\,;\mathbf{ass}\,(b', b + 1)\,;\mathbf{inv}\,(l_{cb}, o_{cb}, \langle b, b'\rangle) : \textsc{bserv}}\,(\textit{seq})$$

Table 13: Type inference for the responder service s_r (Γ_r is $(\Gamma,\ l : \star,\ l_{cb} : \{o_{cb} : \langle \textsc{Int}, \textsc{Int}\rangle\})$)

adopted communication pattern is the asynchronous request-response, so the process, which has the provider role, holds the type definitions of both request operation (*shipReq*) and callback operation (*shipNot*).

To ensure that the net, consisting of only the service node, will never generate errors, it suffices to prove that it is well-typed wrt the empty environment. By the rule (*net*), we have that $\Gamma \vdash a ::^{Op,L} *s : \text{BNET}$, with $\Gamma = \emptyset$, $a : Op$. So, by the rule (*netToServ*) and (*serv*), this holds if the service specification s is well-typed wrt Γ and the local declarations. Formally, we must check that $\Gamma \vdash_a^L s : \text{BSERV}$ holds. This is what the inferences in Tables 14, 15 and 16, show. For the sake of presentation, the inferences are split in a few parts with references between them.

$$
\dfrac{\dfrac{\dfrac{\dfrac{itemsTot : \textsc{Int} \in L}{\emptyset \vdash^L_a itemsTot : \textsc{Int}}\,(var)}{\Gamma' \vdash^L_a itemsTot : \textsc{Int}}\,(weak)}{\Gamma' \vdash^L_a \mathbf{ass}\,(itemsCount, itemsTot)}\,(ass) \qquad \dfrac{\dfrac{itemsCount : \textsc{Int} \in L}{\emptyset \vdash^L_a itemsCount : \textsc{Int}}\,(var)}{\Gamma' \vdash^L_a itemsCount : \textsc{Int}}\,(weak) \qquad \dfrac{\dfrac{a : Op \vdash^L_a a : Op}{\Gamma' \vdash^L_a a : Op}\,(ref)}{shipNot : \bar{\tau}_n \in Op}\,(weak)}{(1)\ \Gamma' \vdash^L_a \mathbf{ass}\,(res_2, req_3) ; \mathbf{inv}\,(cust, shipNot, \overline{res}) : \textsc{bserv}}\,(seq)
$$

$$
\dfrac{\dfrac{\dfrac{p\_orderId : \textsc{Int} \in L}{\emptyset \vdash^L_a p\_orderId : \textsc{Int}}\,(var)}{\Gamma' \vdash^L_a p\_orderId : \textsc{Int}}\,(weak) \qquad \dfrac{\dfrac{itemsCount : \textsc{Int} \in L}{\emptyset \vdash^L_a itemsCount : \textsc{Int}}\,(var)}{\Gamma' \vdash^L_a itemsCount : \textsc{Int}}\,(weak)}{\Gamma' \vdash^L_a \overline{res} : \bar{\tau}_n}\,(\bar{vi}) \qquad \dfrac{\dfrac{cust : \star \vdash^L_a cust : \star}{cust : \star \vdash^L_a cust : \star}\,(ref)}{\Gamma' \vdash^L_a cust : \star}\,(weak)
$$

$$
\dfrac{\Gamma' \vdash^L_a \mathbf{inv}\,(cust, shipNot, \overline{res}) : \textsc{bserv}}{}\,(inv\_cb)
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p\_orderId : \textsc{Int} \in L}{\emptyset \vdash^L_a p\_orderId : \textsc{Int}}\,(prop)}{\Gamma \vdash^L_a p\_orderId : \textsc{Int}}\,(weak) \qquad \dfrac{\dfrac{complete : \textsc{Bool} \in L}{\emptyset \vdash^L_a complete : \textsc{Bool}}\,(var)}{\Gamma \vdash^L_a complete : \textsc{Bool}}\,(weak)}{\Gamma \vdash^L_a \overline{req} : \bar{\tau}_r}\,(\bar{vi})}{shipReq : \bar{\tau}_r \in Op}\,(rec)}{\Gamma \vdash^L_a \mathbf{rec}\,(cust, shipReq, \overline{req}) : \textsc{bserv}}\,(seq) \qquad \dfrac{\dfrac{\dfrac{itemsTot : \textsc{Int} \in L}{\emptyset \vdash^L_a itemsTot : \textsc{Int}}\,(var)}{\Gamma \vdash^L_a itemsTot : \textsc{Int}}\,(weak)}{\Gamma \vdash^L_a \mathbf{inv}\,(cust, shipNot, \overline{res}) : \textsc{bserv}}\,(\bar{vi})}{\Gamma \vdash^L_a \mathbf{rec}\,(cust, shipReq, \overline{req}) ; \mathbf{if}\ (req_2)\ \mathbf{then}\ \{s_{then}\}\ \mathbf{else}\ \{s_{else}\} : \textsc{bserv}}\,(seq)
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{a : Op \vdash^L_a a : Op}{\Gamma \vdash^L_a a : Op}\,(ref)}{shipReq : \bar{\tau}_r \in Op}\,(weak) \qquad \dfrac{\dfrac{complete : \textsc{Bool} \in L}{\emptyset \vdash^L_a complete : \textsc{Bool}}\,(var)}{\Gamma \vdash^L_a complete : \textsc{Bool}}\,(weak)}{(1)\ \Gamma' \vdash^L_a s_{then} : \textsc{bserv}} \qquad (2)\ \Gamma' \vdash^L_a s_{else} : \textsc{bserv}}{\Gamma' \vdash^L_a \mathbf{if}\ (complete)\ \mathbf{then}\ \{s_{then}\}\ \mathbf{else}\ \{s_{else}\} : \textsc{bserv}}\,(if)
$$

Table 14: Type checking of shipping service specification s (Γ' is (Γ, $cust : \star$))

