Form Methods Syst Des (2011) 38: 119–157
DOI 10.1007/s10703-010-0110-0

A WSDL-based type system for asynchronous WS-BPEL processes

A. Lapadula · R. Pugliese · F. Tiezzi

Published online: 24 December 2010

Abstract We tackle the problem of providing rigorous formal foundations to current software engineering technologies for web services, and especially to WSDL and WS-BPEL, two of the most used XML-based standard languages for web services. We focus on a simplified fragment of WS-BPEL sufficiently expressive to model asynchronous interactions among web services in a network context. We present this language as a process calculus-like formalism, that we call WS-CALCULUS, for which we define an operational semantics and a type system. The semantics provides a precise operational model of programs, while the type system forces a clean programming discipline for integrating collaborating services. We prove that the operational semantics of WS-CALCULUS and the type system are 'sound' and apply our approach to some illustrative examples. We expect that our formal development can be used to make the relationship between WS-BPEL programs and the associated WSDL documents precise and to support verification of their conformance.

Keywords Web services · WSDL · WS-BPEL · Process calculi · Type systems

1 Introduction

Service-Oriented Computing (SOC) has been 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 technolo-

The work presented in this paper is a much extended and revised version of Lapadula et al. [52], and has been partially supported by EU Project "Software Engineering for Service-Oriented Overlay Computers" (SENSORIA), IST-2005-016004.

A. Lapadula · R. Pugliese · F. Tiezzi (✉)
Viale Morgagni, 65, 50134 Firenze, Italy
e-mail: tiezzi@dsi.unifi.it

A. Lapadula
e-mail: lapadula@dsi.unifi.it

R. Pugliese
e-mail: rosario.pugliese@unifi.it

gies. 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 (WSs) are presently the most successful instantiation of the SOC paradigm, as it is demonstrated by the fact that companies like IBM, Microsoft, Oracle and Sun invested a lot of effort and resources to promote their use.

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 business coordination languages (such as WS-BPEL [61], WSFL [56], WSCI [5], and XLANG [66]), contract languages (such as WSDL [25] and SWS [7]), and query languages (such as XPath [26] and XQuery [9]). In spite of the many research efforts that have been done in the last few years, current software engineering technologies for WSs still lack rigorous formal foundations and are usually tightly coupled to RPC-based mechanisms, which previous technologies such as CORBA [62] were based on. Although these mechanisms provide a satisfactory solution to hide complexity from developers, they can be problematic for achieving the full potential of WS business processes. The challenges come from the necessity of dealing at once with issues like asynchronous interactions, concurrency, workflow coordination, business transactions, resource usage, failures, security, etc. in a setting where demands and guarantees can be very different for the different components.

We consider two of the most used XML-based languages for WSs: *Web Services Description Language* (WSDL) and *Web Services Business Process Execution Language* (WS-BPEL). 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 data for invoking them and returned by them. The latter is an OASIS standard that permits to describe the activities of the business logic and the interactions to be executed for completing a service as a reaction to a service invocation. A service, in fact, often results from the *orchestration* of other available services, i.e. from their aggregation and invocation according to a given set of rules to meet a business requirement.

Hence, 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 orchestrating other existing ones. But only two of the four different types of operations provided by WSDL are really supported by WS-BPEL: (asynchronous) one-way and (synchronous) request-response. Moreover, it is often implicitly assumed that a service request can be processed in a reasonable amount of time, which in practice means that the invoker is justified to wait for a response related to a synchronous request-response operation. However, in a business process setting, where transactions are usually long-running, such an assumption is not realistic and the interactions should be better modeled as a pair of asynchronous message exchanges. For example, consider a travel agency service that accepts a request and returns a confirmation after some checks, e.g. the customer credit card must have sufficient available funds. This is an interaction where the check may take a significant amount of time. For this reason, an asynchronous messaging approach is considered good practice for web services, much more in the case of service orchestrations. This is not to say that synchronous service behavior is wrong, but experience and practice (see the initiative of the SOA Patterns community [29]) have demonstrated that asynchronous service behavior is desirable, especially when communication costs are high or network latency is unpredictable, and provides the developer with a simpler scalability model.

In this paper we aim at making the relationship between WS-BPEL programs and the associated WSDL documents precise, with special attention to asynchronous interactions.

As a first contribution, we introduce a semantic model for defining WS-BPEL processes interacting asynchronously. Indeed, although WS-BPEL is popular and provides the means for efficient business solutions, its complexity is largely responsible for its limited applicability. Actually, WS-BPEL provides many redundant (and, sometimes, intricate) programming constructs and suggests 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. Moreover, the language comes without a precise semantics and its specification document [61], written in 'natural' language, contains a fair number of acknowledged loose points that may give rise to different interpretations. Some examples are illustrated in [55], where it is demonstrated that different WS-BPEL implementations may have different semantics. Hence, we introduce WS-CALCULUS (*web services calculus*), a process calculus-like formalism that formalizes the semantics of a simplified fragment of WS-BPEL, with special concern for modelling the asynchronous interactions among WS-BPEL processes in a network context. WS-CALCULUS is expressive enough to model many specific aspects of execution of WS-BPEL processes, such as multiple start activities, receive conflicts, delivering of messages, while avoiding the intricacies of dealing with each, possibly redundant, WS-BPEL construct. WS-CALCULUS syntax also allows us to get much more compact and clearer process specifications than the verbose XML-based syntax of WS-BPEL, as it is also witnessed by the example shown in Sect. 5.1.

As a second contribution, we define a type system for WS-CALCULUS terms that, among other properties, forces a clean programming discipline for collaborating processes that communicate asynchronously. Our types only rely on the type information that can be extracted from the WSDL and WS-BPEL documents that specify processes' behaviour and interface. In particular, WS-BPEL enables inter-service message exchanges by exploiting the notion of *partner link*, whose types are defined at the WSDL level. Partner link types are used to directly model peer-to-peer partner collaborations where the actual partner service may be dynamically determined. Thus, a partner link provides a process with the ability to invoke a service and, possibly, allows such a service to respond.

We deem as *error configurations* those configurations where variables are going to get assigned a value of wrong type or the interaction pattern prescribed by a partner link is not respected, either because the operation is not provided or because its type is not compatible with the type of the message invocation. We prove that well-typed terms are 'safe', i.e. error-free, and that the operational semantics and the type system are 'sound', in the sense that terms reached along any reduction sequence starting from well-typed terms are still well-typed. Together, these two properties imply that, other than to prevent assigning values of wrong type to variables, the type system for WS-CALCULUS implements a sort of 'partner link control' that checks the ability of a process to invoke the partner specified by a partner link and the ability of that partner to asynchronously respond. The type system also accommodates the underlying address-passing communication mechanism, which is necessary since process interaction can require the transmission to the partner of the process reference bound to a partner link.

In spite of its simplicity with respect to more powerful type systems for other process calculus-like formalisms (see e.g. [3, 19, 20, 23, 40–42, 45, 46, 49, 70]), and possibly exactly because of its simplicity, we believe that our type system could be used in practice to support verification of conformance between WS-BPEL programs and the associated WSDL documents. Indeed, it enforces the basic constraints imposed by WSDL against WS-BPEL programs, as e.g. that the message types of operation invocations match the operation types declared in the WSDL document (as prescribed in [61, Appendix A. Standard Faults]). In addition, differently from the traditional use of WSDL type declarations, our type system also controls that processes interact according to the prescriptions given by partner links. We expect that our theoretical framework can be the base of an effective software tool that can permit identifying errors during the WS-BPEL programs design phase.

Summary of the rest of the paper Section 2 provides a brief overview of WS-BPEL and WSDL. Syntax and operational semantics of WS-CALCULUS are defined in Sect. 3, while the type system and the type soundness results are presented in Sect. 4. Section 5 illustrates applications of our framework to modelling and analysis of a few business processes. Section 6 reviews related work, while Sect. 7 touches upon directions for future work.

The work we are going to present is an extended and revisited version of our former development introduced in [52]. The preliminary version used many notations and conventions that made the analysis more related to the peculiarities of the calculus than to the problems that might arise when designing business processes. The newer version is neater and more faithfully reflects the relationships between WSDL interfaces and WS-BPEL programs.

2 An overview of WS-BPEL and WSDL

In this paper, we refer to the current standard version of WS-BPEL, i.e. version 2.0 [61], and the version of WSDL it supports, i.e. version 1.1 [25]. We briefly describe them in this section.

WS-BPEL is essentially a linguistic layer on top of WSDL for describing the interactions between parties involved in an orchestration. The logic of the interactions is described in terms of structured patterns of communication activities composed through control flow constructs that enable the representation of complex structures. An orchestration consists of a *process* element containing one activity, a series of partner links, some variable declarations with specific correlation sets and the definition of some fault handlers. Activities are distinguished between *basic* and *structured* activities.

The basic activities enabling messages exchange are of three types: *invoke*, *receive* and *reply*. *invoke* is used to call web service operations, while *receive* is used to provide an operation and, hence, to get messages from the invoking partner; *receive* and *reply* can be assigned to the same operation for implementing a synchronous request-response communication pattern. The information is passed between different activities in an implicit way through the sharing of globally visible variables and is managed through the use of the basic activity *assign*. Other basic activities are: *empty*, to do nothing; *wait*, to delay execution for some amount of time; *throw*, to signal internal faults; *rethrow*, to propagate faults; *exit*, to immediately end a process instance; *compensate* and *compensateScope*, to invoke compensation handlers; *validate*, to validate variables; and *extensionActivity*, to support extensibility by allowing the definition of new activity types.

The structured activities for defining the control flow are: *sequence*, to execute activities sequentially; *if*, to execute activities conditionally; *while* and *repeatUntil*, to repetitively execute activities; *flow*, to execute activities in parallel; *pick*, to execute activities selectively; *forEach*, to (sequentially or in parallel) execute multiple activities; and *scope*, to associate handlers for exceptional events to a primary activity. The control flow can be constrained also by means of *flow links*. A *flow link* is a conditional transition that establishes dependencies among parallel activities by connecting a 'source' activity to a 'target' activity.

When a fault occurs within a scope, normal processing is terminated and the control is transferred to the corresponding *faultHandler*. Other handler definitions can be provided within a scope: a *compensationHandler*, to provide the activities to compensate the successfully executed primary activity; a *terminationHandler*, to control the forced termination of the primary activity; and some *eventHandler*s, to elaborate message or timeout events occurring during execution of the primary activity.

Collaborations with each of the service partners are expressed through WSDL interfaces. The roles within collaborations are specified at *port type* level in constructs called *partner links*. Port types, in turn, are simply collections of web service *operations*. The following example from [61, Sect. 6] illustrates the basic syntax for declaring partner link types:

```
<partnerLinkType name="BuyerSellerLink">
   <role name="Buyer"  portType="BuyerPortType" />
   <role name="Seller" portType="SellerPortType" />
</partnerLinkType>
```

In this example, a partner link of type `BuyerSellerLink` permits specifying a collaboration between a seller service (which provides the operations specified within the port type `SellerPortType`) and a buyer service (which provides the operations within the port type `BuyerPortType`). Such a partner link can be instantiated as follows:

```
<partnerLink name="buyer"
             partnerLinkType="BuyerSellerLink"
             myRole="Seller" partnerRole="Buyer" />
```

Thus, the business process plays the `Seller` role, while its partner plays the `Buyer` role.

Although partner links allow the actual partner service to be dynamically determined, this information is not enough to properly deliver messages to a business process. Indeed, since multiple instances of the same process can be simultaneously active because service operations can be independently invoked by several clients, messages need to be delivered not only to the correct service partner, but also to the correct instance of the process that the partner provides. To achieve this, WS-BPEL relies on the business data exchanged rather than on specific mechanisms, such as WS-Addressing [36] or other low-level methods based on SOAP headers. In other words, WS-BPEL incorporates a message correlation mechanism that exploits *correlation sets*, namely subsets of *message variables*, declaring the parts of a message that can be used to identify a process instance. This way, a message can be delivered to the correct instance on the basis of the values associated to the message variables, independently of any other delivering mechanism. For example, a correlation set containing a single variable storing the purchase order identifier could permit to identify the buyer and seller instances in the buyer-seller collaboration sketched above.

3 Defining business processes with WS-CALCULUS

This section introduces syntax and semantics of WS-CALCULUS (*web services calculus*). While the set of WS-BPEL constructs is not intended to be a minimal one, WS-CALCULUS is instead the result of the tension between handiness and expressiveness which is typical when designing a formalism. Thus, to keep the semantics of the language rigorous but still manageable, the design of WS-CALCULUS focusses on the 'procedural' part of WS-BPEL, which is sufficiently expressive to describe business processes in a primitive form, with special concern for modelling their asynchronous interactions in a network context. Therefore, we intentionally left out other aspects, including timed activities, flow graphs, and event, fault, compensation and termination handlers, which are not particularly relevant for our investigation about the relationship between WS-BPEL and WSDL.

We are indeed mainly interested in analysing the externally observable behaviour of business processes. Therefore, we disregard compensation as its only observable effect is to enable further interactions, which are supposed to undo the effect of previously executed activities, among the WS-BPEL process to be compensated and its partner services. Anyway, such interactions are not dealt with differently from the other ones, e.g. they are defined in the WSDL documents in the usual way. Similarly, we do not consider faults although they may slightly affect the business processes interfaces, i.e. the WSDL definitions. Of course, internal faults raised by the activity *throw* and propagated by *rethrow* (see Sect. 2) do not have externally observable effects. Instead, in case of synchronous interactions, the invoked process can send a fault back to the invoking process. Then, the WSDL definition of a request-response operation has to specify that a fault can be returned. Anyway, at the level of abstraction at which WS-CALCULUS operates, fault messages can be thought of as usual outgoing messages (as shown in the last example of Sect. 3.2). This not only permits to get a simpler formalization of the language, but is also reasonable from a practical point of view. In fact, the use of WSDL faults is not so common as service programmers rather tend to handle possible faults locally and return normal outgoing messages containing an explanation of the failure (see, e.g., the WSs listed in the well-known repository XMethods [69]). Instead, timed activities, as the activity *wait*, do not produce effects directly observable outside processes and do not affect WSDL interfaces. Some other WS-BPEL activities, such as e.g. *repeatUntil* and *forEach*, are disregarded as they can be reasonably encoded in the fragment we consider. Similarly, flow links are expressible in terms of variables and conditional tests (as in [53]).

3.1 Syntax

The *syntax* of WS-CALCULUS, given in Tables 1 and 2, is parameterised with respect to the following syntactic sets, which we assume to be disjoint: *variables* (ranged over by x, y), *basic values*[1] (*boolean*, *integer* and *string*, ranged over by b, n and s, respectively) and *addresses* (ranged over by λ). The language is also parametric with respect to a set of *operation names* (ranged over by o) and a set of *expressions* (ranged over by e).[2] The exact syntax of expressions is deliberately omitted; we just assume that they contain, at least, variables, basic values and addresses. *Join-variables*, that is variables prefixed by "!", are used to store those data that are important to identify process instances for message correlation. Indeed, "!" may be used in the argument of receive activities to indicate which variables are exploited for correlation purposes, thus requiring, for each of them, the received value to coincide with the corresponding value stored in the state (in WS-BPEL jargon, this is called the *correlation consistency constraint* [61, Sect. 9]).

Notationally, we will use u to range over communicable *values* (i.e. basic values and addresses), and w to range over *message variables* (i.e. variables and join-variables). Variables used to refer to remote nodes in communication activities will be called *partner links*. More specifically, in WS-CALCULUS, a partner link is a variable used to store and transmit the

[1] For the sake of simplicity, we only consider a minimal set of basic values that are sufficient for describing our examples. Of course, other kinds of basic values could be added and dealt with in a similar way.

[2] WS-CALCULUS is parametric w.r.t. the set of expressions as well as WS-BPEL is parametric w.r.t. the expression language supporting data manipulation. The default expression language for WS-BPEL processes is XPath 1.0 [26].