Table 15: Γ'' is $(\Gamma', B : \text{BSERV}, l_c : \star)$ and L' is $L \cup \{items : \text{INT}, tot : \text{INT}, ordID : \text{INT}\}$

$$\dfrac{\dfrac{}{\Gamma'' \vdash_a^{L'} rand() : \text{INT}}\,(ran) \quad \dfrac{\dfrac{itemsC : \text{INT} \in L'}{\emptyset \vdash_a^{L'} itemsC : \text{INT}}\,(var)}{\Gamma'' \vdash_a^{L'} itemsC : \text{INT}}\,(weak)}{\Gamma'' \vdash_a^{L'} \mathbf{ass}\,(itemsC, rand()) : \text{BSERV}}\,(ass) \quad \dfrac{\dfrac{}{a : Op \vdash_a^{L'} a : Op}\,(ref)}{\Gamma'' \vdash_a^{L'} a : Op}\,(weak) \quad shipNot : \bar{\tau}_n \in Op \quad \dfrac{\cdots}{\Gamma'' \vdash_a^{L'} \langle ordId, itemC\rangle : \bar{\tau}_n}\,(\bar{w}) \quad \dfrac{\dfrac{}{l_c : \star \vdash_a^{L'} l_c : \star}\,(ref)}{\Gamma'' \vdash_a^{L'} l_c : \star}\,(weak)$$

$$\dfrac{\Gamma'' \vdash_a^{L'} \mathbf{ass}\,(itemsC, rand()) : \text{BSERV} \quad \Gamma'' \vdash_a^{L'} \mathbf{inv}\,(l_c, shipNot, \langle ordId, itemC\rangle) : \text{BSERV}}{(3)\ \Gamma'' \vdash_a^{L'} \mathbf{ass}\,(itemC, rand())\,;\ \mathbf{inv}\,(l_c, shipNot, \langle ordId, itemC\rangle)\,;\ \mathbf{ass}\,(itemsShipped', items + itemC)\,;\ \mathbf{if}\ (itemsShipped' < tot)\ \mathbf{then}\ \{B(\langle itemsShipped', tot, ordId, l_c\rangle)\}\ \mathbf{else}\ \{\mathbf{0}\} : \text{BSERV}}\,(inv\_cb) \quad (4)\ \Gamma'' \vdash_a^{L'} \mathbf{ass}\,(itemsShipped', items + itemC)\,;\ \mathbf{if}\ (itemsShipped' < tot)\ \mathbf{then}\ \{B(\langle itemsShipped', tot, ordId, l_c\rangle)\}\ \mathbf{else}\ \{\mathbf{0}\} : \text{BSERV}$$

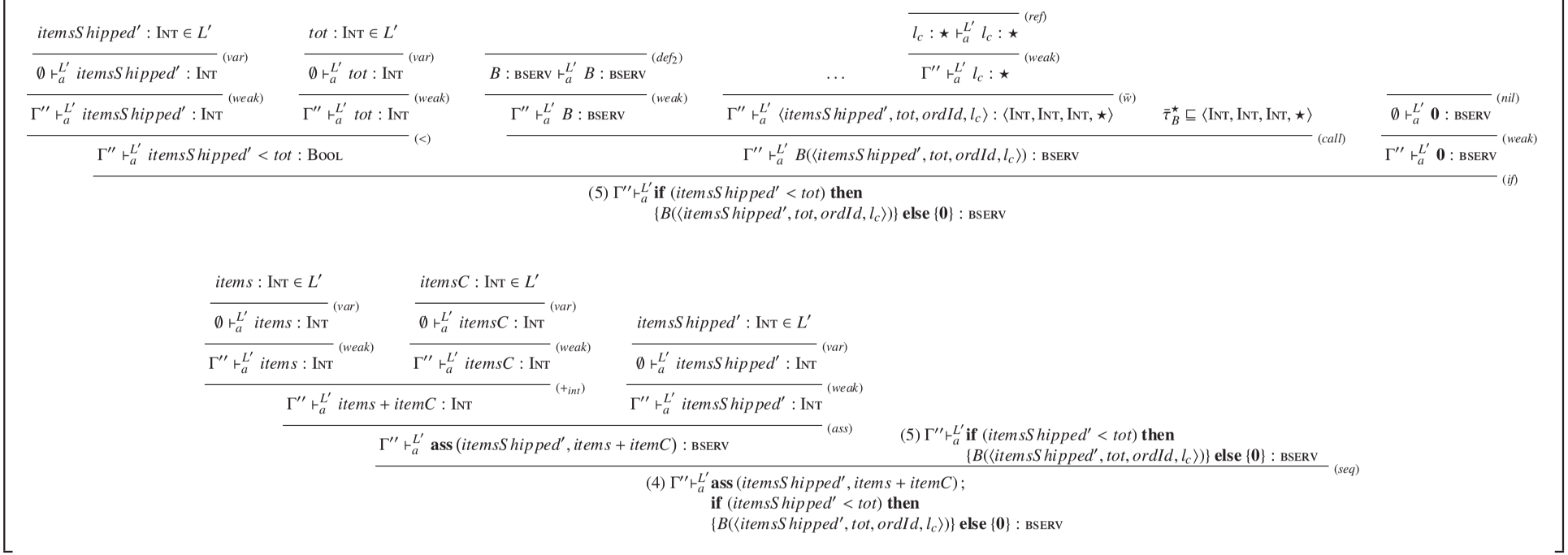
$$(3)\ \Gamma'' \vdash_a^{L'} \mathbf{ass}\,(itemC, rand())\,;\ \mathbf{inv}\,(l_c, shipNot, \langle ordId, itemC\rangle)\,;\ \mathbf{ass}\,(itemsShipped', items + itemC)\,;\ \mathbf{if}\ (itemsShipped' < tot)\ \mathbf{then}\ \{B(\langle itemsShipped', tot, ordId, l_c\rangle)\}\ \mathbf{else}\ \{\mathbf{0}\} : \text{BSERV}$$

$$\dfrac{\dfrac{\dfrac{0 \in \texttt{Int}}{\emptyset \vdash_a^{L} 0 : \text{INT}}\,(int)}{\Gamma' \vdash_a^{L} 0 : \text{INT}}\,(weak) \quad \dfrac{\dfrac{itemsShipped : \text{INT} \in L}{\emptyset \vdash_a^{L} itemsShipped : \text{INT}}\,(var)}{\Gamma' \vdash_a^{L} itemsShipped : \text{INT}}\,(weak)}{\Gamma' \vdash_a^{L} \mathbf{ass}\,(itemsShipped, 0) : \text{BSERV}}\,(ass) \quad \dfrac{(3)\ \Gamma'' \vdash_a^{L'} \mathbf{ass}\,(itemC, rand())\,;\ \mathbf{inv}\,(l_c, shipNot, \langle ordId, itemC\rangle)\,;\ \mathbf{ass}\,(itemsShipped', items + itemC)\,;\ \mathbf{if}\ (itemsShipped' < tot)\ \mathbf{then}\ \{B(\langle itemsShipped', tot, ordId, l_c\rangle)\}\ \mathbf{else}\ \{\mathbf{0}\} : \text{BSERV}}{\Gamma' \vdash_a^{L} B : \text{BSERV}}\,(def_1)$$