Table 1 WS-CALCULUS syntax: operation parameters and expressions

w	$::=$		*/ message variables /*
		x	*/ variable /*
	$\mid$	$!x$	*/ join-variable /*
u	$::=$		*/ values /*
		b	*/ boolean /*
	$\mid$	n	*/ integer /*
	$\mid$	s	*/ string /*
	$\mid$	λ	*/ address /*
e	$::=$	$x \mid u \mid \ldots$	*/ expressions /*

Table 2 WS-CALCULUS syntax

N	$::=$		*/ networks /*
		$\lambda :: \{c\}$	*/ single node /*
	$\mid$	$N \parallel N$	*/ node collection /*
c	$::=$		*/ components /*
		$m > a$	*/ process definition /*
	$\mid$	$m \gg a$	*/ process instance /*
	$\mid$	$\langle o : \bar{u} \rangle$	*/ request /*
	$\mid$	c, c	*/ component collection /*
a	$::=$		*/ activities /*
		$\mathbf{asg}(\bar{x} : \bar{e})$	*/ assign /*
	$\mid$	$\mathbf{inv}(x : o : \bar{y})$	*/ invoke /*
	$\mid$	$\mathbf{if}\, e\, \mathbf{then}\, a\, \mathbf{else}\, a$	*/ conditional /*
	$\mid$	$\mathbf{while}\, e\, \mathbf{do}\, a$	*/ iteration /*
	$\mid$	$a\, ;\, a$	*/ sequence /*
	$\mid$	$a \mid a$	*/ flow /*
	$\mid$	$\sum_{i \in I} \mathbf{rec}(x_i : o_i : \bar{w}_i)\, ;\, a_i$	*/ pick /*

address of a partner, i.e. a service that plays the *partnerRole* in the considered collaboration. Notation $\bar{\cdot}$ denotes tuples of objects, e.g. $\bar{x}$ is a tuple of variables $x_1, \ldots, x_n$. To avoid ambiguities, we assume that names of variables in the same tuple are pairwise distinct. All notations shall extend to tuples component-wise.

Networks are finite collections of nodes. A *node* can be thought of as an 'engine' for executing business processes and is written $\lambda :: \{c\}$. It is identified by its address λ that specifies where the behavioural components c are located. Such components generally offer a service, thus, following the *"everything is a service"* slogan, they will be sometimes called *services*. We will often use the address of the node to identify the components located there, i.e. given node $\lambda :: \{c\}$ we write 'service λ' to denote c.

Components c may be (business) *process definitions*, *process instances* or *requests*. Process definitions and instances, written $m > a$ and $m \gg a$ respectively, behave according to an *activity* description a and a *state* m. A state is a (possibly empty) collection of bindings of variables to values of the form u/x. We shall use notation $m \circ [u/x]$ for the state update recording the value u assigned to the variable x. Notably, state $[u/x, u'/y]$ can also be written as $[u/x] \circ [u'/y]$, or $[\,] \circ [u/x, u'/y]$ or, if $x \neq y$, $[u'/y, u/x]$. The state m in a process definition $m > a$ is a device that permits modelling an 'initial state' thus allowing to concisely express process definitions (without e.g. the need to specify a sequence of initialising assignments). It can also be used to statically initialize partner links at deployment time (as specified by the WS-BPEL attribute "initializePartnerRole" in [61, Sect. 6.2]). A *request*, written $\langle o : \bar{u} \rangle$, represents an invocation of a local operation that has still to be processed and contains the operation name o and the specific data $\bar{u}$ for its execution.

Activities may be *basic activities*, i.e. assignment $\mathbf{asg}(\cdot : \cdot)$ and service invocation $\mathbf{inv}(\cdot : \cdot : \cdot)$, or *structured activities*, i.e. conditional choice $\mathbf{if} \cdot \mathbf{then} \cdot \mathbf{else} \cdot$, iteration $\mathbf{while} \cdot \mathbf{do} \cdot$, sequential composition $\cdot ; \cdot$, parallel composition $\cdot \mid \cdot$ and external choice among receive activities $\sum_{i \in I} \mathbf{rec}(\cdot : \cdot : \cdot) ; \cdot$. Invoke and receive activities specify three arguments: a partner link identifying the partner service, the invoked/provided operation and a tuple of variables for storing the sent/received message values (notice that only receive activities can exploit join-variables). Whenever the external choice is between two activities a_1 and a_2, we shall simply write $a_1 + a_2$, and we shall write **empty** for $\sum_{i \in I} \mathbf{rec}(\cdot : \cdot : \cdot) ; \cdot$ when $I = \emptyset$. Usually we shall omit trailing occurrences of **empty**, writing a instead of $a ; \mathbf{empty}$. In addition, we use parentheses to resolve ambiguity and we let sequential composition have higher priority than flow and pick, i.e. $a_1 ; a_2 \mid a_3 ; a_4$ stands for $(a_1 ; a_2) \mid (a_3 ; a_4)$, and $\mathbf{rec}(x_1 : o_1 : \bar{w}_1) ; a_1 + \mathbf{rec}(x_2 : o_2 : \bar{w}_2) ; a_2$ stands for $(\mathbf{rec}(x_1 : o_1 : \bar{w}_1) ; a_1) + (\mathbf{rec}(x_2 : o_2 : \bar{w}_2) ; a_2)$.

We identify terms up to nodes and components reordering. This means, e.g., that $\lambda_1 :: \{ c_1 \} \parallel \lambda_2 :: \{ c_2 \}$ and $\lambda_2 :: \{ c_2 \} \parallel \lambda_1 :: \{ c_1 \}$, as well as $\lambda :: \{ c_1, c_2 \}$ and $\lambda :: \{ c_2, c_1 \}$, identify the same term. Moreover, for the sake of presentation (see, e.g., rule (N-INV) in Table 7), terminated instances of the form $m \gg \mathbf{empty}$ can be removed from/added to a component c, e.g. $\lambda_1 :: \{ m \gg a,\ m \gg \mathbf{empty} \}$ and $\lambda_1 :: \{ m \gg a \}$ identify the same term.

WS-CALCULUS constructs directly model the homonymous WS-BPEL activities, as shown in Table 3 (where some details, e.g. namespaces and correlation sets, have been omitted to make the reading of the code easier). WS-CALCULUS process instances, requests, network nodes and node collections have no counterpart in WS-BPEL: they have been introduced in WS-CALCULUS for modelling runtime aspects, not directly covered by the WS-BPEL specification but necessary for formally defining its semantics. Due to the quite direct correspondence of basic and structured activities, we can confidently state that WS-CALCULUS's formal semantics provides the corresponding fragment of WS-BPEL with a rigorous semantics that disambiguates the intricate and complex features of the language.

3.2 Examples

To better explain WS-CALCULUS primitives and peculiarities, we now present some simple examples.

Using partner links to provide and invoke operations Consider a typical scenario involving a *buyer* service instance (in other words, an instance of a process definition providing a service located at *buyer*) and a *seller* service instance.

$$\begin{aligned}
&\mathit{buyer} :: \{ [\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}/y_{id}, d/y] \gg \mathbf{inv}(x_S : o_{req} : y_B, y_{id}, y) ; \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res}) \} \\
&\parallel \mathit{seller} :: \{ [\lambda/x_B] \gg \mathbf{rec}(x_B : o_{req} : z_B, z_{id}, z) ; \mathbf{asg}(x_B : z_B) ; \\
&\qquad\qquad\qquad\qquad\qquad a_{elaborating} ; \mathbf{inv}(x_B : o_{res} : z_{id}, z_{res}) \}
\end{aligned}$$

Table 3 Correspondence between WS-CALCULUS constructs and WS-BPEL activities

WS-CALCULUS	WS-BPEL

$\mathbf{asg}(x_1, \ldots, x_n : e_1, \ldots, e_n)$	`<assign>`
	  `<copy> <from>` e_1 `</from> <to> x_1 </to> </copy>`
	  $\ldots$
	  `<copy> <from>` e_n `</from> <to> x_n </to> </copy>`
	`</assign>`

$\mathbf{inv}(x : o : \bar{y})$	`<invoke partnerLink="x" operation="o"`
	        `inputVariable="y" />`

$\mathbf{rec}(x : o : \bar{w})$	`<receive partnerLink="x" operation="o"`
	          `variable="w" />`

$\mathbf{if}\, e\, \mathbf{then}\, a_1\, \mathbf{else}\, a_2$	`<if> <condition>` e `</condition>` a_1 `<else>` a_2 `</else> </if>`
$\mathbf{while}\, e\, \mathbf{do}\, a$	`<while> <condition>` e `</condition>` a `</while>`
$a_1 \,;\, a_2$	`<sequence>` a_1 a_2 `</sequence>`
$a_1 \mid a_2$	`<flow>` a_1 a_2 `</flow>`
$\mathbf{empty} = \sum_{i \in \emptyset} \mathbf{rec}(x_i : o_i : \bar{w}_i) \,;\, a_i$	`<empty />`

$\sum_{i \in I} \mathbf{rec}(x_i : o_i : \bar{w}_i) \,;\, a_i$
with $\mid I \mid = n > 1$
	`<pick>`
	  `<sequence>`
	    `<onMessage partnerLink="x_1" ... />` a_1
	  `</sequence>`
	  $\ldots$
	  `<sequence>`
	    `<onMessage partnerLink="x_n" ... />` a_n
	  `</sequence>`
	`</pick>`

$m > a$	`<process> a </process>`

The buyer instance specifies the partner link x_S to interact with the seller instance both to invoke the operation o_{req} provided by the seller (i.e. to send the request) and to receive the response along the operation o_{res}. Similarly, the seller instance specifies the partner link x_B to receive the request message and to invoke the operation o_{res} offered by the buyer instance. The buyer sends to (the address of) the seller a request containing the callback address *buyer* to be used by the seller to send back the response message, the instance identifier *id* and some other data d, and waits for the response message. When receive activities are elaborated, they always implicitly act on the node where the instance is running. Therefore, the values of the partner links x_S and x_B stored in the states, i.e. *seller* and λ respectively, that are the addresses of the services playing the partner role in the interaction, are not used when elaborating receive activities (as required in [61, Sect. 10.4]). As we will see in Sect. 4, although not necessary for the operational semantics, partner links in the receive activities will turn out to be essential for typing WS-CALCULUS terms. Activity $a_{elaborating}$, which has been left unspecified, elaborates the request data stored in the variable z, generates a response message and assigns it to the variable z_{res}.

Thus, regardless of the value of x_B, the whole network evolves firstly to:

$$\begin{aligned}&\mathit{buyer} :: \{\,[\,\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}/y_{id}, d/y] \gg \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res})\,\}\\ &\parallel\ \mathit{seller} :: \{\,[\,\lambda/x_B] \gg \mathbf{rec}(x_B : o_{req} : z_B, z_{id}, z)\,;\, \mathbf{asg}(x_B : z_B)\,;\\ &\qquad\qquad a_{elaborating}\,;\, \mathbf{inv}(x_B : o_{res} : z_{id}, z_{res})\,,\\ &\qquad \langle o_{res} : (\mathit{buyer}, \mathit{id}, d)\rangle\,\}\end{aligned}$$

and then to

$$\begin{aligned}&\mathit{buyer} :: \{\,[\,\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}/y_{id}, d/y] \gg \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res})\,\}\\ &\parallel\ \mathit{seller} :: \{\,[\,\lambda/x_B, \mathit{buyer}/z_B, \mathit{id}/z_{id}, d/z] \gg \mathbf{asg}(x_B : z_B)\,;\\ &\qquad\qquad a_{elaborating}\,;\, \mathbf{inv}(x_B : o_{res} : z_{id}, z_{res})\,\}\end{aligned}$$

Remarkably, in an asynchronous setting, to reply to an invocation, the partner link used for the callback must be explicitly initialized with the received invoker address. To this aim, the seller service performs an assign activity that replaces the value of x_B with *buyer*.

Alternatively, the seller could directly update the partner link x_B by means of the initial receive activity thus saving the assign activity, as shown in the following term:

$$\mathit{seller} :: \{\,[\,\lambda/x_B] \gg \mathbf{rec}(x_B : o_{req} : x_B, z_{id}, z)\,;\, a_{elaborating}\,;\, \mathbf{inv}(x_B : o_{res} : z_{id}, z_{res})\,\}$$

Creating process instances from process definitions Process definitions must be deployed in order to be invoked and, hence, instantiated. For example, the process definition deployed in the following node can be thought of as a template for creating instances of the buyer service described in the previous example:

$$\begin{aligned}&\mathit{buyer} :: \{\,\langle o_{new} : \mathit{id}_1, d_1\rangle,\ \ \langle o_{new} : \mathit{id}_2, d_2\rangle,\\ &\qquad [\,\mathit{seller}/x_S, \mathit{buyer}/y_B] > \mathbf{rec}(x : o_{new} : y_{id}, y)\,;\\ &\qquad\qquad \mathbf{inv}(x_S : o_{req} : y_B, y_{id}, y)\,;\\ &\qquad\qquad \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res})\,\}\end{aligned}$$

As in [61], the creation of a process instance is always implicit. The receive activity providing the operation o_{new} can consume the two requests $\langle o_{new} : \mathit{id}_1, d_1\rangle$ and $\langle o_{new} : \mathit{id}_2, d_2\rangle$, and cause two new instances of the process to be created as follows:

$$\begin{aligned}&\mathit{buyer} :: \{\,[\,\mathit{seller}/x_S, \mathit{buyer}/y_B] > \mathbf{rec}(x : o_{new} : y_{id}, y)\,;\\ &\qquad\qquad \mathbf{inv}(x_S : o_{req} : y_B, y_{id}, y)\,;\\ &\qquad\qquad \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res}),\\ &\qquad [\,\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}_1/y_{id}, d_1/y] \gg \mathbf{inv}(x_S : o_{req} : y_B, y_{id}, y)\,;\\ &\qquad\qquad \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res}),\\ &\qquad [\,\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}_2/y_{id}, d_2/y] \gg \mathbf{inv}(x_S : o_{req} : y_B, y_{id}, y)\,;\\ &\qquad\qquad \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res})\,\}\end{aligned}$$

Using join-variables to implement correlation Join-variables can only occur within message variables of receive activities. The term "join", borrowed from [61], indicates the variables used for message correlation. Indeed, join-variables permit maintaining values that are relevant for delivering messages over multiple service instances.

When a message variable is a join-variable (i.e. it is prefixed by "!"), the related receive activity must attempt to initiate it, if the join-variable has not been yet initiated. Instead, if the join-variable is already initiated, its value and the related received datum must be identical. For example, consider a variant of the buyer-seller scenario where, after some computations, we obtain the following network:

$$\begin{aligned}&\mathit{buyer} :: \{\, [\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}_1/y_{id}, d_1/y] \gg \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res}),\\ &\qquad\qquad [\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}_2/y_{id}, d_2/y] \gg \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res})\,\}\\ &\parallel\ \mathit{seller} :: \{\, [\mathit{buyer}/x_B, \mathit{id}_1/z_{id}, d_1/z, r/z_{res}] \gg \mathbf{inv}(x_B : o_{res} : z_{id}, z_{res})\,\}\end{aligned}$$

The response message sent by the seller service instance can be intercepted by both instances of the buyer service because no correlation information is specified. Thus, although the message contains the identifier id_1, it can be consumed by the instance characterized by the identifier id_2:

$$\begin{aligned}&\mathit{buyer} :: \{\, [\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}_1/y_{id}, d_1/y] \gg \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res}),\\ &\qquad\qquad [\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}_1/y_{id}, d_2/y, r/y_{res}] \gg \mathbf{empty}\,\}\\ &\parallel\ \mathit{seller} :: \{\, [\mathit{buyer}/x_B, \mathit{id}_1/z_{id}, d_1/z, r/z_{res}] \gg \mathbf{empty}\,\}\end{aligned}$$

Instead, to identify service instances thus guaranteeing that exactly the service instance identified by id_1 receives the message, we can exploit the correlation mechanism as follows:

$$\begin{aligned}&\mathit{buyer} :: \{\, [\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}_1/y_{id}, d_1/y] \gg \mathbf{rec}(x_S : o_{res} :!y_{id}, y_{res}),\\ &\qquad\qquad [\mathit{seller}/x_S, \mathit{buyer}/y_B, \mathit{id}_2/y_{id}, d_2/y] \gg \mathbf{rec}(x_S : o_{res} :!y_{id}, y_{res})\,\}\\ &\parallel\ \mathit{seller} :: \{\, [\mathit{buyer}/x_B, \mathit{id}_1/z_{id}, d_1/z, r/z_{res}] \gg \mathbf{inv}(x_B : o_{res} : z_{id}, z_{res})\,\}\end{aligned}$$

Now, the message sent by the seller instance can be elaborated only by the receive activity within the first instance of the buyer service, for which id_1 coincides with the value recorded by y_{id}. Notably, the seller service does not need join-variables to reply to the buyer service.

Therefore, to ensure that the messages received by the buyer service specified in the *creating process instances* example are correctly delivered, the process definition at *buyer* has to be modified as follows:

$$\begin{aligned}[\mathit{seller}/x_S, \mathit{buyer}/y_B] > {}&\mathbf{rec}(x : o_{new} : y_{id}, y)\,;\\ &\mathbf{inv}(x_S : o_{req} : y_B, y_{id}, y)\,;\\ &\mathbf{rec}(x_S : o_{res} :!y_{id}, y_{res})\end{aligned}$$

Of course, a service can specify more than one join-variable for the same receive activity. For instance, the process definition above could use two join-variables $!y_{buyerName}$ and $!y_{time\&date}$ to univocally identify a buyer instance:

$$\begin{aligned}[\mathit{seller}/x_S, \mathit{buyer}/y_B] > {}&\mathbf{rec}(x : o_{new} : y_{buyerName}, y_{time\&date}, y)\,;\\ &\mathbf{inv}(x_S : o_{req} : y_B, y_{buyerName}, y_{time\&date}, y)\,;\\ &\mathbf{rec}(x_S : o_{res} :!y_{buyerName}, !y_{time\&date}, y_{res})\end{aligned}$$

Multiple service instantiation: shared join-variables When some of the start activities of a process definition are receives put in a parallel composition, then the firstly executed receive creates a new instance that then performs the remaining receives. To correlate the messages directed to the same instance, as prescribed in [61, Sect. 10.4], the receive activities need to share some join-variables.