$$\dfrac{\dfrac{\cdots}{\Gamma' \vdash_a^{L} \langle itemsShipped, req_3, req_1, cust\rangle : \langle \text{INT}, \text{INT}, \text{INT}, \star\rangle}\,(\bar{w}) \quad \dfrac{\dfrac{}{cust : \star \vdash_a^{L} cust : \star}\,(ref)}{\Gamma' \vdash_a^{L} cust : \star}\,(weak) \quad \bar{\tau}_B \sqsubseteq \langle \text{INT}, \text{INT}, \text{INT}, \star\rangle}{\Gamma' \vdash_a^{L} B(\langle itemsShipped, req_3, req_1, cust\rangle) : \text{BSERV}}\,(call)$$

$$(2)\ \Gamma' \vdash_a^{L} \mathbf{ass}\,(itemsShipped, 0)\,;\ B(\langle itemsShipped, req_3, req_1, cust\rangle) : \text{BSERV}$$

Table 16: Type checking of a shipping service subterm



s	::=	$\ldots \mid ls \mid_L ls$	(flow graph)
ls	::=	$(jc) \overset{sjf}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})$	(join condition and outgoing links)
	$\mid$	$s \Rightarrow (\overline{fl}, \bar{e})$	(outgoing links)
jc	::=	**true** $\mid$ **false** $\mid$ fl	(join conditions)
	$\mid$	$\neg jc \mid jc \vee jc \mid jc \wedge jc$	
sjf	::=	$yes \mid no$	(suppress join failure)

Table 17: ws-calculus plus Flow links

5 Extensions of ws-calculus

In this section, we present some features not initially included in our calculus and, in particular, we extend ws-calculus in order to model flow graphs, timed activities, scopes and compensation handling. Extension of the typing system to deal with new constructs is in progress.

5.1 Flow graphs

A well-known characteristic of WS-BPEL is that its set of structured activities is not intended to be the minimal required set. There are cases, for example, where one activity can replace another. This is the case of the *flow activity*, used in [BCG+05] to structure workflow processing, that may be easily encoded within ws-calculus. Here, we present a possible encoding of the flow graph activity in ws-calculus.

In WS-BPEL, parallel execution of activities may be synchronized by establishing synchronization dependencies (here called *flow links* or *links* for short) among activities. At the beginning of the parallel execution, all links are inactive and only those activities with no synchronization dependencies can execute. When the execution of an activity completes, a *transition condition* is evaluated to determine the status of the outgoing links that can be *positive*, *negative* or *undefined*. Once all incoming links to an activity are active (that is, they have been assigned either a positive or negative state), a guard, called *join condition*, is evaluated. When an activity in the flow graph cannot execute (that is, the join condition fails), a *join failure* fault can be emitted to signal that some activities in the flow are not completed. An attribute called *suppress join failure* can be set to "*yes*" to ensure that join condition failures do not cause the flow activity to throw the *join failure* fault.

Default values for join and transition conditions are the logical `or` of the status of the incoming links and the boolean value **true** respectively, while the default setting of the *suppress join failure* attribute is "*no*".

In order to introduce the flow graph activity, we extend the syntax of ws-calculus as illustrated in Table 17. A flow graph activity $ls \mid_L ls$ is the parallel composition of two "linked" services ls, which can synchronize by means of the set of flow links denoted by L. In general, we assume that different flow link sets used by a service specification must be pairwise disjoint. A linked service ls is equipped with a set of incoming flow links that forms the join condition, and a set of outgoing flow links that represent the transition conditions. We denote incoming flow links and join condition with $(jc) \overset{sjf}{\Rightarrow}$. The outgoing links are represented by $\Rightarrow (\overline{fl}_{i\in I}, \bar{e}_{i\in I})$ where each pair (fl_i, e_i) denotes a transition condition, such that fl_i is a flow link and e_i is a boolean expression. The attribute sjf

$$
\begin{array}{rcl}
[\![ls \mid_L ls]\!] & = & [\![ls]\!] \mid [\![ls]\!] \\
[\![(jc) \stackrel{yes}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})]\!] & = & \textbf{if } (jc) \textbf{ then } \{[\![s]\!]; \textbf{ass}\left(\overline{fl}, \bar{e}\right)\} \textbf{ else } \{\textbf{ass}(outLinkOf(s), \overline{\textbf{false}})\} \\
[\![(jc) \stackrel{no}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})]\!] & = & \textbf{if } (jc) \textbf{ then } \{[\![s]\!]; \textbf{ass}\left(\overline{fl}, \bar{e}\right)\} \textbf{ else } \{\textbf{throw}\,(joinfault)\} \\
[\![s \Rightarrow (\overline{fl}, \bar{e})]\!] & = & [\![s]\!]; \textbf{ass}(\overline{fl}, \bar{e}) \\
[\![\textbf{if } (e) \textbf{ then } \{s_1\} \textbf{ else } \{s_2\}]\!] & = & \textbf{if } (e) \textbf{ then } \{\textbf{ass}(outLinkOf(s_2), \overline{\textbf{false}}); [\![s_1]\!]\} \\
 & & \quad \textbf{else } \{\textbf{ass}(outLinkOf(s_1), \overline{\textbf{false}}); [\![s_2]\!]\} \\
[\![\sum_{i\in I} \textbf{rec}\,(r_i, o_i, \bar{w}_i)\,; s_i]\!] & = & \sum_{i\in I} \textbf{rec}\,(r_i, o_i, \bar{w}_i)\,; \textbf{ass}(\bigcup_{j\in I,\, j\neq i} outLinkOf(s_j), \overline{\textbf{false}}); [\![s_i]\!]
\end{array}
$$

Table 18: Flow link encoding

permits to suppress a possible join failure, obtaining the so called *Dead-Path Elimination* (DPE) effect (see [BCG+05]).