To illustrate, let us consider a service implementing the well-known children game Rock/Paper/Scissors. The service is defined to receive two turn-messages (through operation o_{turn}) from two players A and B, as described by the following definition:

$$\begin{aligned}\lambda :: \{\, [\lambda/x_C] > (\, \mathbf{rec}(x_A : o_{turn} : x_A, y_A, !z) \mid \mathbf{rec}(x_B : o_{turn} : x_B, y_B, !z)\,)\,;\\ a_{winner}\,;\, (\, \mathbf{inv}(x_A : o_{win} : x_C, z) \mid \mathbf{inv}(x_B : o_{win} : x_C, z)\,)\,\}\end{aligned}$$

The arrival order of the turn-messages cannot be statically predicted, thus the possibility to instantiate the service in multiple ways may be useful. To correlate messages for the same instance, the join-variable $!z$ is used. After the turn-messages from both players have been received, the activity a_{winner} determines the winner and assigns its address to the variable x_C. The result is then communicated to the players. By assuming that in case of equal turns the transmitted address is that of the service provider λ, a_{winner} can be defined as follows:

$$\mathbf{if}\ y_A \neq y_B\ \mathbf{then}\ (\mathbf{if}\ e_{Awin}\ \mathbf{then}\ \mathbf{asg}(x_C : x_A)\ \mathbf{else}\ \mathbf{asg}(x_C : x_B))\ \mathbf{else\ empty}$$

where e_{Awin} is the boolean condition $(y_A = \text{“}paper\text{”} \wedge y_B = \text{“}rock\text{”}) \vee (y_A = \text{“}scissor\text{”} \wedge y_B = \text{“}paper\text{”}) \vee (y_A = \text{“}rock\text{”} \wedge y_B = \text{“}scissor\text{”})$ that holds *true* when the player A wins.

Handling fault messages As we said before, when a WS-BPEL process instance invokes an operation provided by a service partner, this latter might return in response a fault message. This results in a fault internal to the process instance that can be caught and handled by using a fault handler.

In our framework, fault messages are dealt with like usual messages, then they are simply caught by receive activities and handled by their continuations. To illustrate such situation, let us consider the following variant of the buyer-seller scenario described in the first example of this section:

$$\begin{aligned}&buyer :: \{\, [\,seller/x_S, buyer/y_B, id/y_{id}, d/y] \gg \mathbf{inv}(x_S : o_{req} : y_B, y_{id}, y)\,;\\ &\qquad\qquad (\, \mathbf{rec}(x_S : o_{res} : y_{id}, y_{res})\,;\, a_{okContinuation}\\ &\qquad\qquad +\ \mathbf{rec}(x_S : o_{fault} : y_{id}, y_{err})\,;\, a_{fh}\,)\,\}\\ &\parallel\ seller :: \{\, [\,\lambda/x_B] \gg \mathbf{rec}(x_B : o_{req} : x_B, z_{id}, z)\,;\\ &\qquad\qquad \mathbf{if}\ isOk(z)\ \ \mathbf{then}\ a_{elaboratingAndReplying}\ \mathbf{else}\ \mathbf{inv}(x_B : o_{fault} : z_{id}, z)\,\}\end{aligned}$$

Here, the seller instance checks the request data (stored in z) by means of an unspecified function $isOk(\cdot)$ and, if the check fails, the request data are sent back by invoking the operation o_{fault}. The buyer instance exploits a pick activity to choose between two alternative behaviours; this way, the reception of an invocation of operation o_{fault}, responsible for receiving fault messages, triggers the execution of the fault handling activity a_{fh}.

3.3 Operational semantics

We will only consider networks that are *well-formed* in the sense that they comply with the following two syntactic constraints. Firstly, distinct nodes have different addresses (so that nodes are uniquely identified by their address). Secondly, at least one start activity of every process definition must be a pick (as required in [61, Sect. 5.5]), where the *start activities* of $m > a$ are deemed to be all those activities in a that are not (syntactically) preceded by other ones, considering that a_1 precedes a_2 in $a_1\,;a_2$ and that, for any $i \in I$, $\mathbf{rec}(x_i : o_i : \bar{w}_i)$ precedes a_i in $\sum_{i\in I} \mathbf{rec}(x_i : o_i : \bar{w}_i)\,;a_i$. This means that process definitions are sort of templates that must be necessarily instantiated in order to get executable entities and instantiation takes place when one of the receives of a pick activity consumes a request message and, hence, causes the creation of a new process instance. Well-formedness could be easily checked through a standard (and trivial) static analysis.

Table 4 WS-CALCULUS operational semantics: instances (symmetric of rules (FLOW$_1$) and (FLOW$_2$) omitted)

$$m \gg \mathbf{asg}(\bar{x} : \bar{e}) \xrightarrow{\tau} m \circ [\,m(\bar{e})/\bar{x}\,] \gg \mathbf{empty} \quad (\text{ASSIGN})$$

$$m \gg \mathbf{inv}(x : o : \bar{y}) \xrightarrow{m(x)\,\langle o : m(\bar{y})\rangle} m \gg \mathbf{empty} \; (\text{INVOKE})$$

$$m \gg \textstyle\sum_{i \in I} \mathbf{rec}(x_i : o_i : \bar{w}_i)\,;\, a_i \xrightarrow{(o_k : \bar{w}_k)} m \gg a_k \quad \textit{with } k \in I \;\; (\text{PICK})$$

$$\frac{m \gg a_1 \xrightarrow{\alpha} m' \gg a_1'}{m \gg a_1\,;\,a_2 \xrightarrow{\alpha} m' \gg a_1'\,;\,a_2} \; (\text{SEQ}_1) \qquad \frac{m \gg a_1 \xrightarrow{\alpha} m' \gg a_1'}{m \gg a_1 \mid a_2 \xrightarrow{\alpha} m' \gg a_1' \mid a_2} \; (\text{FLOW}_1)$$

$$m \gg \mathbf{empty}\,;\,a \xrightarrow{\tau} m \gg a \; (\text{SEQ}_2) \qquad m \gg \mathbf{empty} \mid a \xrightarrow{\tau} m \gg a \; (\text{FLOW}_2)$$

$$\frac{m(e) = \mathbf{true}}{m \gg \mathbf{if}\,e\,\mathbf{then}\,a_1\,\mathbf{else}\,a_2 \xrightarrow{\tau} m \gg a_1} \; (\text{IF}_1) \qquad \frac{m(e) = \mathbf{false}}{m \gg \mathbf{if}\,e\,\mathbf{then}\,a_1\,\mathbf{else}\,a_2 \xrightarrow{\tau} m \gg a_2} \; (\text{IF}_2)$$

$$\frac{m(e) = \mathbf{true}}{m \gg \mathbf{while}\,e\,\mathbf{do}\,a \xrightarrow{\tau} m \gg a\,;\,\mathbf{while}\,e\,\mathbf{do}\,a} \; (\text{WH}_1) \qquad \frac{m(e) = \mathbf{false}}{m \gg \mathbf{while}\,e\,\mathbf{do}\,a \xrightarrow{\tau} m \gg \mathbf{empty}} \; (\text{WH}_2)$$

The *operational semantics* of WS-CALCULUS is defined by a *reduction relation* over networks, written $\rightarrowtail$. It relies on a labelled transition relation $\xrightarrow{\alpha}$ over process instances, where label α is generated by the following productions:

$$\alpha \;::=\; \tau \;\mid\; \lambda\,\langle o : \bar{u}\rangle \;\mid\; (o : \bar{w})$$

The meaning of labels is as follows: τ denotes a silent internal action, $\lambda\,\langle o : \bar{u}\rangle$ denotes invocation of operation o located at λ with data $\bar{u}$, and $(o : \bar{w})$ denotes waiting along operation o using message variables $\bar{w}$ for a message from any partner.

To define $\xrightarrow{\alpha}$ we need a partial function for evaluating expressions: it takes an expression and returns a value. Expressions to be evaluated can contain variables; thus, evaluation of an expression e uses a state m storing the values of the variables that may occur within e. We write $m(e)$ such an evaluation function, but we do not explicitly define it because the exact syntax of expressions is deliberately not specified (in fact, WS-CALCULUS is parameterised w.r.t. the syntax of expressions). We just assume that expressions, to be successfully evaluated, may not contain uninitialized variables.

Now, $\xrightarrow{\alpha}$ can be defined as the least relation over process instances induced by the rules in Table 4. Most of the rules are straightforward, we only remark a few points. Rule (ASSIGN) states that an assignment can proceed only if the expressions in its argument can be evaluated (otherwise, the evaluation function $m(\bar{e})$ is undefined). In case of successful evaluation, the state of the instance is updated. Similarly, a service invocation can proceed only if the variables in its argument are initialized[3] (rule (INVOKE)). Rule (PICK) states that the pick activity can execute any of its receive activities and then proceed accordingly. Rules (FLOW$_1$) and (FLOW$_2$) state that executions of two parallel activities are interleaved and that **empty** activities can be removed. For simplicity, symmetric rules for parallel composition have been omitted.

To define the reduction relation, we need a mechanism for checking if the assignments of values $\bar{u}$ to message variables $\bar{w}$ comply with a given state m. Such a mechanism is rendered

[3]This slightly differs from WS-BPEL specification, where taking place of abnormal situations (like, e.g., execution of an invoke activity whose argument contains uninitialized variables) raises faults. Our simplification is however allowable since it does not change processes' interaction ability, which is the aspect we focus on in this paper.

Table 5 WS-CALCULUS operational semantics: matching function

$m \diamond (x, u) = [u/x]$	$\dfrac{m(x) = u \ \vee\ m(x)\ \mathit{undefined}}{m \diamond (!x, u) = [u/x]}$	$\dfrac{m \diamond (w, u) = m_1 \quad m \diamond (\bar{w}, \bar{u}) = m_2}{m \diamond ((w, \bar{w}), (u, \bar{u})) = m_1 \circ m_2}$

Table 6 WS-CALCULUS operational semantics: is there any active receive matching $\langle o : \bar{u}\rangle$?

$\mathit{arec}((c_1, c_2), o, \bar{u})$	if $\mathit{arec}(c_1, o, \bar{u})$ or $\mathit{arec}(c_2, o, \bar{u})$	$\mathit{arec}(m \gg a, o, \bar{u})$	if $\mathit{arec}_m(a, o, \bar{u})$
$\mathit{arec}_m(a_1 \mid a_2, o, \bar{u})$	if $\mathit{arec}_m(a_1, o, \bar{u})$ or $\mathit{arec}_m(a_2, o, \bar{u})$	$\mathit{arec}_m(a_1\, ;\, a_2, o, \bar{u})$	if $\mathit{arec}_m(a_1, o, \bar{u})$
$\mathit{arec}_m(\sum_{i \in I} \mathbf{rec}(x_i : o_i : \bar{w}_i)\, ;\, a_i, o, \bar{u})$	if $\exists k \in I$ such that $o_k = o$ and $m \diamond (\bar{w}_k, \bar{u}) = m'$		

Table 7 WS-CALCULUS operational semantics: networks (symmetric of rule (N-PAR) omitted)

$$\frac{m \gg a \xrightarrow{\lambda_2\, \langle o:\bar{u}\rangle} m \gg a'}{\lambda_1 :: \{m \gg a,\ c_1\} \parallel \lambda_2 :: \{c_2\} \rightarrowtail \lambda_1 :: \{m \gg a',\ c_1\} \parallel \lambda_2 :: \{\langle o : \bar{u}\rangle,\ c_2\}} \ (\text{N-INV})$$

$$\frac{m \gg a \xrightarrow{(o:\bar{w})} m \gg a' \qquad m \diamond (\bar{w}, \bar{u}) = m'}{\lambda :: \{m \gg a,\ \langle o : \bar{u}\rangle,\ c\} \rightarrowtail \lambda :: \{m \circ m' \gg a',\ c\}} \ (\text{N-REC})$$

$$\frac{m \gg a \xrightarrow{(o:\bar{w})} m \gg a' \qquad m \diamond (\bar{w}, \bar{u}) = m' \qquad \neg \mathit{arec}(c, o, \bar{u})}{\lambda :: \{m > a,\ \langle o : \bar{u}\rangle,\ c\} \rightarrowtail \lambda :: \{m > a,\ m \circ m' \gg a',\ c\}} \ (\text{N-START})$$

$$\frac{m \gg a \xrightarrow{\tau} m' \gg a'}{\lambda :: \{m \gg a,\ c\} \rightarrowtail \lambda :: \{m' \gg a',\ c\}} \ (\text{N-TAU}) \qquad \frac{N_1 \rightarrowtail N_1'}{N_1 \parallel N_2 \rightarrowtail N_1' \parallel N_2} \ (\text{N-PAR})$$

as a partial function, written $m \diamond (\bar{w}, \bar{u})$, that in case the check succeeds, returns a collection of bindings m' implementing the assignments. It is defined inductively on the structure of $\bar{w}$ by the rules in Table 5. These rules state that $\bar{w}$ and $\bar{u}$ must have the same number of fields and corresponding fields match; a variable x matches any value u by returning the binding $[u/x]$; the same binding is also returned when a join-variable $!x$ matches a value u, which happens when either $m(x)$ does match with u or $m(x)$ is undefined.

We also exploit the predicate $\mathit{arec}(c, o, \bar{u})$, defined in Table 6 inductively on the syntax of components, to detect if there exists an active receive activity within a process instance in c matching the message $\langle o : \bar{u}\rangle$. This predicate relies on the auxiliary predicate $\mathit{arec}_m(a, o, \bar{u})$ that checks an activity a for the presence of a receive matching $\langle o : \bar{u}\rangle$ and complying with state m. The meaning of the rules is straightforward; notably, the predicates are undefined for process definitions and activities conditional choice and iteration.

Finally, we can define the reduction relation $\rightarrowtail$ as the least relation over networks induced by the rules in Table 7. Let us now comment on the rules. Rule (N-INV) states that service invocation corresponds to adding a request message to the dataspace of the invoked node. The request is a pair, containing the name of the invoked operation o and the message $\bar{u}$ (i.e. the data to be passed to o). Hence, the invocation of a remote service is asynchronous because the invoker can proceed before its request is elaborated. Notice also that a request message does not automatically collect the address of the invoker. This choice is dictated by our aim at modelling service communications in a real loose-coupled way where no default addressing mechanism is assumed. Anyway, to allow the receiver to send a callback message, the invoker can explicitly add its address to the message as shown in the buyer-seller scenario described in Sect. 3.2. Interestingly, rule (N-INV) requires two nodes because request messages can only be exchanged with existing 'external nodes'. In other words, two components located at the same node cannot communicate using this mechanism. Rule (N-REC) states that a receive activity cannot progress until a matching request has been received. Thus, differently from an invoke activity, it is blocking. Requests are delivered to the correct process instance by exploiting the operation name contained in the request, which must be identical to the name occurring in the label of the transition performed by the process instance, and by exploiting the correlation mechanism implemented by the matching function. In practice, the values contained in the request and corresponding to join-variables within the receive argument permit to determine the correct instance to which the request must be delivered. When the reduction takes place, the matching request is consumed and the state of the instance is updated with the corresponding assignments. Rule (N-START) permits to create a new process instance on receipt of a request that cannot be delivered to any existing instance. The premise $\neg arec(c, o, \bar{u})$ prevents interferences with rule (N-REC) as illustrated by the following example:

$$\begin{aligned}\lambda :: \{\, & [\,] > (\,\mathbf{rec}(x : o_1 :!y) \mid \mathbf{rec}(x : o_2 :!y)\,)\,;a,\\ & [\,10/y\,] \gg \mathbf{rec}(x : o_1 :!y)\,;a,\\ & \langle o_1 : 10\rangle\,\}\end{aligned}$$

where only the process instance $[\,10/y\,] \gg \mathbf{rec}(x : o_1 :!y)\,;a$ can proceed by consuming the request and evolving as follows:

$$\lambda :: \{\,[\,] > (\,\mathbf{rec}(x : o_1 :!y) \mid \mathbf{rec}(x : o_2 :!y)\,)\,;a,\ \ [\,10/y\,] \gg a\,\}$$

Notably, incidental receive conflicts among process instances or among process definitions are non-deterministically resolved. In such cases, the programmer has not properly set the correlation values thus the correlation mechanism cannot uniquely identify a service instance. Receive conflicts within the same process instance are dealt with similarly.[4] Rule (N-TAU) establishes that conditional and iterative constructs can silently choose and evolve, and that states can be silently updated. Notably, the state update semantics is simplified by the fact that the effect of an assignment is global w.r.t. a process instance. Finally, rule (N-PAR) states that if a part of a larger network evolves, the whole network evolves accordingly.

In Sect. 3.2, we have shown a few examples of WS-CALCULUS terms behaving as expected. On the contrary, here we want to illustrate some erroneous network configurations that we would like not to reach while computation proceed, but cannot do so if we only rely on the operational semantics. It is worth noting that in all considered examples WS-CALCULUS operational semantics does not point out any runtime error, at most the computation gets stuck.

Two trivial examples are

$$[\,2/x, \mathbf{true}/y\,] \gg \mathbf{asg}(x : y)$$

where a boolean value is going to be assigned to the integer variable x, and

$$[\,2/x\,] \gg \mathbf{inv}(x : o : y)$$

where x is going to be used as a partner link. Furthermore, since WS-CALCULUS, as well as WS-BPEL, is parametric with respect to the expression language, it may be equipped with operators for manipulating basic values, such as integer addition or boolean negation.

[4] Actually, this slightly differs from WS-BPEL specification that prescribes to raise a fault. Again, our simplification is allowable as this fault is an internal one.

Applying them to type-incompatible values can lead to erroneous network configurations, as in

$$[\,2/x\,] \gg \mathbf{asg}(y : \mathbf{not}\, x)$$

While the above situations indicate an error involving a single component, there are other cases where the error is related to wrong interactions between process instances and partner services. For example, a typical problem occurring in an asynchronous context is represented by the following network

$$\begin{array}{l} A :: \{\, [\,\text{“}ok\text{”}/y\,] \gg \mathbf{rec}(x_B : o_{res} : x_B)\, ; \mathbf{inv}(x_B : o_{res} : y)\,\} \\ \parallel\ B :: \{\, [\,A/x_A,\, B/z\,] \gg \mathbf{inv}(x_A : o_{req} : z)\,\} \end{array}$$

Process B sends its address to A that uses this value for initializing its partner link x_B. Thus, after three reduction steps, the whole network evolves to

$$\begin{array}{l} A :: \{\, [\,\text{“}ok\text{”}/y,\, B/x_B\,] \gg \mathbf{inv}(x_B : o_{res} : y)\,\} \\ \parallel\ B :: \{\, [\,A/x_A,\, B/z\,] \gg \mathbf{empty}\,\} \end{array}$$

Now, process A expects the partner identified by x_B, that is B, to provide a reply operation o_{res} to be invoked. Instead, B does not provide such an operation, thus, the collaboration between A and B cannot be further carried out.

All the situations above are actually considered as errors falling within the WS-BPEL fault classification [61, Appendix A and B]. For example, the first error above is classified as *SA00043* stating that assign activities must handle type-compatible variables. Although the WS-BPEL classification takes into account many different kinds of errors, as e.g. undefined variables and partner link uniqueness, in our opinion, some kind of errors dealing with interactions between process instances and partner services are overlooked. The task of the typing discipline we introduce in the next section is to statically prevent that situations like those we have just described can take place at runtime.

4 A typing discipline for collaborating services

The types for WS-CALCULUS we present in this section are closely inspired by WSDL type definitions. Actually, they do not contain any information in addition to that which can be extracted from the WSDL document associated to a WS-BPEL process specification. The assignments of types to WS-CALCULUS objects (i.e. values, variables, partner links, addresses, and nodes) correspond to the type declarations that are found in the WS-BPEL process specifications. The type system for WS-CALCULUS uses the type information both to prevent assign or receive activities from associating values with variables which are not *type-compatible* and to implement a sort of ‘partner link control’ that checks the ability of a process to invoke the partner specified by a partner link and the ability of that partner to asynchronously respond. As shown in the previous section, the interaction can take place by means of the transmission of the process address to the partner, hence by relying on an address-passing communication mechanism. Such an aspect complicates the definition of our type system but, at the same time, makes it more useful to single out those processes that are not able to carry out successfully the business collaboration with their partners.

Table 8 Syntax of types

δ	$::=$		*/ types /*
		υ	*/ value type /*
	$\mid$	ℓ	*/ partner link type /*
υ	$::=$		*/ value types /*
		β	*/ basic type /*
	$\mid$	γ	*/ address type /*
β	$::=$	BOOL $\mid$ INT $\mid$ STR	*/ basic types /*
γ	$::=$	$\{\pi_i\}_{i\in I}$	*/ address types /*
π	$::=$	$[o_i\langle\bar{\upsilon}_i\rangle]_{i\in I}$	*/ port types /*
ℓ	$::=$	$\langle\pi_1,\pi_2\rangle$	*/ partner link types /*
ω	$::=$	$\{x_i : \delta_i\}_{i\in I}$	*/ local types /*

4.1 Syntax of types

As previously said, a WS-CALCULUS node can be thought of as an 'engine' for executing business processes. Now that types come into the picture, nodes are enriched with a declarative part ω that defines their type. *Typed nodes* will be written as $\lambda ::^{\omega} \{c\}$, while *typed networks* are finite collections of typed nodes. The type of a node collects all information about the format of the messages exchanged in collaborations involving the components running at the node. It also specifies the type of each such collaboration, the so-called *partner link type*, as a pair of roles such that one role depends on the other. In practice, it represents the type information that can be extracted from the WSDL and WS-BPEL documents of the processes running there[5]. For example, in an asynchronous request-response interaction, the role of the service provider, which is the one that performs the receive activity over the request operation, must be paired with the role of the requestor, which performs the receive activity over the callback operation. Roles are rendered as port types; therefore, as in the case of WSDL declarations, a collaboration is expressed at the port type level and not at the operation level. Indeed, for WS-BPEL (and, hence, also for WSDL), this choice is dictated by the need to preserve compositionality and loose coupling while modelling asynchronous collaborating services.

The syntax of types is defined in Table 8. A *local type* ω consists of definitions of *value types* υ and *partner link types* ℓ. The two kinds of value types are *basic types* β (we only consider BOOL, INT and STR types) and *address types* γ. An address type is a collection of *port types* $\{\pi_i\}_{i\in I}$, where π_i range over port types. The values determined by an address type are addresses, e.g. if $x : \{\pi_1, \pi_2, \pi_3\}$ is an address type definition then we can only assign to x addresses compatible with port types π_1, π_2 and π_3. Port types $[o_i\langle\bar{\upsilon}_i\rangle]_{i\in I}$ are collections of *operation types* $o\langle\bar{\upsilon}\rangle$, where the tuple $\bar{\upsilon}$ is the type of the arguments required by the operation o. It is worth noting that basic values and addresses are the only values exchanged among WS-CALCULUS terms. Partner link types are defined as pairs of port types $\langle\pi_1, \pi_2\rangle$,

[5] Usually, any WS-BPEL process has its own associated WSDL document. Instead, in WS-CALCULUS, for the sake of simplicity, process definitions located at the same node $\lambda ::^{\omega} \{c\}$ share the same node type ω.

Table 9 Correspondence between WS-CALCULUS types and WSDL & WS-BPEL type declarations

WS-CALCULUS types	WSDL & WS-BPEL
BOOL	`xsd:boolean`
INT	`xsd:integer`
STR	`xsd:string`
γ	`wsa:EndpointReference`

$\pi = [o_1\langle\bar{v}_1\rangle, \ldots, o_n\langle\bar{v}_n\rangle]$	`<portType name="π">`
	`  <operation name="o_1">`
	`    <input message="v_1" />`
	`  </operation>`
	`  ...`
	`  <operation name="o_n">`
	`    <input message="v_n" />`
	`  </operation>`
	`</portType>`

$\ell = \langle\pi_1, \pi_2\rangle$	`<partnerLinkType name="ℓ">`
	`  <role name="my" portType="π₁" />`
	`  <role name="partner" portType="π₂" />`
	`</partnerLinkType>`

where π_1 refers to the role played by the considered service and π_2 to the role played by the partner service. To specify collaborations where a single port type (i.e. a single role) suffices, we use the symbol [/] to denote an empty port type.

Table 9 sheds light on the correspondence between WS-CALCULUS types and WSDL plus WS-BPEL type declarations. In particular, consider that address types are usually specified in WSDL by relying on WS-Addressing [36] 'endpoint reference types'. It is worth noting that type declarations within a local type ω are specified inside the WS-BPEL document; in particular, declarations of the form $x : v$ correspond to `<variable>` elements, while those of the form $x : \ell$ to `<partnerLink>` elements.

As specified in [61, Sect. 3], operation overloading is not permitted, e.g. operation types such as $o\langle\text{INT}\rangle$ and $o\langle\text{INT}, \text{BOOL}\rangle$ cannot occur in the same local typing. Thus, the notion of network well-formedness extends to typed networks by taking into account also this syntactic constraint. Anyway, type declarations within nodes do not play any role in the operational semantics of WS-CALCULUS and the operational rules in Tables 4 and 7 still hold for typed networks. In other words, type declarations do not affect the transitions and are not modified by them. Therefore, types within nodes have been omitted in Sect. 3; on the contrary, the results presented in this section will be based on transitions between typed terms.

We conclude with a few simple examples aimed at clarifying the intuitive meaning of the type declarations.

In the simplest interaction pattern, where a requestor B invokes a service provider A and the interaction immediately terminates, a single one-way operation o_{req} suffices:

$$\begin{aligned} & A ::^{\omega_A} \{ [\, B/x_B] > \mathbf{rec}(x_B : o_{req} : x) \} \\ \parallel\ & B ::^{\omega_B} \{ [\, A/x_A, 2/y] \gg \mathbf{inv}(x_A : o_{req} : y) \} \end{aligned}$$

where $\omega_A = \{ x_B : \langle [o_{req} \langle \text{INT} \rangle], [/] \rangle , x : \text{INT} \}$ and $\omega_B = \{ x_A : \langle [/], [o_{req} \langle \text{INT} \rangle] \rangle , y : \text{INT} \}$. The service A, which provides the operation o_{req}, specifies the partner link x_B whose type contains a single role played by A itself. Symmetrically, B specifies the partner link x_A and does not play any role in the collaboration, because the invoked operation o_{req} is provided by the partner.

The more complex asynchronous request-response interaction pattern is expressed by connecting two one-way operations (request and callback):

$$\begin{aligned} & A ::^{\omega_A} \{ [B/x_B] > \mathbf{rec}(x_B : o_{req} : x) ; \mathbf{inv}(x_B : o_{res} : x) \} \\ & \parallel B ::^{\omega_B} \{ [A/x_A, 2/y] \gg \mathbf{inv}(x_A : o_{req} : y) ; \mathbf{rec}(x_A : o_{res} : y) \} \end{aligned}$$

where ω_A and ω_B are respectively defined as $\{ x_B : \langle [o_{req} \langle \text{INT} \rangle], [o_{res} \langle \text{INT} \rangle] \rangle , x : \text{INT} \}$ and $\{ x_A : \langle [o_{res} \langle \text{INT} \rangle], [o_{req} \langle \text{INT} \rangle] \rangle , y : \text{INT} \}$. Now, both services play a role in the collaboration, according to the provided operation. Notice that the provider service A (statically) knows the address of the requestor (indeed, its initial state is $[B/x_B]$). The example can be slightly modified to allow the provider to discover at run-time the address of the requestor and, hence, to dynamically bind it to the partner link x_B:

$$\begin{aligned} & A ::^{\omega_A} \{ [\ ] > \mathbf{rec}(x_B : o_{req} : x_B, x) ; \mathbf{inv}(x_B : o_{res} : x) \} \\ & \parallel B ::^{\omega_B} \{ [A/x_A, B/y_B, 2/y] \gg \mathbf{inv}(x_A : o_{req} : y_B, y) ; \mathbf{rec}(x_A : o_{res} : y) \} \end{aligned}$$

where, this time, ω_A and ω_B are defined as $\{ x_B : \langle [o_{req} \langle \{ \pi \}, \text{INT} \rangle], \pi \rangle , x : \text{INT} \}$ and $\{ x_A : \langle \pi, [o_{req} \langle \{ \pi \}, \text{INT} \rangle] \rangle , y_B : \{ \pi \} , y : \text{INT} \}$, respectively, where π stands for $[o_{res} \langle \text{INT} \rangle]$. Notably, due to address-passing, the first argument of the operation o_{req}, as well as the variable y_B, has type $\{ \pi \}$, which is an address type.

4.2 Type checking

Type environments, ranged over by θ, map addresses to address types. This information is exploited to properly deal with address-passing, because invoke and receive activities can use partner links as message variables for exchanging node addresses. For example, the type checking will control that a variable of type $\langle \pi_1, \pi_2 \rangle$ must be assigned an address that, according to the type environment, provides the port type π_2. Type environments also store in a compact form the type of the interaction patterns that all processes in the network should follow along their computations.

The judgement $\theta \vdash N$, defined by the inference rules in Table 10, states that a network N is well-typed under the type environment θ. Rule (T-NET) says that a network is well-typed under a type environment, if each node of the network is well-typed under the same environment. Rule (T-NODE) says that a node is well-typed if its components are well-typed too. The premise $\theta(\lambda) = \mathit{myRoles}(\omega)$, where we let $\mathit{myRoles}(\omega)$ denote the set $\{ \pi \mid x : \langle \pi, \pi' \rangle \in \omega \}$, checks if the type environment θ contains the type information concerning the components located at λ. In fact, for a given node $\lambda ::^{\omega} \{ c \}$, $\mathit{myRoles}(\omega)$ includes the port types provided by the components located at λ (to be used in order to collaborate with them). Therefore, (T-NET) and (T-NODE) together require the type environment θ used to typecheck a network to contain type associations for all nodes of the network. Actually, such associations could be easily extracted from the local types of the considered nodes; e.g. given the network $\lambda_1 ::^{\omega_1} \{ c_1 \} \parallel \cdots \parallel \lambda_n ::^{\omega_n} \{ c_n \}$, a suitable type environment is $[\mathit{myRoles}(\omega_1)/\lambda_1, \dots, \mathit{myRoles}(\omega_n)/\lambda_n]$.

The judgement $\omega \vdash_\theta c$, defined by the inference rules in Table 11, states that a collection of components is well-typed w.r.t. a type environment θ and a local type ω. Rule (T-COMP)

Table 10 Inference rules for $\theta \vdash N$

$\dfrac{\theta \vdash N_1 \quad \theta \vdash N_2}{\theta \vdash N_1 \parallel N_2}$ (T-NET)	$\dfrac{\omega \vdash_\theta c \quad \theta(\lambda) = \mathit{myRoles}(\omega)}{\theta \vdash \lambda ::^{\omega} \{c\}}$ (T-NODE)

Table 11 Inference rules for $\omega \vdash_\theta c$

$\dfrac{\omega \vdash_\theta c_1 \quad \omega \vdash_\theta c_2}{\omega \vdash_\theta c_1, c_2}$ (T-COMP)	$\dfrac{o\langle\bar{\upsilon}\rangle \Subset \mathit{myRoles}(\omega) \quad \omega \vdash_\theta \bar{u} : \bar{\upsilon}' \quad \bar{\upsilon} \sqsubseteq \bar{\upsilon}'}{\omega \vdash_\theta \langle o : \bar{u}\rangle}$ (T-REQ)	
$\dfrac{\omega \vdash_\theta m \gg a}{\omega \vdash_\theta m > a}$ (T-DEF)	$\dfrac{\omega \vdash_\theta a}{\omega \vdash_\theta [\ ] \gg a}$ (T-INST1)	$\dfrac{\omega \vdash_\theta m \gg \mathbf{asg}(\bar{x} : \bar{u})\, ; a}{\omega \vdash_\theta m \circ [\bar{u}/\bar{x}] \gg a}$ (T-INST2)

Table 12 Inference rules for $\omega \vdash_\theta \bar{e} : \bar{\upsilon}$ and $\omega \vdash_\theta \bar{w} : \bar{\upsilon}$