The encoding[4] of flow graphs in ws-calculus is shown in Table 18. The auxiliary function $outLinkOf(s)$ returns the tuple (the order is not important) of the outgoing links used by s and is defined as follows:

$$
\begin{array}{l}
outLinkOf(s \Rightarrow (\overline{fl}, \bar{e})) = outLinkOf(s)\,,\ \overline{fl} \\
outLinkOf((jc) \stackrel{sjf}{\Rightarrow} s \Rightarrow (\overline{fl}, \bar{e})) = outLinkOf(s)\,,\ \overline{fl} \\
outLinkOf(\mathbf{0}) = outLinkOf(\mathbf{exit}) = \ldots = \emptyset \\
outLinkOf(A(\bar{x})) = outLinkOf(s) \quad \text{if } A(\bar{v}) \stackrel{def}{=} s \\
outLinkOf(s_1; s_2) = \ldots = outLinkOf(s_1 \mid s_2) = outLinkOf(s_1)\,,\ outLinkOf(s_2) \\
outLinkOf(ls_1 \mid_L ls_2) = outLinkOf(ls_1)\,,\ outLinkOf(ls_2)
\end{array}
$$

In the following, we comment some interesting encoding rules. Primarily, we say that a flow graph processing is considered as a usual parallel composition. A join condition is encoded as boolean condition within an **if-then-else** statement, in which the transition conditions are translated by means of assignments of forms $\textbf{ass}\left(\overline{fl}, e\right)$. In case of *suppress join failure* equals to *no*, a join condition failure produces a fault signal that can be caught by a proper fault handler (for this feature, we refer the interested readers to Section 5.3 where fault handlers are introduced). Structured activities **pick** and **switch** are translated so that, when an activity is selected, the outgoing links from the other activities of discarded branch are set to $false$.

5.2 Timed activities

A timed extension of ws-calculus is presented in order to model *waiting* activities of WS-BPEL that allow to wait for a given time period or until a certain time has passed.

The syntax of ws-calculus is now parameterized with respect to a new set `Time` of *time units* used to represent absolute time units (ranged over by t, t', ...) and time slots (ranged over by δ, δ',...). Thus, the syntax is extended so that each node of a net is equipped with a clock t,

[4] We have deliberately omitted the uninteresting cases, such as **rec**, **inv**, etc.

n	$::=$	$a ::_t^{Op,L} C$	(nodes)
C	$::=$	$\ldots \mid m, t \gg s$	(timed instances)
s	$::=$	$\ldots \mid \sum_{i \in I} g_i; s_i \mid \mathbf{wait}_{\mathrm{f}}(\delta) \mid \mathbf{wait}_{\mathrm{u}}(t)$	(wait activities)
g	$::=$	$\mathbf{rec}\,(r, o, \bar{w}) \mid \mathbf{wait}_{\mathrm{f}}(\delta) \mid \mathbf{wait}_{\mathrm{u}}(t)$	(guards)

Table 19: Timed WS-CALCULUS

representing an absolute time for each located service instance[5]. Two timed activities $\mathbf{wait}_{\mathrm{f}}(t)$ and $\mathbf{wait}_{\mathrm{u}}(t)$ permit to specify, respectively, the delay for a certain period of time or until a certain deadline is reached. Consequently, the **pick** activity is modified in order to wait occurrences for a message arrive or for a time-out event.

Extended Semantics. The extended semantics is given in terms of transitions between configurations $m, t \gg s \xrightarrow{\alpha} m, t' \gg s'$, where t and t' are time units and the label α is now generated by

$$\alpha ::= \ldots \mid \delta$$

This kind of labels permits to express events corresponding to time elapsing. Some relevant transition rules are presented in Table 20. Given a time slot δ and a clock t, the auxiliary function $timeOut(g, \delta, t)$ returns a boolean value stating that the time period has expired for the specified guard g:

$$timeOut(g, \delta, t) = \begin{cases} \mathbf{true} & \text{if } g = \mathbf{wait}_{\mathrm{f}}(\delta) \\ \mathbf{true} & \text{if } g = \mathbf{wait}_{\mathrm{u}}(t + \delta) \\ \mathbf{false} & otherwise \end{cases}$$

Evaluations of expressions and basic activities, except for $\mathbf{wait}_{\mathrm{f}}(t)$ and $\mathbf{wait}_{\mathrm{u}}(t)$, have an instantaneous execution, i.e. they don't consume time units. Consequently, rules for exit, assign, invoke, receive, if-then-else, sequence, flow and pick, are modified according to the new definition of configuration (i.e. adding the same clock before and after the transition). For example, the rule for the sequence activity is modified as follows:

$$\frac{m, t \gg s_1 \xrightarrow{\alpha} m, t \gg s_1' \quad \alpha \notin \mathtt{Time}}{m, t \gg s_1; s_2 \xrightarrow{\alpha} m, t \gg s_1'; s_2} \quad (Sequence)$$

Let us briefly comment some the rules of the operational semantics of timed ws-calculus processes. Rule (*tReceive*) permits time elapsing before the execution of the **receive** activity. Rules ($Wait_f 1$) and ($Wait_f 2$) permit to update the argument of $\mathbf{wait}_{\mathrm{f}}(\delta)$ until the timeout expires. Rules ($Wait_u 1$) and ($Wait_u 2$) permit time elapsing until the clock of the instance reaches the argument of $\mathbf{wait}_{\mathrm{u}}(t)$. Time elapsing in a sequence activity is governed by rule (*tSequence*). To synchronize time elapsing in a flow activity we use rule (*tFlow*). ($tPick_n$) rules determine that the time elapsing cannot make a choice within a pick activity, except when a timeout occurs.

The main modifications of the operational semantics for nets are represented by the following two rules:

$$\frac{\forall\, m, t \gg s \in C \quad m, t \gg s \xrightarrow{\delta} m, t + \delta \gg s'}{\{a ::_t C\} \rightarrowtail \{a ::_{t+\delta} C_\delta\}} \quad (tSync)$$

[5]Furthermore, for each new service instantiation, the clock t is passed to the new instance in order to properly synchronize its local clock.

$$m, t \gg \mathbf{rec}\,(r, o, \bar{w}) \xrightarrow{\delta} m, t + \delta \gg \mathbf{rec}\,(r, o, \bar{w}) \quad (tReceive)$$

$$m, t \gg \mathbf{wait}_{\mathrm{f}}(\delta) \xrightarrow{\delta} m, t + \delta \gg \mathbf{0} \quad (Wait_{\mathrm{f}}1)$$

$$\frac{\delta' < \delta}{m, t \gg \mathbf{wait}_{\mathrm{f}}(\delta) \xrightarrow{\delta'} m, t + \delta' \gg \mathbf{wait}_{\mathrm{f}}(\delta - \delta')} \quad (Wait_{\mathrm{f}}2)$$

$$\frac{t + \delta = t'}{m, t \gg \mathbf{wait}_{\mathrm{u}}(t') \xrightarrow{\delta} m, t \gg \mathbf{0}} \quad (Wait_{\mathrm{u}}1)$$