$\dfrac{u \in \mathtt{Int}}{\omega \vdash_\theta u : \mathrm{INT}}$ (T-INT)	$\dfrac{u \in \mathtt{Str}}{\omega \vdash_\theta u : \mathrm{STR}}$ (T-STR)	$\dfrac{u \in \{\mathbf{true}, \mathbf{false}\}}{\omega \vdash_\theta u : \mathrm{BOOL}}$ (T-BOOL)
$\dfrac{x : \upsilon \in \omega}{\omega \vdash_\theta x : \upsilon}$ (T-VAR)	$\dfrac{x : \upsilon \in \omega}{\omega \vdash_\theta !x : \upsilon}$ (T-JOIN)	$\dfrac{x : \ell \in \omega}{\omega \vdash_\theta x : \ell}$ (T-LINK)
$\dfrac{\theta(u) = \{\pi_i\}_{i \in I}}{\omega \vdash_\theta u : \{\pi_i\}_{i \in I}}$ (T-ADDR)	$\dfrac{\omega \vdash_\theta e_1 : \upsilon_1 \quad \cdots \quad \omega \vdash_\theta e_n : \upsilon_n}{\omega \vdash_\theta e_1, \ldots, e_n : \upsilon_1, \ldots, \upsilon_n}$ (T-EXPR)	
$\dfrac{\omega \vdash_\theta x : \langle\pi_1, \pi_2\rangle}{\omega \vdash_\theta x : \{\pi_2\}}$ (T-ROLE)	$\dfrac{\omega \vdash_\theta w_1 : \upsilon_1 \quad \cdots \quad \omega \vdash_\theta w_n : \upsilon_n}{\omega \vdash_\theta w_1, \ldots, w_n : \upsilon_1, \ldots, \upsilon_n}$ (T-TUPLE)	

is used to split the judgement for a collection into the judgements for each single component. Rule (T-REQ) states that a message request delivered to a node is well-typed if the node provides the requested operation with the proper types. Notation $o\langle\bar{\upsilon}\rangle \Subset \mathit{myRoles}(\omega)$ is a short-hand to mean that there exists some port type $\pi \in \mathit{myRoles}(\omega)$ such that $o\langle\bar{\upsilon}\rangle \in \pi$. Condition $\bar{\upsilon} \sqsubseteq \bar{\upsilon}'$ ensures that the type $\bar{\upsilon}$ of the arguments of the invoked operation o conforms to the type $\bar{\upsilon}'$ of the corresponding transmitted values. The symbol $\sqsubseteq$ denotes the *subtyping preorder* over value types induced by letting $\beta \sqsubseteq \beta'$ if $\beta = \beta'$ and $\gamma \sqsubseteq \gamma'$ if $\gamma \subseteq \gamma'$. The preorder extends component-wise to tuples of value types, i.e. $\bar{\upsilon} \sqsubseteq \bar{\upsilon}'$ if $\bar{\upsilon}$ and $\bar{\upsilon}'$ have the same number of fields and the corresponding fields are in the subtyping relation. Thus, if the type of the invoked operation contains an address type $\{\pi_i\}_{i \in I}$ then, as a set, it must be a subset of the address type $\{\pi'_j\}_{j \in J}$ of the corresponding value contained in the message $\bar{u}$. The intuition here is that it is safe to send along o addresses that provide more operations than those a receiver might initially expect. By means of rule (T-DEF), process definitions are typed in terms of the corresponding process instance. Finally, rule (T-INST1) establishes that a process instance $[\ ] \gg a$ with empty state is well-typed if the activity a is well-typed w.r.t. the same type environment θ and local type ω, while rule (T-INST2) establishes that a process instance $m \circ [\bar{u}/\bar{x}] \gg a$ is well-typed if the activity resulting from the sequential composition of the assign activity producing the state $[\bar{u}/\bar{x}]$ with the activity a is well-typed in the state m w.r.t. the same type environment θ and local type ω.

The inference rules for the judgements $\omega \vdash_\theta \bar{e} : \bar{\upsilon}$ and $\omega \vdash_\theta \bar{w} : \bar{\upsilon}$ are in Table 12. Since the syntax of expressions has not been specified, our type system only includes inference rules for basic values and variables. Once the syntax of expressions will have been precisely established, specific inference rules should be added for each operator. For example, if integer addition would be added to the syntax of expressions, then the following rule should be

Table 13 Inference rules for $\omega \vdash_\theta a$

$$\frac{\omega \vdash_\theta o\langle\bar{v}\rangle \in \mathit{partnerRole}(x) \quad \omega \vdash_\theta \bar{y} : \bar{v}' \quad \bar{v} \sqsubseteq \bar{v}'}{\omega \vdash_\theta \mathbf{inv}(x : o : \bar{y})} \; (\text{T-INV})$$

$$\frac{\omega \vdash_\theta \bar{x} : \bar{v} \quad \omega \vdash_\theta \bar{e} : \bar{v}' \quad \bar{v} \sqsubseteq \bar{v}'}{\omega \vdash_\theta \mathbf{asg}(\bar{x} : \bar{e})} \; (\text{T-ASG}) \qquad \frac{\omega \vdash_\theta a_1 \quad \omega \vdash_\theta a_2}{\omega \vdash_\theta a_1 \,;\, a_2} \; (\text{T-SEQ})$$

$$\frac{\omega \vdash_\theta a_1 \quad \omega \vdash_\theta a_2}{\omega \vdash_\theta a_1 \mid a_2} \; (\text{T-FLOW}) \qquad \frac{\forall i \in I \quad \omega \vdash_\theta \mathbf{rec}(x_i : o_i : \bar{w}_i) \,;\, a_i}{\omega \vdash_\theta \sum_{i \in I} \mathbf{rec}(x_i : o_i : \bar{w}_i) \,;\, a_i} \; (\text{T-PICK})$$

$$\frac{\omega \vdash_\theta e : \text{BOOL} \quad \omega \vdash_\theta a_1 \quad \omega \vdash_\theta a_2}{\omega \vdash_\theta \mathbf{if}\, e\, \mathbf{then}\, a_1\, \mathbf{else}\, a_2} \; (\text{T-IF}) \qquad \frac{\omega \vdash_\theta e : \text{BOOL} \quad \omega \vdash_\theta a}{\omega \vdash_\theta \mathbf{while}\, e\, \mathbf{do}\, a} \; (\text{T-WH})$$

$$\frac{\omega \vdash_\theta o\langle\bar{v}\rangle \in \mathit{myRole}(x) \quad \omega \vdash_\theta \bar{w} : \bar{v}' \quad \bar{v}' \sqsubseteq \bar{v}}{\omega \vdash_\theta \mathbf{rec}(x : o : \bar{w})} \; (\text{T-REC})$$

added to the type system:

$$\frac{\omega \vdash_\theta e_1 : \text{INT} \quad \omega \vdash_\theta e_2 : \text{INT}}{\omega \vdash_\theta e_1 + e_2 : \text{INT}} \; (\text{T-OP}_+)$$

We comment now on the most significant rules. Rules (T-INT), (T-STR) and (T-BOOL) simply check if a value belongs to the set of values corresponding to its type (`Int` and `Str` denote the set of integer and string values, respectively). Address values are typed by exploiting the type environment θ (rule (T-ADDR)), while variables (i.e. basic variables, join-variables and partner links) are typed by exploiting the local type ω (rules (T-VAR), (T-JOIN), (T-LINK) and (T-ROLE)). In particular, rule (T-ROLE) states that a partner link of type $\langle\pi_1, \pi_2\rangle$ can be used as an address variable of type $\{\pi_2\}$. This rule permits checking assignments of type-compatible values involving partner links variables, e.g. addresses can be assigned to such variables only if the corresponding nodes provide the port type π_2, at least.

The judgement $\omega \vdash_\theta a$, defined by the inference rules in Table 13, states that activity a is well-typed w.r.t. a type environment θ and a local type ω. Notation $\omega \vdash_\theta o\langle\bar{v}\rangle \in \mathit{partnerRole}(x)$ stands for the judgement $\omega \vdash_\theta x : \langle\pi_1, \pi_2 \cup [o\langle\bar{v}\rangle]\rangle$, while notation $\omega \vdash_\theta o\langle\bar{v}\rangle \in \mathit{myRole}(x)$ stands for the judgement $\omega \vdash_\theta x : \langle\pi_1 \cup [o\langle\bar{v}\rangle], \pi_2\rangle$, where in both cases π_1 and π_2 are generic port types. Rule (T-INV) states that an invoke activity is well-typed when it is performed in collaboration with a partner providing the type definition of the invoked operation. The premises ensure that the message type $\bar{v}$ of the invoked operation conforms to the type $\bar{v}'$ of the variables argument of the invoke activity. Similarly to rule (T-REQ) in Table 11, the underlying intuition is that e.g. it is safe to send addresses providing more operations than those a receiver might expect. Rule (T-ASG) checks type-compatible assignments. Also in this case the subtyping judgement between arguments is required to hold. Rule (T-PICK) exploits rules (T-SEQ) and (T-REC) to check well/typedness of the component activities. As a special case, when $I = \emptyset$, we get the rule

$$\omega \vdash_\theta \mathbf{empty} \quad (\text{T-EMPTY})$$

Differently from rule (T-INV), in rule (T-REC) the operation type is provided by the node performing the receive operation. In addition, by the subtyping relation, it is considered safe to receive addresses that provide more operations than those the receiver might actually use.

One of the major contributions of our type system is to shed light on the most peculiar and intricate aspects of the relationship between WSDL and WS-BPEL, such as the handling of partner links. As we have already pointed out, our typing discipline requires partner links to be typed by pairs of port types characterizing the two roles involved in a collaboration. The type checking rules exploit the first element of such pairs when checking receive activities and the second element when checking invoke activities. No rule uses the two elements at the same time. This is also the case when dealing with address-passing, which requires to transform (by applying rule (T-ROLE)) partner link types $\langle \pi_1, \pi_2 \rangle$ into address types $\{\pi_2\}$, since the types of message variables/data do not include partner link types.

4.3 Type soundness

The main property of our type system is that if a network is well-typed then it never reaches an error configuration (Corollary 1). The proof proceeds in the style of [68] by first proving *subject reduction*, namely that networks well-typedness is an invariant of the reduction relation (Theorem 1), and then proving *type safety*, namely that well-typed networks are error-free (Theorem 2). The errors that our typing discipline permits to prevent are characterised by the predicate $\Uparrow_\lambda^\theta$ defined by the inference rules in Table 14.

The subject reduction theorem exploits an auxiliary lemma stating that if a process instance is well-typed then its continuation after a labelled transition is well-typed too.

Lemma 1 *If $\omega \vdash_\theta m \gg a$ and $m \gg a \xrightarrow{\alpha} m' \gg a'$, then $\omega \vdash_\theta m' \gg a'$.*

Proof The proof is by induction on the depth of the inference of the labelled transition $m \gg a \xrightarrow{\alpha} m' \gg a'$ and case analysis on the last applied rule of Table 4. For the base step, we only show two significant cases.

In the case of rule (ASSIGN), we have $a = \mathbf{asg}(\bar{x} : \bar{e})$, $m = [\bar{u}/\bar{y}]$ (for some $\bar{y}$ and $\bar{u}$), $m' = m \circ [m(\bar{e})/\bar{x}]$ and $a' = \mathbf{empty}$. From the hypothesis $\omega \vdash_\theta m \gg \mathbf{asg}(\bar{x} : \bar{e})$, by rule (T-INST2), we get $\omega \vdash_\theta [\ ] \gg \mathbf{asg}(\bar{y} : \bar{u})\,;\mathbf{asg}(\bar{x} : \bar{e})$, and, by rule (T-INST1), we get $\omega \vdash_\theta \mathbf{asg}(\bar{y} : \bar{u})\,;\mathbf{asg}(\bar{x} : \bar{e})$. Then, by rule (T-SEQ), we have $\omega \vdash_\theta \mathbf{asg}(\bar{y} : \bar{u})$ and $\omega \vdash_\theta \mathbf{asg}(\bar{x} : \bar{e})$. From the latter, by rule (T-ASG), we get $\omega \vdash_\theta \bar{x} : \bar{v}$, $\omega \vdash_\theta \bar{e} : \bar{v}'$ and $\bar{v} \sqsubseteq \bar{v}'$. By assuming that the evaluation function $m(\cdot)$ is type-preserving,[6] we obtain $\omega \vdash_\theta m(\bar{e}) : \bar{v}'$. Hence, again by using rule (T-ASG), we get $\omega \vdash_\theta \mathbf{asg}(\bar{x} : m(\bar{e}))$. By rules (T-EMPTY) and (T-SEQ) (two times), we get $\omega \vdash_\theta \mathbf{asg}(\bar{y} : \bar{u})\,;\mathbf{asg}(\bar{x} : m(\bar{e}))\,;\mathbf{empty}$. Now, by rule (T-INST1), $\omega \vdash_\theta [\ ] \gg \mathbf{asg}(\bar{y} : \bar{u})\,;\mathbf{asg}(\bar{x} : m(\bar{e}))\,;\mathbf{empty}$, and by (T-INST2), $\omega \vdash_\theta m \gg \mathbf{asg}(\bar{x} : m(\bar{e}))\,;\mathbf{empty}$. Now, again by applying rule (T-INST2), we get $\omega \vdash_\theta m \circ [m(\bar{e})/\bar{x}] \gg \mathbf{empty}$, as to be proved.

In the case of rule (PICK), we have $a = \sum_{i \in I} \mathbf{rec}(x_i : o_i : \bar{w}_i)\,; a_i$, $m = [\bar{u}/\bar{y}]$ (for some $\bar{y}$ and $\bar{u}$), $m' = m$ and $a' = a_k$ for some $k \in I$. By hypothesis and rules (T-INST2) and (T--INST1), we have $\omega \vdash_\theta \mathbf{asg}(\bar{y} : \bar{u})\,; a$. By rule (T-SEQ), we get $\omega \vdash_\theta \mathbf{asg}(\bar{y} : \bar{u})$ and $\omega \vdash_\theta a$. By rules (T-PICK) and (T-SEQ), we obtain $\omega \vdash_\theta a_k$ for each $k \in I$. By applying rule (T-SEQ) again, we have $\omega \vdash_\theta \mathbf{asg}(\bar{y} : \bar{u})\,; a_k$ for each $k \in I$. Finally, by applying rules (T-INST1) and (T-INST2), we obtain $\omega \vdash_\theta m \gg a_k$, as to be proved.

For the induction step, we only show the case of rule (SEQ$_1$), since that of rule (FLOW$_1$) proceeds similarly. Then, suppose $a = a_1\,; a_2$, $m = [\bar{u}/\bar{x}]$ (for some $\bar{x}$ and $\bar{u}$), $m \gg a_1 \xrightarrow{\alpha} m' \gg a_1'$ and $a' = a_1'\,; a_2$. From the hypothesis $\omega \vdash_\theta m \gg a_1\,; a_2$, by rule (T-INST2), we have $\omega \vdash_\theta [\ ] \gg \mathbf{asg}(\bar{x} : \bar{u})\,; a_1\,; a_2$, and, by rule (T-INST1), we have $\omega \vdash_\theta \mathbf{asg}(\bar{x} : \bar{u})\,; a_1\,; a_2$.

[6] Since the syntax of expressions and, hence, the evaluation function are not specified (see Sect. 3), the type-preservation property of $m(\cdot)$ cannot be proved. It is however perfectly reasonable to require it as a condition to be satisfied by $m(\cdot)$.

By rule (T-SEQ), we get $\omega \vdash_\theta \textbf{asg}(\bar{x} : \bar{u})\,;a_1$ and $\omega \vdash_\theta a_2$. Now, by rules (T-INST1) and (T-INST2), we have $\omega \vdash_\theta m \gg a_1$. Thus, by induction hypothesis, we get $\omega \vdash_\theta m' \gg a'_1$. The thesis then follows by applying rules (T-INST2), (T-INST1) and (T-SEQ), first to infer that $\omega \vdash_\theta \textbf{asg}(\bar{y} : \bar{u}')\,;a'_1\,;a_2$ (for some proper $\bar{y}$ and $\bar{u}'$ such that $m' = [\bar{u}'/\bar{y}]$), then to infer that $\omega \vdash_\theta m' \gg a'_1\,;a_2$, as to be proved. □

To prove subject reduction, we also need an auxiliary lemma stating that in a well-typed network the port type of the partner role of a partner link must be defined in the local type of the node to which the partner link refers to.

Lemma 2 *If* $\theta \vdash \lambda_1 ::^{\omega_1} \{m \gg a,\ c_1\} \parallel \lambda_2 ::^{\omega_2} \{c_2\}$ *and* $m(x) = \lambda_2$, *then* $\omega_1 \vdash_\theta x : \langle \pi', \pi \rangle$ *and* $\pi \in \mathit{myRoles}(\omega_2)$, *for some port types* π' *and* π.