$$\frac{t + \delta < t'}{m, t \gg \mathbf{wait}_{\mathrm{u}}(t') \xrightarrow{\delta} m, t + \delta \gg \mathbf{wait}_{\mathrm{u}}(t')} \quad (Wait_{\mathrm{u}}2)$$

$$\frac{m, t \gg s_1 \xrightarrow{\delta} m, t + \delta \gg s_1'}{m, t \gg s_1; s_2 \xrightarrow{\delta} m, t + \delta \gg s_1'; s_2} \quad (tSequence)$$

$$\frac{m, t \gg s_1 \xrightarrow{\delta} m, t + \delta \gg s_1' \quad m, t \gg s_2 \xrightarrow{\delta} m, t + \delta \gg s_2'}{m, t \gg s_1 \mid s_2 \xrightarrow{\delta} m, t + \delta \gg s_1' \mid s_2'} \quad (tFlow)$$

$$\frac{\forall\, i \in I \;\; m, t \gg g_i \xrightarrow{\delta} m, t + \delta \gg g_i' \quad timeOut(g_i, \delta, t) = \mathbf{false}}{m, t \gg \sum_{i \in I} g_i; s_i \xrightarrow{\delta} m, t + \delta \gg \sum_{i \in I} g_i'; s_i} \quad (tPick1)$$

$$\frac{m, t \gg \mathbf{wait}_{\mathrm{f}}(\delta) \xrightarrow{\delta} m, t + \delta \gg \mathbf{0}}{m, t \gg \mathbf{wait}_{\mathrm{f}}(\delta); s + \sum_{i \in I} g_i; s_i \xrightarrow{\delta} m, t + \delta \gg s} \quad (tPick2)$$

$$\frac{m, t \gg \mathbf{wait}_{\mathrm{u}}(t + \delta) \xrightarrow{\delta} m, t + \delta \gg \mathbf{0}}{m, t \gg \mathbf{wait}_{\mathrm{u}}(t + \delta); s + \sum_{i \in I} g_i; s_i \xrightarrow{\delta} m, t + \delta \gg s} \quad (tPick3)$$

Table 20: Timed ws-calculus operational semantics: instances

$$\frac{\emptyset, t \gg s \xrightarrow{r(a', o, \bar{w})} \emptyset, t \gg s' \quad \emptyset \rhd (\bar{w} := \bar{u}) = \langle m, \sigma \rangle \quad \mathbf{rec}\,(a', o, \bar{w}) \notin eR(C)}{\{a ::_t * s \mid \langle a', o, \bar{u} \rangle \mid C\} \rightarrowtail \{a ::_t * s \mid m, t \gg s' \cdot \sigma \mid C\}} \quad (Receive_{aS})$$

Rule (*tSync*) says that all the instances of a service are time-synchronized (i.e. time elapsing transitions are synchronous). The notation C_δ indicates the set of components obtained from C by

replacing each instance $m, t \gg s$ with $m, t + \delta \gg s'$. The last rule ($Receive_{aS}$) permits to initiate the local clock of a new service instance.

5.3 Scopes and compensation handling

Error handling in ws-calculus relies on the concept of *compensation*, that is, specific activities that attempt to reverse the effects of a previous group of activities. WS-BPEL, such as ws-calculus, provides a variant of *Sagas* compensation protocol [GMS87] in which the behavior of a compensation handler can be thought of as an optional continuation of the behavior of the associated group of activities. To reason about compensation, we extend the syntax by including scoping constructs $[s : h]_\kappa$ as a means to explicitly grouping activities together (see Table 21). When we declare a scoping construct, we have to include, optionally, the list of fault handlers by concluding the list with the standard fault handler followed by the compensation handler (standard fault and compensation handlers must be, in order, the last). Thus, we extend the syntax to signal an internal fault within a service by using **throw**(ϕ), where ϕ identifies a fault signal. The fault handling behaviour is standard: in case of a fault signal ϕ, the corresponding activity **catch**(ϕ){s}, if exists within the scope, is selected, otherwise the default **catchAll**{s} activity is chosen. When it is necessary to rethrow a caught fault to the next enclosing scope, we'll use the special fault name *this*. The compensate activity **undo**(κ) can be used to invoke a compensation handler associated to an inner scope κ that has already completed normally without faulting. This construct can be invoked only from within a fault handler or another compensation handler. When the scope name is omitted, **undo** causes all inner scopes to execute their compensation handlers in reverse order. Such as WS-BPEL, we consider illegal mixed usage of both activities **undo** and **undo**(κ) within a single scope (since the latter disables the availability of former compensation activity). Moreover, we consider incorrect handlers containing scope constructs. We assume services having the following properties of *undo-correctness*: for each handler h_i of a scope containing **undo**(κ) (resp. **undo**) there exists at least an inner scope named κ.

We use $\mathbb{FID}$, $\mathbb{SID}$ to denote the sets of fault and scope identifiers, respectively. Frequently we use $\mathbb{SID}_{\mathbb{N}}$ to denote the set of indexed scope identifiers.

The operational semantics of compensation exploits mappings, called *compensation function*, from *indexed* scope identifiers κ_n to the installed compensation handlers of successfully completed scopes. We use indexed identifiers since a scope may have been executed several times within a loop, so that κ_n may denote the state of each successfully completed iteration. Specific symbols *nsc* and *undone* represent, respectively, a scope *not successfully completed* and a scope in which the compensation handler has been already invoked. We always assume that initially for each configuration we have $\delta(\kappa_n) = nsc$, for all $\kappa_n \in \mathbb{SID}_{\mathbb{N}}$.

Fault handling or compensation behavior are treated as protected services by the semantics of the forced termination activity **exit**. Thus, we extend the syntax by adding a *protection* activity $\uparrow s \uparrow$ of StAC$_i$ [BF04]. Forced termination semantics exploits an auxiliary function **halt**(s) representing the forced termination of s except for (recursively nested) protected activities. Table 22 shows only the case in which the function returns services different from **0**.