Proof From the first hypothesis, by rule (T-NET), it follows that $\theta \vdash \lambda_1 ::^{\omega_1} \{m \gg a,\ c_1\}$. By rules (T-NODE), (T-COMP), (T-INST2) and (T-INST1), we get $\omega_1 \vdash_\theta \textbf{asg}(\bar{x} : \bar{u})\,;a$, where $\bar{x}$ and $\bar{u}$ are such that $m = [\bar{u}/\bar{x}]$. Now, by rule (T-SEQ), we have that $\omega_1 \vdash_\theta \textbf{asg}(\bar{x} : \bar{u})$. Since by hypothesis $m(x) = \lambda_2$, when applying rule (T-ASG) it is checked in particular that, for some port type π, $\omega_1 \vdash_\theta x : \{\pi\}$, $\omega_1 \vdash_\theta \lambda_2 : \{\pi_i\}_{i \in I}$, and $\{\pi\} \sqsubseteq \{\pi_i\}_{i \in I}$. Now, the first check, by rule (T-ROLE), implies $\omega_1 \vdash_\theta x : \langle \pi', \pi \rangle$, for some port type π'; the second check, by rule (T-ADDR), implies $\theta(\lambda_2) = \{\pi_i\}_{i \in I}$; and the last check implies $\{\pi\} \subseteq \theta(\lambda_2)$. From the first hypothesis, by rule (T-NET), it follows that $\theta \vdash \lambda_2 ::^{\omega_2} \{c_2\}$. Hence, by rule (T-NODE), we get $\theta(\lambda_2) = \mathit{myRoles}(\omega_2)$, which implies $\pi \in \mathit{myRoles}(\omega_2)$, as to be proved. □

We can now prove the *subject reduction* theorem stating preservation of well-typedness under networks evolution. As a matter of notation, we write $N \equiv \lambda_1 ::^{\omega_1} \{c_1\} \parallel \cdots \parallel \lambda_k ::^{\omega_k} \{c_k\}$ to mean that N *is of the form* $\lambda_1 ::^{\omega_1} \{c_1\} \parallel \cdots \parallel \lambda_k ::^{\omega_k} \{c_k\}$.

Theorem 1 (Subject reduction) *If* $\theta \vdash N$ *and* $N \rightarrowtail N'$, *then* $\theta \vdash N'$.

Proof The proof is by induction on the depth of the inference of the reduction $N \rightarrowtail N'$ and case analysis on the last applied rule of Table 7.
For the base step, we have the following cases to consider:

(N-INV): In this case we have $N \equiv \lambda_1 ::^{\omega_1} \{m \gg a,\ c_1\} \parallel \lambda_2 ::^{\omega_2} \{c_2\}$ and $N' \equiv \lambda_1 ::^{\omega_1} \{m \gg a',\ c_1\} \parallel \lambda_2 ::^{\omega_2} \{\langle o : \bar{u} \rangle,\ c_2\}$. Moreover, the reduction $N \rightarrowtail N'$ is caused by an activity-level transition $m \gg a \xrightarrow{\lambda_2\,\langle o : \bar{u} \rangle} m \gg a'$ pointing out that a is performing an invoke activity $\textbf{inv}(x : o : \bar{y})$ with $\lambda_2 = m(x)$ and $\bar{u} = m(\bar{y})$. Now, by the well-typedness hypothesis, we have $\omega_1 \vdash_\theta m \gg a$. On the one hand, by Lemma 1, this implies that $\omega_1 \vdash_\theta m \gg a'$. On the other hand, by first applying rules (T-INST2) and (T-INST1), and then, possibly, other rules in Table 13 according to the syntax of a, we eventually end up to apply rule (T-INV), thus we get that its premises do hold, i.e., for some port types π_1 and π_2, $\omega_1 \vdash_\theta x : \langle \pi_1, \pi_2 \cup [o\langle \bar{v} \rangle] \rangle$, $\omega_1 \vdash_\theta \bar{y} : \bar{v}'$ and $\bar{v} \sqsubseteq \bar{v}'$. Now, by assuming that the evaluation function $m(\cdot)$ is type-preserving, we get $\omega_1 \vdash_\theta m(\bar{y}) : \bar{v}'$. Thus, since the rules in the first line of Table 12 are independent of the local type occurring in the judgement, we get $\omega_2 \vdash_\theta m(\bar{y}) : \bar{v}'$. Since $m(x) = \lambda_2$, by Lemma 2, we get $\pi_2 \cup [o\langle \bar{v} \rangle] \in \mathit{myRoles}(\omega_2)$, i.e. $o\langle \bar{v} \rangle \Subset \mathit{myRoles}(\omega_2)$. Hence, all premises of rule (T-REQ) do hold thus, by applying the rule, we get $\omega_2 \vdash_\theta \langle o : \bar{u} \rangle$. Finally, because of the syntactic form of N', the thesis follows by applying rules (T-COMP), (T-NODE) and (T-NET).

Table 14 Error configurations $N \Uparrow^{\theta}_{\lambda}$ (with $\rhd \in \{\gg, >\}$ and symmetric of rule (E-PAR) omitted)

$$\frac{x : \langle \pi_1, \pi_2 \rangle \in \omega_1 \qquad \pi_2 \notin \theta(\lambda_2)}{(\lambda_1 ::^{\omega_1} \{ m \circ [\lambda_2/x] \rhd a,\ c_1 \}) \Uparrow^{\theta}_{\lambda_1}} \ \text{(E-PARTNERROLE)}$$

$$\frac{m \gg a \xrightarrow{(o\,:\,\bar{w})} m \gg a' \qquad o\langle \bar{v} \rangle \not\Subset \theta(\lambda)}{(\lambda ::^{\omega} \{ m \rhd a,\ c \}) \Uparrow^{\theta}_{\lambda}} \ \text{(E-MYROLE)}$$

$$\frac{m \gg a \xrightarrow{\lambda_2\,\langle o\,:\,\bar{u} \rangle} m \gg a' \qquad o\langle \bar{v} \rangle \Subset \theta(\lambda_2) \qquad \bar{v} \not\sqsubseteq \bar{v}' \text{ with } \omega_1 \vdash_{\theta} \bar{u} : \bar{v}'}{(\lambda_1 ::^{\omega_1} \{ m \gg a,\ c_1 \}) \Uparrow^{\theta}_{\lambda_1}} \ \text{(E-MSG}_1\text{)}$$

$$\frac{m \gg a \xrightarrow{(o\,:\,\bar{w})} m \gg a' \qquad o\langle \bar{v} \rangle \Subset \theta(\lambda) \qquad \bar{v}' \not\sqsubseteq \bar{v} \text{ with } \omega \vdash_{\theta} \bar{w} : \bar{v}'}{(\lambda ::^{\omega} \{ m \rhd a,\ c \}) \Uparrow^{\theta}_{\lambda}} \ \text{(E-MSG}_2\text{)}$$

$$\frac{v \not\sqsubseteq v' \text{ with } \omega \vdash_{\theta} x : v \text{ and } \omega \vdash_{\theta} u : v'}{(\lambda ::^{\omega} \{ m \circ [u/x] \rhd a,\ c \}) \Uparrow^{\theta}_{\lambda}} \ \text{(E-ASG)} \qquad \frac{N_1 \Uparrow^{\theta}_{\lambda}}{N_1 \parallel N_2 \Uparrow^{\theta}_{\lambda}} \ \text{(E-PAR)}$$

(N-REC) and (N-START): We proceed in a way similar to (N-INV). We only show the case of rule (N-REC), since that of rule (N-START) is analogous. Hence, we have $N \equiv \lambda ::^{\omega} \{ m \gg a,\ \langle o : \bar{u} \rangle,\ c \}$ and $N' \equiv \lambda ::^{\omega} \{ m \circ m' \gg a',\ c \}$. Moreover, the reduction $N \rightarrowtail N'$ is caused by an activity-level transition $m \gg a \xrightarrow{(o\,:\,\bar{w})} m \gg a'$ pointing out that a is performing a receive activity $\mathbf{rec}(x : o : \bar{w})$. Now, by hypothesis and rules (T-NODE) and (T-COMP), we get $\theta(\lambda) = \mathit{myRoles}(\omega)$, $\omega \vdash_{\theta} c$, $\omega \vdash_{\theta} m \gg a$ and $\omega \vdash_{\theta} \langle o : \bar{u} \rangle$. From $\omega \vdash_{\theta} m \gg a$, on the one hand, by Lemma 1, we get that $\omega_1 \vdash_{\theta} m \gg a'$. On the other hand, by first applying rules (T-INST2) and (T-INST1), and then, possibly, other rules in Table 13 according to the syntax of a, we eventually end up to apply rule (T-REC), thus we get $\omega \vdash_{\theta} x : \langle \pi_1 \cup [o\langle \bar{v} \rangle], \pi_2 \rangle$, $\omega \vdash_{\theta} \bar{w} : \bar{v}'$ and $\bar{v}' \sqsubseteq \bar{v}$. From $\omega \vdash_{\theta} \langle o : \bar{u} \rangle$, by rule (T-REQ), we get $o\langle \bar{v} \rangle \Subset \mathit{myRoles}(\omega)$, that is $\pi \cup [o\langle \bar{v} \rangle] \in \mathit{myRoles}(\omega)$ for some port type π, $\omega \vdash_{\theta} \bar{u} : \bar{v}''$ and $\bar{v} \sqsubseteq \bar{v}''$. Now, from $\omega \vdash_{\theta} \bar{w} : \bar{v}'$, by applying rules (T-JOIN) and (T-VAR), we get $\omega \vdash_{\theta} \bar{y} : \bar{v}'$, where $\bar{y}$ is obtained from $\bar{w}$ by removing possible occurrences of operator "!". From this, $\omega \vdash_{\theta} \bar{u} : \bar{v}''$ and $\bar{v}' \sqsubseteq \bar{v}''$ (that follows because of transitivity of the preorder $\sqsubseteq$), by applying rule (T-ASG), we get $\omega \vdash_{\theta} \mathbf{asg}(\bar{y} : \bar{u})$. By rule (T-SEQ), we get $\omega \vdash_{\theta} \mathbf{asg}(\bar{y} : \bar{u})\,;a'$ and, again by rule (T-SEQ), we get $\omega \vdash_{\theta} \mathbf{asg}(\bar{x} : \bar{u}')\,; \mathbf{asg}(\bar{y} : \bar{u})\,; a'$, for some $\bar{x}$ and $\bar{u}'$ such that $m = [\bar{u}'/\bar{x}]$. Now, by rule (T--INST1), we get $\omega \vdash_{\theta} [\ ] \gg \mathbf{asg}(\bar{x} : \bar{u}')\,; \mathbf{asg}(\bar{y} : \bar{u})\,; a'$, and by rule (T-INST2), we get $\omega \vdash_{\theta} m \gg \mathbf{asg}(\bar{y} : \bar{u})\,; a'$. By rule (T-INST2) again, we get $\omega \vdash_{\theta} m \circ m' \gg a'$ for $m' = [\bar{u}/\bar{y}]$. Finally, because of the syntactic form of N', the thesis follows by applying rules (T-COMP), (T-NODE) and (T-NET).

(N-TAU): This case easily follows from Lemma 1.

For the induction step, we only have to consider the case of rule (N-PAR). Then, the thesis easily follows by induction hypothesis and rule (T-NET). □

The errors that our type system can prevent are characterised by the predicate $\Uparrow^{\theta}_{\lambda}$. The predicate holds true on *error configurations*, meaning intuitively that the behaviour of a component running at λ can violate a type constraint stored in θ. To capture the presence of errors in networks, we exploit the type environment θ since, as we have pointed out at the beginning of Sect. 4.2, it contains in a compact form the type specifications of all networks nodes, as e.g. the interaction patterns that all processes in the network should follow along their computations. Thus, θ is at the same time a repository for the patterns establishing how interactions should take place and a technical device to implement the check that interactions really proceed as prescribed.

The rules defining $\Uparrow_\lambda^\theta$ are shown in Table 14, where $\not\Subset$ means that $\Subset$ does not hold, $\not\sqsubseteq$ means that the subtype preorder $\sqsubseteq$ does not hold, and $m \rhd a$ stands both for process definitions ($m > a$) and for process instances ($m \gg a$). Rule (E-PARTNERROLE) states that there is an error when the partner role π_2 of a partner link x is not provided as a port type in the local type of the node to which the partner link refers to. For example, consider the typed version of the network presented at the beginning of this section

$$\begin{aligned} & A ::^{\omega_A} \{ [\text{“}ok\text{”}/y] \gg \mathbf{rec}(x_B : o_{req} : x_B) \,;\, \mathbf{inv}(x_B : o_{res} : y) \} \\ \| \; & B ::^{\omega_B} \{ [A/x_A, B/z] \gg \mathbf{inv}(x_A : o_{req} : z) \} \end{aligned}$$

where in the local type ω_A the partner link type of x_B is $\langle [o_{req}\langle\{\pi\}\rangle],\pi\rangle$, with $\pi = [o_{res}\langle \text{STR}\rangle]$, while the local type ω_B is empty. The type checking reveals an error in the collaboration between the two instances since x_B is not compatible with B, in fact $\pi \notin myRoles(\omega_B)$. Rule (E-MYROLE) states that there is an error when the type declaration of the requested operation is not found in the type of the local node. Indeed, the component $m \rhd a$ must be the provider of the operation, since by hypothesis it performs a receive activity along such operation. Rules (E-MSG$_1$) and (E-MSG$_2$) point out errors due to the fact that the wanted operation is provided but its type is not compatible with the type of the message variables. Rule (E-ASG) indicates errors that occur when some stored values are incompatible with the type of the corresponding variables. Finally, rule (E-PAR) is a standard contextual rule.

We now prove the *type safety* theorem stating that well-typed networks are error-free.

Theorem 2 (Type safety) *If $\theta \vdash N$ then $N \Uparrow_\lambda^\theta$ does not hold, for every address λ of N.*

Proof From the hypothesis, it follows that for all nodes $\lambda ::^\omega \{c\}$ of N we have that $\theta(\lambda) = myRoles(\omega)$. We proceed by contradiction and show that if $N \Uparrow_\lambda^\theta$ then $\theta \vdash N$ could not be derived. We now reason by induction on the inference of predicate $N \Uparrow_\lambda^\theta$ using the rules in Table 14. For the base step, we have the following cases to consider:

(E-PARTNERROLE) Thus $N \equiv \lambda_1 ::^{\omega_1} \{m \circ [\lambda_2/x] \rhd a,\ c_1\}$. This means that the process located at λ_1 uses the partner link x to refer to node λ_2 but the port type π_2 is not defined in the local type ω_2 of λ_2, hence $\pi_2 \notin myRoles(\omega_2)$. Now, suppose that $\theta \vdash N$ can be derived. Then, by rules (T-INST2) and (T-INST1), and, possibly, (T-DEF), in case of process definition, we can derive $\omega_1 \vdash_\theta \mathbf{asg}(\bar{y}, x : \bar{u}, \lambda_2) \,;\, a$, for some $\bar{y}$ and $\bar{u}$ such that $m = [\bar{u}/\bar{y}]$. By rule (T-SEQ) in Table 13, we get $\omega_1 \vdash_\theta \mathbf{asg}(\bar{y}, x : \bar{u}, \lambda_2)$, which, by rule (T-ASG), would imply in particular that $\omega_1 \vdash_\theta x : \{\pi_2\}$ and $\{\pi_2\} \sqsubseteq \theta(\lambda_2)$. However, this cannot be inferred because we already know that $\theta(\lambda_2) = myRoles(\omega_2)$ and $\pi_2 \notin myRoles(\omega_2)$.

(E-MYROLE) We reason like in the previous case. The thesis trivially follows by making use of the fact that the premise $o\langle\bar{v}\rangle \not\Subset \theta(\lambda)$ implies that rule (T-REC) in Table 13 cannot be applied since the premise $\omega \vdash_\theta o\langle\bar{v}\rangle \in myRole(x)$, i.e. $\omega \vdash_\theta x : \langle \pi_1 \cup \{o\langle\bar{v}\rangle\}, \pi_2\rangle$, does not hold.

(E-MSG$_1$) and (E-MSG$_2$) These cases are proved in a way similar to the previous case. Consider for example the case of rule (E-MSG$_1$). The premise $\bar{v} \not\sqsubseteq \bar{v}'$ with $\omega \vdash_\theta \bar{u} : \bar{v}'$ implies that rule (T-INV) in Table 13 cannot be applied. But application of rule (T-INV) is needed in the derivation of $\theta \vdash N$, since $m \gg a$ can execute an invoke. Thus, $\theta \vdash N$ cannot be derived. In the case of rule (E-MSG$_2$), the proof proceeds similarly by observing that the premise $\bar{\upsilon}' \not\sqsubseteq \bar{\upsilon}$ with $\omega \vdash_\theta \bar{w} : \bar{\upsilon}'$ prevents application of rule (T-REC) that is necessary in the derivation of $\theta \vdash N$.

(E-ASG) Also in this case the proof proceeds as the previous cases. Indeed, the premise $\upsilon \not\sqsubseteq \upsilon'$ with $\omega \vdash_\theta x : \upsilon$ and $\omega \vdash_\theta u : \upsilon'$ prevents application of rule (T-ASG) that is necessary in the derivation of $\theta \vdash N$.

The case of (E-PAR) follows by induction. Indeed, suppose that $N_1 \parallel N_2 \Uparrow^\theta_\lambda$ because, for example, $N_1 \Uparrow^\theta_\lambda$. By induction hypothesis we cannot infer that $\theta \vdash N_1$ and, therefore, we cannot apply (T-NET), the only possible rule to infer $\theta \vdash N_1 \parallel N_2$. □

We can finally prove our major result stating that well-typed networks are free from errors throughout their evolution.