Extended Semantics. The extended semantics for compensations is given in terms of transitions between configurations $m \vdash (s, \delta) \xrightarrow{\alpha} (s', \delta')$, where δ stands for the above-mentioned mappings (i.e. compensation function) from scope identifiers to compensation handlers. The reduction relation is presented in Table 24, where the label α is now generated by:

$$\alpha \ ::= \ \ldots \ \mid \ \phi \ \mid \ \kappa_n \ \mid \ repeatedCmp$$

s	::=	$\ldots$	
	$\mid$	$[s : h]_\kappa$	(scopes)
	$\mid$	$\mathbf{throw}(\phi)$	(throwing)
	$\mid$	$\mathbf{undo}(\kappa) \mid \mathbf{undo}$	(compensating)
	$\mid$	$\uparrow s \uparrow$	(protecting)
h	::=		(handlers)
		$\mathbf{catch}(\phi)\{s\} : h$	(specific faults)
	$\mid$	$\mathbf{catchAll}\{s\} : \mathbf{comp}\{s\}$	(default handlers)
δ	::=	$(\kappa_n \mapsto s) \mid (\kappa_n \mapsto nsc) \mid (\kappa_n \mapsto undone) \mid \delta \circ \delta$	(comp. function)

Table 21: ws-calculus plus Compensation

$$\mathbf{halt}(s_1; s_2) = \mathbf{halt}(s_1) \quad \mathbf{halt}(s_1 \mid s_2) = \mathbf{halt}(s_1) \mid \mathbf{halt}(s_2) \quad \mathbf{halt}(\uparrow s \uparrow) = \uparrow s \uparrow$$

$$\mathbf{halt}([s : h]_\kappa) = \uparrow \mathbf{halt}(s); s' \uparrow \quad \textit{with} \ \mathbf{catch}(\natural)\{s'\} \in h \ \textit{or} \ \mathbf{catchAll}\{s'\} \in h$$

Table 22: Forced Termination

A service that signals an internal fault is described by Rule (*Throw*). A protection activity performs the same action performed by the protected service such as in (*Protect*). ($ScopeExc_1$) permits to perform all the action of the inner service except for fault signals and *scope completion* signals. Scope completion signals are consumed by Rule ($ScopeExc_2$).

A *compensation stack push* function, denoted by $csp(h, \kappa_n)$ with $\kappa_n \in \mathbb{SID}_{\mathbb{N}}$, replaces each **undo** (resp. $\mathbf{undo}(\kappa)$) occurring in h with $\uparrow \mathbf{undo}; \mathbf{undo}(\kappa_n) \uparrow$ (resp. $\uparrow \mathbf{undo}(\kappa); \mathbf{undo}(\kappa_n) \uparrow$). The most relevant definitions of this function are in Table 23. For the rules concerning fault handling ($Fault_1$) and ($Fault_2$), we consider the list h as follows:

$$h = \mathbf{catch}(\phi_1)\{s_1\} : \mathbf{catch}(\phi_2)\{s_2\} : \ldots \mathbf{catch}(\phi_n)\{s_n\} : \mathbf{catchAll}\{s_{all}\} : \mathbf{comp}\{s_c\}$$

When a fault signal occurs within a scope, we force the termination of the inner service by protecting the execution of the resulting service composed with the fault handling. Rule (*ScopeComp*) says that a scope completion signal is produced when a scope successfully complete installing the compensation handler into the compensation function. Rule (*Undo*) says that **undo** and $\mathbf{undo}(\kappa)$ produce a silent action since their role is only to be the prefix of sequence of $\mathbf{undo}(\kappa_n)$ activities. Moreover, this rule says that invoking a compensation handler that has not been installed (the scope is not still successfully completed) is equivalent to the empty activity. Rule (*Cmp*) permits to perform a specified compensation handler. Rule (*RptCmp*) describes the semantics of a service invoking a compensation handler more than once saying that the fault signal *repeatCmp* is thrown.

$csp(\mathbf{0}, \kappa_n) = \mathbf{0}$	$csp(s_1; s_2, \kappa_n) = csp(s_1, \kappa_n); csp(s_2, \kappa_n)$
$csp(s_1 \mid s_2, \kappa_n) = csp(s_1, \kappa_n) \mid csp(s_2, \kappa_n)$	$csp([s : h]_{\kappa'}, \kappa_n) = [csp(s, \kappa_n) : h]_{\kappa'}$
$csp(\mathbf{undo}, \kappa_n) = \uparrow \mathbf{undo}; \mathbf{undo}(\kappa_n) \uparrow$	$csp(\mathbf{undo}(\kappa), \kappa_n) = \uparrow \mathbf{undo}(\kappa); \mathbf{undo}(\kappa_n) \uparrow$
$csp(\mathbf{undo}(\kappa'), \kappa_n) = \mathbf{undo}(\kappa') \; \forall \kappa' \in \mathbb{SID} : \kappa' \neq \kappa$	$csp(\uparrow s \uparrow, \kappa_n) = \uparrow csp(s, \kappa_n) \uparrow$
$csp(\mathbf{undo}(\kappa'_m), \kappa_n) = \mathbf{undo}(\kappa'_m) \; \forall \kappa'_m \in \mathbb{SID}_{\mathbb{N}}$	$csp(\mathbf{throw}(\phi), \kappa_n) = \mathbf{throw}(\phi)$
$csp(\mathbf{catch}(\phi)\{s\}, \kappa_n) = \mathbf{catch}(\phi)\{csp(s, \kappa_n)\}$	$csp(\mathbf{catchAll}\{s\}, \kappa_n) = \mathbf{catchAll}\{csp(s, \kappa_n)\}$

Table 23: Compensation stack push function

6 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 two illustrative examples. We have extended ws-calculus to also deal with such constructs as flow graphs, timed activities, scopes and compensation handling, thus giving a complete semantic account of WS-BPEL executable processes.

We are currently extending the typing system, and the related results, to the enriched language of Section 5. 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 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

$$\frac{\phi \in \mathbb{FID}}{m \vdash (\mathbf{throw}(\phi), \delta) \xrightarrow{\phi} (\mathbf{0}, \delta)} \quad (Throw)$$

$$\frac{m \vdash (s, \delta) \xrightarrow{\alpha} (s', \delta')}{m \vdash (\uparrow s \uparrow, \delta) \xrightarrow{\alpha} (\uparrow s' \uparrow, \delta')} \quad (Protect)$$

$$\frac{m \vdash (s, \delta) \xrightarrow{\alpha} (s', \delta') \quad \alpha \notin (\mathbb{FID} \cup \mathbb{SID}_{\mathbb{N}})}{m \vdash ([s : h]_\kappa, \delta) \xrightarrow{\alpha} ([s' : h]_\kappa, \delta')} \quad (ScopeExc_1)$$

$$\frac{m \vdash (s, \delta) \xrightarrow{\kappa'_n} (s', \delta') \quad \kappa'_n \in \mathbb{SID}_{\mathbb{N}}}{m \vdash ([s : h]_\kappa, \delta) \xrightarrow{\tau} ([s' : csp(h, \kappa'_n)]_\kappa, \delta')} \quad (ScopeExc_2)$$