Corollary 1 (Type soundness) *Let $\theta \vdash N$. Then $N' \Uparrow^\theta_\lambda$ does not hold for every addresses λ and network N' such that $N \rightarrowtail^* N'$ (where $\rightarrowtail^*$ denotes the reflexive and transitive closure of $\rightarrowtail$).*

Proof Theorem 1 implies that $\theta \vdash N'$, while Theorem 2 implies that $N' \Uparrow^\theta_\lambda$ does not hold for every address λ of N'. □

5 WS-CALCULUS at work

In this section, we apply our framework to illustrate two commonly used interaction patterns. The first pattern describes a shipping service scenario borrowed from the WS-BPEL official specification document [61, Sect. 15.1]. This example will allow us to illustrate both how to use WS-CALCULUS to formalise WS-BPEL programs and most of the language features, including correlation sets, shared variables and control flow structures. The second pattern will allow us to explain how to use our framework to analyse a generic asynchronous interaction pattern that offers a way to exchange messages, via an intermediary service, with services requiring synchronous interactions. This last pattern allows such synchronous services and their requestors to elaborate messages independently, by remaining temporally decoupled. Specific instantiations of asynchronous interaction patterns are catalogued in [29] (see, e.g., *Asynchronous Queuing* and *Service Callback* patterns).

For readability, long typing inferences are split in a few parts, each one may refer the other parts. As a matter of notation, if $\langle\mathbf{k}\rangle$ occurs in the left hand side of the conclusion of an inference then it denotes the whole inference, while if $(\mathbf{k})$ occurs in a premise then it is a reference to the inference whose conclusion is labelled by $\langle\mathbf{k}\rangle$ and has to be replaced by such an inference.

5.1 Defining a shipping service in WS-CALCULUS

The shipping service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The service offers two types of shipments: shipments where the items are held and shipped together and shipments where the items are shipped piecemeal until the order is fulfilled. A skeleton description follows:

```
receive shipOrder
if shipComplete
    send shipNotice
else
```

```
        itemsShipped := 0
        while itemsShipped < itemsTotal
        itemsCount   := opaque   // non-deterministic
                                 // assignment corresponding
                                 // e.g. to interaction with
                                 // a back-end system
        send shipNotice
        itemsShipped = itemsShipped + itemsCount
```

The `portType` and `partnerLinkType` descriptions representing the collaborations between the shipping service and the customer are reported in the following simplified WSDL elements:

```
  <wsdl:definitions>
    <portType name="shippingServicePT">
      <operation name="shippingRequest">
        <input message="shippingRequestMsg" />
      </operation>
    </portType>

    <portType name="shippingServiceCustomerPT">
      <operation name="shippingNotice">
        <input message="shippingNoticeMsg" />
      </operation>
    </portType>

    <partnerLinkType name="shippingLT">
      <role name="shippingService" portType="shippingServicePT" />
      <role name="shippingServiceCustomer"
            portType="shippingServiceCustomerPT" />
    </partnerLinkType>
  </wsdl:definitions>
```

The partner link type `shippingLT` represents the collaboration dependencies between the shipping service and the requesting customers that must provide a callback operation to enable notices to be sent (through the port type `shippingServiceCustomerPT`). Notably, also the customer port type is defined here.

The WS-BPEL program corresponding to the above description follows (to make the reading of the code easier, we have omitted irrelevant details[7] and highlighted the basic activities `receive`, `invoke` and `assign`):

```
<process name="shippingService">
  <partnerLinks>
    <partnerLink name="customer" partnerLinkType="shippingLT"
      partnerRole="shippingServiceCustomer"
      myRole="shippingService" />
  </partnerLinks>
  <correlationSets>
    <correlationSet name="shipOrder" properties="shipOrderID" />
  </correlationSets>
  <sequence>
  <receive partnerLink="customer"
    operation="shippingRequest"
    variable="shipRequest">
    <correlations>
```

[7]The fully detailed version of the WS-BPEL process and the associated WSDL document can be found in [61].

```
      correlation set="shipOrder" initiate="yes" />
    </correlations>
  </receive>

<if>
<condition>
  bpel:getVariableProperty('shipRequest','shipComplete')
</condition>
<sequence>

  <assign>
    <copy>
      <from variable="shipRequest" property="shipOrderID" />
      <to variable="shipNotice" property="shipOrderID" />
    </copy>
    <copy>
      <from variable="shipRequest" property="itemsCount" />
      <to variable="shipNotice" property="itemsCount" />
    </copy>
  </assign>

  <invoke partnerLink="customer" operation="shippingNotice"
    inputVariable="shipNotice">
    <correlations>
      <correlation set="shipOrder" />
    </correlations>
  </invoke>

</sequence>
<else>
  <sequence>
    <assign>
      <copy>
        <from>0</from>
        <to>$itemsShipped</to>
      </copy>
    </assign>

    <while>
      <condition>
        $itemsShipped
        &lt;
        bpel:getVariableProperty('shipRequest','itemsTotal')
      </condition>
      <sequence>

        <assign>
          <copy>
            <opaqueFrom />
            <to variable="shipNotice" property="shipOrderID" />
          </copy>
          <copy>
            <opaqueFrom />
            <to variable="shipNotice" property="itemsCount" />
          </copy>
        </assign>

        <invoke partnerLink="customer" operation="shippingNotice"
          inputVariable="shipNotice">
          <correlations>
            <correlation set="shipOrder" />
          </correlations>
        </invoke>
        <assign>
          <copy>
            <from>
            $itemsShipped
            +
            bpel:getVariableProperty('shipNotice','itemsCount')
            </from>
            <to>$itemsShipped</to>
          </copy>
        </assign>
        </sequence>
      </while>
    </sequence>
  </else>
  </if>
  </sequence>
</process>
```

A deployment of the above WS-BPEL process is given in WS-CALCULUS by the following node:

$$S ::^{\omega_S} \{ [\, C/x_C, 0/y_{shipped}] > a_S \}$$

We assume that the partner link x_C is initialized to a customer address C providing the operation $o_{shippingNotice}$. We use the following local type ω_S and activity a_S:

$$\begin{aligned}\omega_S = \{&x_C : \langle [\, o_{shippingRequest}\langle \text{INT, BOOL, INT}\rangle], [\, o_{shippingNotice}\langle \text{INT, INT}\rangle]\rangle,\\ &z_{id} : \text{INT},\ z_c : \text{BOOL},\ z_{items} : \text{INT},\ y_{shipped} : \text{INT},\ y_{count} : \text{INT}\}\end{aligned}$$

$$\begin{aligned}a_S = {} & \mathbf{rec}(x_C : o_{shippingRequest} : z_{id}, z_c, z_{items})\,;\\ & \mathbf{if}\ z_c\ \mathbf{then}\\ & \quad \mathbf{inv}(x_C : o_{shippingNotice} : z_{id}, z_{items})\\ & \mathbf{else}\\ & \quad \mathbf{while}\ y_{shipped} < z_{items}\ \mathbf{do}\\ & \quad\quad \mathbf{asg}(y_{count} : rand(z_{items} - y_{shipped}))\,;\\ & \quad\quad \mathbf{inv}(x_C : o_{shippingNotice} : z_{id}, y_{count})\,;\\ & \quad\quad \mathbf{asg}(y_{shipped} : y_{shipped} + y_{count})\end{aligned}$$

Here x_C is the partner link associated to the customer service, $o_{shippingRequest}$ is the operation used to receive the shipping request, and z_{id}, z_c and z_{items} are the variables used for storing the request message: z_{id} stores the order identifier which is used to correlate the ship notice(s) with the ship order, z_c stores a boolean indicating if the order has to be shipped complete or not, and z_{items} stores the number of items in the order. Shipping notices are sent to customers using the address C stored in x_C and the operation $o_{shippingNotice}$. A notice message is a tuple composed of the order identifier and the number of items in the shipping notice. When partial shipment is acceptable, $y_{shipped}$ is used to record the number of items already shipped. Expression $rand(k)$ is a function that returns a random positive integer number not greater than k and represents an internal interaction with the back-end system.

Now, consider the following network containing the deployment of the shipping service definition and a customer instance invoking the service:

$$\begin{aligned}N \equiv {} & S ::^{\omega_S} \{ [\, C/x_C, 0/y_{shipped}] > a_S \}\\ & \parallel C ::^{\omega_C} \{ [\, S/x_S, 123/y_{id}, \mathbf{false}/y_c, 50/y_{items}, 0/z_{shipped}] \gg a_C \}\end{aligned}$$

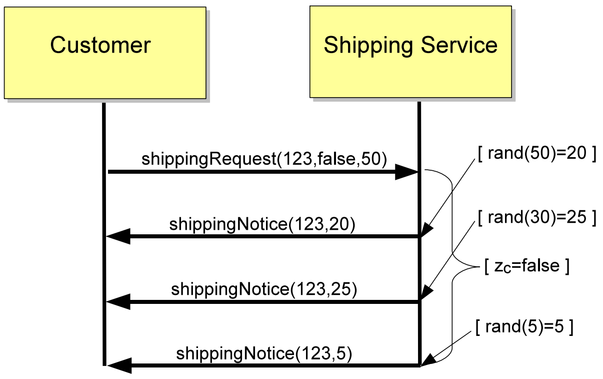

Fig. 1 A computation in the shipping service scenario

where the local type ω_C and activity a_C of the customer are defined as follows:

$$\begin{aligned}\omega_C = \{&x_S : \langle [\,o_{shippingNotice}\langle \text{INT}, \text{INT}\rangle],\ [\,o_{shippingRequest}\langle \text{INT}, \text{BOOL}, \text{INT}\rangle]\rangle,\\ &y_{id} : \text{INT},\ y_c : \text{BOOL},\ y_{items} : \text{INT},\ z_{shipped} : \text{INT},\ z_{count} : \text{INT}\}\end{aligned}$$

$$\begin{aligned}a_C = {}&\mathbf{inv}(x_S : o_{shippingRequest} : y_{id}, y_c, y_{items})\,;\\ &\mathbf{while}\, z_{shipped} < y_{items}\ \mathbf{do}\\ &\quad \mathbf{rec}(x_S : o_{shippingNotice} :!y_{id}, z_{count})\,;\\ &\quad \mathbf{asg}(z_{shipped} : z_{shipped} + z_{count})\end{aligned}$$

At the first computational step, the customer's invocation is delivered to the node S. The request is then consumed and an instance of the shipping service is created. The computation can now go on, e.g., as shown in the customized UML sequence diagram of Fig. 1, where it is supposed that the shipment successfully completes.

We could easily prove that the network N will remain error-free by showing that N is well-typed under the type environment θ, i.e. $\theta \vdash N$, for θ defined by the following bindings:

$$\begin{aligned}\theta(S) &= \{\,[\,o_{shippingRequest}\langle \text{INT}, \text{BOOL}, \text{INT}\rangle]\,\}\\ \theta(C) &= \{\,[\,o_{shippingNotice}\langle \text{INT}, \text{INT}\rangle]\,\}\end{aligned}$$

5.2 Analysing an asynchronous messaging broker

Although business processes do not naturally fit synchronous client-server collaborations, as mentioned in the Introduction, request-response operations are frequently used, even to invoke long-running web services. In fact, avoiding synchronous interactions in real business scenarios is not a simple task. In addition, when services developed by third parties are involved, signatures of synchronous operations cannot be directly modified and rendered asynchronous through pairs of one-way interactions. To overcome this problem, *asynchronous messaging* patterns [29] may be exploited.

Consider a service resource R that needs a large amount of time for message elaboration and requires its consumer C to interact with it synchronously. Here, for example, we let R receive messages containing an integer value n and send back responses containing the nth prime number. An intermediary service Q allows customers to communicate with R asynchronously by providing an indirect access mechanism. For this, service Q requires the customers to provide a callback address to which it can send the response message.

Although the synchronous interaction between R and Q is simply simulated, since only one-way operations are provided in WS-CALCULUS, our analysis is not undermined.

The asynchronous messaging network is modelled in WS-CALCULUS as

$$N \equiv C ::^{\omega_C} \{m_C \gg a_C\} \parallel Q ::^{\omega_Q} \{m_Q > a_Q\} \parallel R ::^{\omega_R} \{m_R > a_R\}$$

C is a process instance that behaves according to the following state m_C and activity a_C:

$$m_C \equiv [\,Q/x_Q, C/x_C, 1207/y] \qquad a_C \equiv \mathbf{inv}(x_Q : o_{req} : x_C, y)\,; \\ \mathbf{rec}(x_Q : o_{res} : z, !y)$$

Activity a_C sends a message containing the callback address C and the correlation value 1207 to service Q. Service Q, at a later point in time, sends a response containing also the necessary correlation information. This explains the presence of the join-variable $!y$, whose role is to correlate the value of y into the response with the value 1207 into the request. This would turn out to be necessary in presence of more than one customer instance within node C. The local type ω_C contains definitions of the partner link type $x_Q : \langle \pi, [\,o_{req}\langle\{\pi\}, \text{INT}\rangle]\rangle$ and value types $x_C : \{\pi\}$, $y : \text{INT}$ and $z : \text{INT}$, where π denotes the port type $[\,o_{res}\langle \text{INT}, \text{INT}\rangle]$. Notice that the partner link x_Q is used here to model the collaboration between services C and Q.

Q is a service that must be first instantiated by a request from the customer C. Its definition follows:

$$m_Q \equiv [\,R/x_R] \qquad a_Q \equiv \mathbf{rec}(x_C : o_{req} : x_C, y)\,; \mathbf{inv}(x_R : o_{num} : y)\,; \\ \mathbf{rec}(x_R : o_{prime} : z, !y)\,; \mathbf{inv}(x_C : o_{res} : z, y)$$

For the sake of simplicity, services Q and R are configured to be ready to communicate without any preliminary address exchange, i.e. each service already knows the address of the other one. When invoked, service Q creates an instance that will forward the received request to service R. Notice that service Q will use a join-variable $!y$ for message correlation. The local type ω_Q contains the partner link types $x_C : \langle [\,o_{res}\langle\{\pi\}, \text{INT}\rangle], \pi\rangle$ and $x_R : \langle [\,o_{prime}\langle \text{INT}, \text{INT}\rangle], [\,o_{num}\langle \text{INT}\rangle]\rangle$ and the value types $y : \text{INT}$ and $z : \text{INT}$, where also in this case π denotes the port type $[\,o_{res}\langle \text{INT}, \text{INT}\rangle]$.

Finally, service R is defined as follows:

$$m_R \equiv [\,Q/x_Q] \qquad a_R \equiv \mathbf{rec}(x_Q : o_{num} : y)\,; \mathbf{asg}(z : prime(y))\,; \\ \mathbf{inv}(x_Q : o_{prime} : z, y)$$

When invoked, service R creates an instance that will elaborate the received value and send the response back to service Q. Since no other message is expected from the intermediary service, service R does not require any correlation information. Notably, the expression $prime(n)$ denotes a function which returns the nth prime number. The local type ω_R contains the partner link type $x_Q : \langle [\,o_{num}\langle \text{INT}\rangle], [\,o_{prime}\langle \text{INT}, \text{INT}\rangle]\rangle$ and the value types $y : \text{INT}$ and $z : \text{INT}$.