$$\frac{m \vdash (s, \delta) \xrightarrow{\phi_i} (s', \delta') \quad \phi_i \in \mathbb{FID}}{m \vdash ([s : h]_\kappa, \delta) \xrightarrow{\tau} (\uparrow \mathbf{halt}(s'); s_i \uparrow, \delta')} \quad (Fault_1)$$

$$\frac{m \vdash (s, \delta) \xrightarrow{\phi} (s', \delta') \quad \forall \phi_i \in h : \phi \neq \phi_i}{m \vdash ([s : h]_\kappa, \delta) \xrightarrow{\tau} (\uparrow \mathbf{halt}(s'); s_{all} \uparrow \cdot (this \mapsto \phi), \delta')} \quad (Fault_2)$$

$$\frac{\delta(\kappa_n) = nsc \quad \nexists m \in \mathbb{NAT} : m < n \wedge \delta(\kappa_m) = nsc}{m \vdash ([\mathbf{0} : h]_\kappa, \delta) \xrightarrow{\kappa_n} (\mathbf{0}, \delta \circ (\kappa_n \mapsto s_c))} \quad (ScopeComp)$$

$$\frac{U = \mathbf{undo} \vee U = \mathbf{undo}(\kappa) \quad with\ \kappa \in \mathbb{SID}}{m \vdash (U, \delta) \xrightarrow{\tau} (\mathbf{0}, \delta)} \quad (Undo)$$

$$\frac{\kappa_n \in \mathbb{SID}_{\mathbb{N}} \quad \delta(\kappa_n) = s_c}{m \vdash (\mathbf{undo}(\kappa_n), \delta) \xrightarrow{\tau} (s_c, \delta \circ (\kappa_n \mapsto undone))} \quad (Cmp)$$

$$\frac{\kappa_n \in \mathbb{SID}_{\mathbb{N}} \quad \delta(\kappa_n) = undone}{m \vdash (\mathbf{undo}(\kappa_n), \delta) \xrightarrow{repeatedCmp} (\mathbf{0}, \delta)} \quad (RptCmp)$$

Table 24: Operational semantics: compensation

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.

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/`.

[BF04] Michael J. Butler and Carla Ferreira. An operational semantics for stac, a language for modelling long-running business transactions. In *COORDINATION*, pages 87–104, 2004.

[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.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD Conference*, pages 249–259, 1987.

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

[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.

Op	$::=$	$\emptyset \mid \{o : \bar{\tau}\} \mid Op \cup Op$	(operation type sets)
bt	$::=$	$\textsc{Bool} \mid bt_1 \mid \ldots \mid bt_n$	(basic types)
τ	$::=$	$bt \mid Op$	(message field types)
t	$::=$	$\bar{\tau} \mid \textsc{bnet} \mid \textsc{bserv}$	(generic types)

Table 25: Modified type syntax

Appendix. A revisited type system

In this section we present a revisited type system for WS-CALCULUS, that is more standard and designed to infer types of variables and properties without using local declarations, differently from one described in Section 3.

This approach requires some smooth modifications to the syntax of WS-CALCULUS. Now, the syntax of our calculus is parameterized with respect to the following syntactic sets, which we assume to be countable and pairwise disjoint: *properties* (ranged over by p), *values* (ranged over by u, this is the set of communicable objects, i.e. basic values and addresses) and corresponding *variables* (ranged over by v), 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, values, variables and properties.

Notationally, we prefer letters $a, a', \ldots$ when we want to stress the use of a value as an address, $l, l', \ldots$ when we want to stress the use of a variable as a partner link. We will use $x, x', y, y', \ldots$ for properties, variables and values, and the operator $\ulcorner \cdot \urcorner$ that forces partner links to be already initialized.

The syntax of types is defined in Table 25. The main change is the removal of local declarations L from nodes, now written as $a ::^{Op} C$. Moreover, basic types are parametrically defined (we explicitly define only the type of booleans, because we need it to type switch activities).

Tables 26 and 28 show inferences rules for net and services, respectively. Instead, Table 27 show inference rules for values, variables and expressions. In this latter case, since we are parametric wrt expressions, we assume to be able to infer the type of any well-typed expression.

We notice that, differently from Section 3, we write $\Gamma \vdash_{Op} S : t$ to indicate that S has type t wrt the type environment Γ and the type Op, where S is a metavariable denoting services, variables, properties and values, and Op denotes the set of operation types defined in the considered node. [6]

To define the type system, we exploit a few auxiliary functions:

- $typeOf(x)$ returns the type of x if it is a basic value, the function is undefined otherwise;
- $isVar(x)$ returns **true** if x is a variable, **false** otherwise;
- $envExt_{\Gamma,Op}(s)$ returns the set of type associations created in s from the environment Γ and Op as the operation type set of considered node. The function is defined inductively on the syntax of services as follows:
  - $envExt_{\Gamma,Op}(\mathbf{rec}\,(x, o, \bar{y}_{i\in I})) =$

$$\begin{cases} \{y_i : \tau_i \mid i \in I\} \cup \{x : \star \mid isVar(x) = \mathbf{true} \wedge x \neq \ulcorner l \urcorner\} & \text{if } o : \bar{\tau}_{i\in I} \in Op \\ \{y_i : \tau_i \mid i \in I \wedge \Gamma \vdash_{Op} x : Op' \wedge o : \bar{\tau}_{i\in I} \in Op'\} & \text{otherwise} \end{cases}$$

[6] We use Op, rather than the local address a, only to reduce the size of the rules. Indeed, we can always obtain Op from a by $\Gamma \vdash_a a : Op$.

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

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

Table 26: Inference rules for $\Gamma \vdash N : \textsc{bnet}$

$$\frac{typeOf(u) = bt}{\emptyset \vdash_{Op} u : bt} \quad (bval) \qquad a : Op' \vdash_{Op} a : Op' \quad (addr)$$

$$l : Op' \vdash_{Op} \ulcorner l \urcorner : Op' \quad (arec) \qquad p : bt \vdash_{Op} p : bt \quad (prop)$$

$$v : \tau \vdash_{Op} v : \tau \quad (var) \qquad \frac{\Gamma' \vdash_{Op} S : t \quad \Gamma \leqslant \Gamma'}{\Gamma \vdash_{Op} S : t} \quad (weak)$$

$$\frac{\Gamma \vdash_{Op} x_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_{Op} x_n : \tau_n}{\Gamma \vdash_{Op} \langle x_1, \ldots, x_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \quad (\bar{x}) \qquad \frac{\Gamma \vdash_{Op} e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_{Op} e_n : \tau_n}{\Gamma \vdash_{Op} \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \quad (\bar{e})$$

Table 27: Inference rules for $\Gamma \vdash_{Op} S : t$

- $envExt_{\Gamma,Op}(\mathbf{ass}\,(\bar{x}_{i\in I}, \bar{e})) = \{x_i : \tau_i \mid i \in I \wedge \Gamma \vdash_{Op} \bar{e} : \bar{\tau}_{i\in I}\}$
- $envExt_{\Gamma,Op}(\mathbf{inv}\,(x, o, \bar{y})) = \emptyset$
- $envExt_{\Gamma,Op}(\mathbf{exit}) = \emptyset$
- $envExt_{\Gamma,Op}(A(\bar{x})) = A : \textsc{bserv}$
- $envExt_{\Gamma,Op}(s_1; s_2) = envExt_{(\Gamma,\, envExt_{\Gamma,Op}(s_1)),Op}(s_2)$
- $envExt_{\Gamma,Op}(s_1 \mid s_2) = envExt_{\Gamma,Op}(s_1) \cup envExt_{\Gamma,Op}(s_2)$
- Given $G_1 = envExt_{\Gamma,Op}(s_1)$ and $G_2 = envExt_{\Gamma,Op}(s_2)$, we have:
$$envExt_{\Gamma,Op}(\mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\}) = \begin{cases} G & \text{if } G_1 = G_2 = G \\ \mathbf{undef} & \text{otherwise} \end{cases}$$
- Given $G_i = envExt_{\Gamma,Op}(\mathbf{rec}\,(x_i, o_i, \bar{y}_i)\,; s_i)$ for $i \in I$, we have:
$$envExt_{\Gamma,Op}(\textstyle\sum_{i\in I} \mathbf{rec}\,(x_i, o_i, \bar{y}_i)\,; s_i) = \begin{cases} G & \text{if } \forall\, i \in I \text{ it holds that } G_i = G \\ \mathbf{undef} & \text{otherwise} \end{cases}$$

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

$$\frac{\Gamma \vdash_{Op} s : \text{BSERV}}{\Gamma \vdash_{Op} \_s : \text{BSERV}} \ (serv) \qquad \frac{\Gamma \vdash_{Op} C_1 : \text{BSERV} \qquad \Gamma \vdash_{Op} C_2 : \text{BSERV}}{\Gamma \vdash_{Op} C_1 \mid C_2 : \text{BSERV}} \ (par)$$

$$\emptyset \vdash_{Op} \mathbf{0} : \text{BSERV} \quad (nil) \qquad \emptyset \vdash_{Op} \mathbf{exit} : \text{BSERV} \quad (exit)$$

$$\Gamma \vdash_{Op} \mathbf{ass}\,(\bar{x}, \bar{e}) : \text{BSERV} \quad (ass)$$

$$\frac{\Gamma \vdash_{Op} x : Op' \quad o : \bar{\tau} \in Op' \quad \Gamma \vdash_{Op} \bar{y} : \bar{\tau}' \quad \bar{\tau} \sqsubseteq \bar{\tau}' \quad Op \cap Op' = \emptyset}{\Gamma \vdash_{Op} \mathbf{inv}\,(x, o, \bar{y}) : \text{BSERV}} \ (inv)$$

$$\frac{o : \bar{\tau} \in Op \quad \Gamma \vdash_{Op} \bar{y} : \bar{\tau}' \quad \bar{\tau} \sqsubseteq \bar{\tau}' \quad \Gamma \vdash_{Op} x : \star \ \vee \ (\Gamma \vdash_{Op} x : Op' \ \wedge \ isVar(x) = \mathbf{false})}{\Gamma \vdash_{Op} \mathbf{inv}\,(x, o, \bar{y}) : \text{BSERV}} \ (inv\_cb)$$

$$\frac{\Gamma \vdash_{Op} x : Op' \quad o : \bar{\tau} \in Op' \quad Op \cap Op' = \emptyset}{\Gamma \vdash_{Op} \mathbf{rec}\,(x, o, \bar{y}) : \text{BSERV}} \ (rec\_cb)$$

$$\frac{o : \bar{\tau} \in Op \quad x = \ulcorner l \urcorner \Rightarrow \Gamma \vdash_{Op} l : \star}{\Gamma \vdash_{Op} \mathbf{rec}\,(x, o, \bar{y}) : \text{BSERV}} \ (rec)$$

$$\frac{\Gamma \vdash_{Op} e : \text{BOOL} \quad \Gamma \vdash_{Op} s_1 : \text{BSERV} \quad \Gamma \vdash_{Op} s_2 : \text{BSERV}}{\Gamma \vdash_{Op} \mathbf{if}\ (e)\ \mathbf{then}\ \{s_1\}\ \mathbf{else}\ \{s_2\} : \text{BSERV}} \ (if)$$

$$\frac{\Gamma \vdash_{Op} s_1 : \text{BSERV} \quad \Gamma, envExt_{\Gamma,Op}(s_1) \vdash_{Op} s_2 : \text{BSERV}}{\Gamma \vdash_{Op} s_1; s_2 : \text{BSERV}} \ (seq)$$

$$\frac{\Gamma \vdash_{Op} s_1 : \text{BSERV} \quad \Gamma \vdash_{Op} s_2 : \text{BSERV}}{\Gamma \vdash_{Op} s_1 \mid s_2 : \text{BSERV}} \ (flow)$$

$$\frac{\forall\, i \in I \quad \Gamma \vdash_{Op} \mathbf{rec}\,(x_i, o_i, \bar{y}_i)\,; s_i : \text{BSERV}}{\Gamma \vdash_{Op} \sum_{i \in I} \mathbf{rec}\,(x_i, o_i, \bar{y}_i)\,; s_i : \text{BSERV}} \ (pick)$$

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

$$\frac{\Gamma, A : \text{BSERV}, \bar{v}_{i \in I} : \bar{\tau}^\star_{i \in I} \vdash_{Op} s : \text{BSERV}}{\Gamma \vdash_{Op} A : \text{BSERV}} \ A(\bar{v}_{i \in I} : \bar{\tau}^\star_{i \in I}) \stackrel{def}{=} s \quad (def_1)$$

$$A : \text{BSERV} \vdash_{Op} A : \text{BSERV} \quad (def_2)$$

Table 28: Inference rules for $\Gamma \vdash_{Op} s : \text{BSERV}$