According to our framework, to ensure that N will remain error-free, it suffices to prove that the judgement $\theta \vdash N$ can be derived, i.e. that N is well-typed w.r.t. the typing environment θ defined by the following bindings:

$$\theta(C) = \{\,[\,o_{res}\langle \text{INT}, \text{INT}\rangle]\,\}$$

$$\theta(Q) = \{\,[\,o_{req}\langle\{\,[\,o_{res}\langle \text{INT}, \text{INT}\rangle]\,\}, \text{INT}\rangle], [\,o_{prime}\langle \text{INT}, \text{INT}\rangle]\,\}$$

$$\theta(R) = \{\,[\,o_{num}\langle \text{INT}\rangle]\,\}$$

Table 15 Type inference for the consumer service $\omega_C \vdash_\theta m_C \gg a_C$

$$\frac{\omega_C \vdash_\theta x_C : \{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\} \quad \omega_C \vdash_\theta y : \mathrm{INT}}{\langle \mathbf{4}\rangle \omega_C \vdash_\theta x_C,\ y : \{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\ \mathrm{INT}}\ (\textsc{t-tuple})$$

$$\frac{\omega_C \vdash_\theta o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle \in \mathit{myRole}(x_Q) \qquad \dfrac{\omega_C \vdash_\theta z : \mathrm{INT} \quad \omega_C \vdash_\theta !y : \mathrm{INT}}{\omega_C \vdash_\theta z,\ !y : \mathrm{INT},\ \mathrm{INT}}\ (\textsc{t-tuple})}{\langle \mathbf{3}\rangle\ \ \omega_C \vdash_\theta \mathbf{rec}(x_Q : o_{res} : z, !y)}\ (\textsc{t-rec})$$

$$\frac{\omega_C \vdash_\theta o_{res}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle \in \mathit{partnerRole}(x_Q) \qquad (\mathbf{4})}{\langle \mathbf{2}\rangle \omega_C \vdash_\theta \mathbf{inv}(x_Q : o_{req} : x_C, y)}\ (\textsc{t-inv})$$

$$\frac{\dfrac{\dfrac{(\mathbf{1} - \text{Table 16}) \qquad \dfrac{(\mathbf{2}) \qquad (\mathbf{3})}{\omega_C \vdash_\theta \mathbf{inv}(x_Q : o_{req} : x_C, y)\,;\mathbf{rec}(x_Q : o_{res} : z, !y)}\ (\textsc{t-seq})}{\omega_C \vdash_\theta \mathbf{asg}(x_Q, x_C, y : Q, C, 1207)\,;\mathbf{inv}(x_Q : o_{req} : x_C, y)\,;\mathbf{rec}(x_Q : o_{res} : z, !y)}\ (\textsc{t-seq})}{\omega_C \vdash_\theta [\,] \gg \mathbf{asg}(x_Q, x_C, y : Q, C, 1207)\,;\mathbf{inv}(x_Q : o_{req} : x_C, y)\,;\mathbf{rec}(x_Q : o_{res} : z, !y)}\ (\textsc{t-inst1})}{\omega_C \vdash_\theta [Q/x_Q, C/x_C, 1207/y] \gg \mathbf{inv}(x_Q : o_{req} : x_C, y)\,;\mathbf{rec}(x_Q : o_{res} : z, !y)}\ (\textsc{t-inst2})$$

Table 16 Type inference for the arguments of $\omega_C \vdash_\theta \mathbf{asg}(x_Q, x_C, y : Q, C, 1207)$

$$\frac{\theta(Q) = \{[o_{req}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle], [o_{prime}\langle \mathrm{INT},\mathrm{INT}\rangle]\}}{\langle \mathbf{5}\rangle\ \ \omega_C \vdash_\theta Q : \{[o_{res}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle], [o_{prime}\langle \mathrm{INT},\mathrm{INT}\rangle]\}}\ (\textsc{t-addr})$$

$$\frac{(\mathbf{5}) \qquad \dfrac{\theta(C) = \{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\}}{\omega_C \vdash_\theta C : \{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\}}\ (\textsc{t-addr}) \qquad \dfrac{1207 \in \mathtt{Int}}{\omega_C \vdash_\theta 1207 : \mathrm{INT}}\ (\textsc{t-val})}{\langle \mathbf{3}\rangle\ \ \omega_C \vdash_\theta Q,\ C,\ 1207 : \{[o_{req}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle], [o_{prime}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\ \{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\ \mathrm{INT}}\ (\textsc{t-tuple})$$

$$\frac{\dfrac{x_Q : \langle[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle], [o_{req}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle]\rangle \in \omega_C}{\omega_C \vdash_\theta x_Q : \langle[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle], [o_{req}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle]\rangle}\ (\textsc{t-link})}{\langle \mathbf{4}\rangle\ \ \omega_C \vdash_\theta x_Q : \{[o_{req}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle]\}}\ (\textsc{t-role})$$

$$\frac{(\mathbf{4}) \qquad \dfrac{x_C : \{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\} \in \omega_C}{\omega_C \vdash_\theta x_C : \{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\}}\ (\textsc{t-var}) \qquad \dfrac{y : \mathrm{INT} \in \omega_C}{\omega_C \vdash_\theta y : \mathrm{INT}}\ (\textsc{t-var})}{\langle \mathbf{2}\rangle\ \ \omega_C \vdash_\theta x_Q,\ x_C,\ y : \{[o_{req}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle]\},\ \{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\ \mathrm{INT}}\ (\textsc{t-tuple})$$

$$\frac{(\mathbf{2}) \qquad (\mathbf{3}) \qquad \{[o_{req}\langle\{[o_{res}\langle \mathrm{INT},\mathrm{INT}\rangle]\},\mathrm{INT}\rangle]\} \sqsubseteq \theta(Q)}{\langle \mathbf{1}\rangle\ \ \omega_C \vdash_\theta \mathbf{asg}(x_Q, x_C, y : Q, C, 1207)}\ (\textsc{t-asg})$$

By rules (T-NET) and (T-NODE), this means that we must check that the judgements $\omega_C \vdash_\theta m_C \gg a_C$, $\omega_Q \vdash_\theta m_Q > a_Q$ and $\omega_R \vdash_\theta m_R > a_R$ hold. This is what the inferences in Tables 15, 16, 17 and 18 show. For the sake of readability, obvious and repeated typing inferences for partner links, values and operation parameters, as well as obvious subtyping checks, are omitted. In particular, we have omitted the checks that states m_Q and m_R are well-typed (as instead done for state m_C in Table 16) and directly show the inferences of judgements $\omega_Q \vdash_\theta a_Q$ and $\omega_R \vdash_\theta a_R$.

We just comment on the most significant inferences for service C in Tables 15 and 16, since those for services Q and R are very similar. Initially, by applying rules (T-INST2), (T-INST1) and (T-SEQ) in Table 15 and then rule (T-ASG) in Table 16, well-typedness of state m_C is checked. In Table 15, rules (T-INV) and (T-REC) permit checking if the partner link type of x_Q provides the operation types for o_{req} and o_{res} according to their respective argument types. For example, operation o_{req} is checked by verifying that its definition belongs to some port type provided by x_Q and that its signature for arguments x_C and y is well-typed.

In Table 16, rule (T-ROLE) permits checking the binding for x_Q by verifying that the address type of Q is a subtype of the value type of x_Q. In particular, it is checked that the

Table 17 Type inference for the intermediary service $\omega_Q \vdash_\theta a_Q$

$$\frac{\omega_Q \vdash_\theta o_{res}\langle \textsc{int}, \textsc{int}\rangle \in \mathit{partnerRole}(x_C) \quad \omega_Q \vdash_\theta z,\ y : \textsc{int},\ \textsc{int}}{\langle \mathbf{4}\rangle\ \ \omega_Q \vdash_\theta \mathbf{inv}(x_C : o_{rec} : z, y)}\ (\textsc{t-inv})$$

$$\frac{\omega_Q \vdash_\theta o_{prime}\langle \textsc{int}, \textsc{int}\rangle \in \mathit{myRole}(x_R) \quad \omega_Q \vdash_\theta z,\ !y : \textsc{int},\ \textsc{int}}{\langle \mathbf{3}\rangle\ \ \omega_Q \vdash_\theta \mathbf{rec}(x_R : o_{prime} : z, !y)}\ (\textsc{t-rec})$$

$$\frac{\omega_Q \vdash_\theta o_{num}\langle \textsc{int}\rangle \in \mathit{partnerRole}(x_R) \quad \omega_Q \vdash_\theta y : \textsc{int}}{\langle \mathbf{2}\rangle\ \ \omega_Q \vdash_\theta \mathbf{inv}(x_R : o_{num} : y)}\ (\textsc{t-inv})$$

$$\frac{\omega_Q \vdash_\theta o_{req}\langle \{\,[\,o_{res}\langle \textsc{int}, \textsc{int}\rangle]\,\}, \textsc{int}\rangle \in \mathit{myRole}(x_C) \quad \omega_Q \vdash_\theta x_C,\ y : \{\,[\,o_{res}\langle \textsc{int}, \textsc{int}\rangle]\,\},\ \textsc{int}}{\langle \mathbf{1}\rangle\ \ \omega_Q \vdash_\theta \mathbf{rec}(x_C : o_{req} : x_C, y)}\ (\textsc{t-rec})$$

$$\frac{(\mathbf{1}) \qquad \dfrac{(\mathbf{2}) \qquad \dfrac{(\mathbf{3}) \qquad (\mathbf{4})}{\omega_Q \vdash_\theta \mathbf{rec}(x_R : o_{prime} : z, !y) \,;\, \mathbf{inv}(x_C : o_{res} : z, y)}\ (\textsc{t-seq})}{\omega_Q \vdash_\theta \mathbf{inv}(x_R : o_{num} : y) \,;\, \mathbf{rec}(x_R : o_{prime} : z, !y) \,;\, \mathbf{inv}(x_C : o_{res} : z, y)}\ (\textsc{t-seq})}{\omega_Q \vdash_\theta \mathbf{rec}(x_C : o_{req} : x_C, y) \,;\, \mathbf{inv}(x_R : o_{num} : y) \,;\, \mathbf{rec}(x_R : o_{prime} : z, !y) \,;\, \mathbf{inv}(x_C : o_{res} : z, y)}\ (\textsc{t-seq})$$

Table 18 Type inference for the resource service $\omega_R \vdash_\theta a_R$

$$\frac{\omega_Q \vdash_\theta o_{prime}\langle \textsc{int}, \textsc{int}\rangle \in \mathit{partnerRole}(x_R) \quad \omega_Q \vdash_\theta z,\ !y : \textsc{int},\ \textsc{int}}{\langle \mathbf{3}\rangle\ \ \omega_R \vdash_\theta \mathbf{inv}(x_R : o_{prime} : z, !y)}\ (\textsc{t-inv})$$

$$\frac{\omega_R \vdash_\theta z : \textsc{int} \qquad \dfrac{\omega_R \vdash_\theta y : \textsc{int}}{\omega_R \vdash_\theta \mathit{prime}(y) : \textsc{int}}\ (\textsc{t-prime})}{\langle \mathbf{2}\rangle\ \ \omega_R \vdash_\theta \mathbf{asg}(z : \mathit{prime}(y))}\ (\textsc{t-asg})$$

$$\frac{\omega_R \vdash_\theta o_{num}\langle \textsc{int}\rangle \in \mathit{myRole}(x_Q) \quad \omega_R \vdash_\theta y : \textsc{int}}{\langle \mathbf{1}\rangle\ \ \omega_R \vdash_\theta \mathbf{rec}(x_Q : o_{num} : y)}\ (\textsc{t-rec})$$

$$\frac{(\mathbf{1}) \qquad \dfrac{(\mathbf{2}) \qquad (\mathbf{3})}{\omega_Q \vdash_\theta \mathbf{asg}(z : \mathit{prime}(y)) \,;\, \mathbf{inv}(x_Q : o_{prime} : z, y)}\ (\textsc{t-seq})}{\omega_R \vdash_\theta \mathbf{rec}(x_Q : o_{num} : y) \,;\, \mathbf{asg}(z : \mathit{prime}(y)) \,;\, \mathbf{inv}(x_Q : o_{prime} : z, y)}\ (\textsc{t-seq})$$

value type of x_Q, that is $\{\,[\,o_{req}\langle\{\,[\,o_{res}\langle \textsc{int}, \textsc{int}\rangle]\,\}, \textsc{int}\rangle]\,\}$, is a subset of $\theta(Q)$. Rule ($\textsc{t-var}$) instead permits checking if the type of the value variable x_C is a subtype of the address type of C. The other inferences in Table 16 are obvious.

The inferences in Tables 15, 16, 17 and 18 prove that the network N behaves correctly. Now, we slightly modify N so that its execution would eventually reach an error configuration; we show that our type system can statically detect this situation. Indeed, suppose we have a service consumer C' obtained from C by removing the receive activity and using a local type $\omega_{C'}$ for which the type of x_Q is now $\langle [/], [\,o_{req}\langle\{\,[/]\,\}, \textsc{int}\rangle]\rangle$ and the type of x_C is $\{\,[/]\,\}$.

Services C' and Q interact as follows:

$$\begin{aligned}
&C' ::^{\omega_{C'}} \{ m_C \gg \mathbf{inv}(x_Q : o_{req} : x_C, y) \} \\
&\ \parallel\ Q ::^{\omega_Q} \{ m_Q > a_Q \} \\
&\ \parallel\ R ::^{\omega_R} \{ m_R > a_R \} \\
&\rightarrowtail\ \rightarrowtail \\
&C' ::^{\omega_{C'}} \{ m_C \gg \mathbf{empty} \} \\
&\ \parallel\ Q ::^{\omega_Q} \{ m_Q > a_Q,\ \ m_Q \circ [\,C/x_C, 1207/y] \gg (\mathbf{inv}(x_R : o_{num} : y) \,;\, \ldots) \} \\
&\ \parallel\ R ::^{\omega_R} \{ m_R > a_R \} \\
&\ \equiv\ N'
\end{aligned}$$

By rule (E-PARTNERROLE), an error occurs at node Q because, according to ω_Q, x_C has type $\langle \pi_1, [\, o_{res}\langle \text{INT}, \text{INT}\rangle]\rangle$ and, according to $\omega_{C'}$, $\theta(C') = \{\, [/]\, \}$, hence $o_{res}\langle \text{INT}, \text{INT}\rangle \notin \theta(C')$. In fact, our type system statically detects this error, since the judgement $\omega_Q \vdash_\theta$ $\mathbf{inv}(x_Q : o_{rec} : x_C, y)$ does not hold. Indeed, the premise $\{\, [\, o_{req}\langle \{\, [\, o_{res}\langle \text{INT}, \text{INT}\rangle]\, \}, \text{INT}\rangle]\, \} \sqsubseteq \{\, [/]\, \}$ is not satisfied. Hence, service C', as well as the whole network, is not well-typed.

6 Related work

Following [59], we have put forward the use of a type system to define basic contracts for web services. An alternative approach that is worth to be mentioned is based on the schema language introduced in [21]. This work presents a very basic contract language modelling WSDL documents as schema types with channels. In particular, a sophisticated (and, at the same time, computationally efficient) subschema relation has been defined for checking if an exchanged XML document conforms to its schema type. However, such contract language, differently from ours, does not take into account partner links types, which play a central role in the relationship between WS-BPEL processes and their associated WSDL documents. Indeed, rather than to WS-BPEL, it has been applied in [22] to the programming language `PiDuce` [22], an extension of the asynchronous π-calculus [4] with XML values and datatypes, that does not resort to the notion of partner link. A general theory of contracts for web services is developed in [23]. Differently from ours, that work abstracts from the language used to express web services and does not take into account the many specific mechanisms about execution of WS-BPEL programs we have modelled, as e.g. address-passing, correlation and service instantiation.

Other well-known service-oriented approaches that use the concept of dual (or complementary) types are based on session-types and have been explored in e.g. [8, 11, 12, 19, 20, 32, 40, 41, 49, 70]. Session-types are emerged as a powerful tool for taking into account behavioural and non-functional properties of conversational interactions. They permit to express and enforce many relevant policies for, e.g., constraining the sequences of messages accepted by services, ensuring service interoperability and compositionality, and guaranteeing absence of deadlock in service composition. Moreover, other form of behavioural types, see e.g. [2, 3, 18, 24, 42, 45, 46], can be exploited to express other dynamic aspects that cannot be captured by only relying on sessions. Our work differs for the fact that, by electing WS-BPEL and WSDL as starting points, we distill a core model of interaction whose dynamical aspects at linguistic-level are fully taken into account by relying only on the 'conversational data' for message delivering (i.e. partner links, operation signatures and correlation data) contained in WS-BPEL and WSDL specifications, rather than on session-oriented constructs. These latter constructs indeed rely on private channels (the so-called *session channels*) along which communication takes place. Instead, in WS-CALCULUS, as well as in WS-BPEL, the notion of private channel is not exploited and, hence, the assumptions at the base of the session-types theory do not hold. At type-level, instead, in our technology-oriented investigation we do not consider behavioural aspects (for, e.g., constraining the sequences of messages accepted by services or guaranteeing absence of deadlock in service composition), since they go further on what the technology (in particular WSDL) currently supports.

A secondary, but not minor, 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. Since we wanted to study those problems arising when executing WS-BPEL processes, then we have focused on service orchestration rather than on service choreography, that instead provides a means to describe service interactions in a top-view way (these aspects have been considered, e.g., in [16, 20]). The mechanism of correlation sets was first investigated in [67], that however only consider interaction of different instances of a single business process.

Several formal semantics of WS-BPEL were proposed in the literature. Many of these efforts aim at formalizing a semantics for WS-BPEL using Petri nets [39, 57, 64] or workflow [1, 63], but do not cover such *dynamical aspects* as service instantiation and message correlation. Another bunch of related works using process calculi focus instead on small and relatively simple fragments of WS-BPEL (e.g. [27, 33, 37]) or are targeted to formalize the semantics of WS-BPEL by encoding parts of the language into more foundational languages, such as π-calculus (e.g. [28, 58]). A very general and flexible framework for error recovery has been introduced in [38]; this framework extends SOCK [37], a language for service composition with dynamic compensation, and models in particular the dependency between fault handling and the request-response communication pattern. The language closest to WS-CALCULUS is B*lite* [55], a lightweight language for web services orchestration designed around some of WS-BPEL relevant features like process termination, message correlation, long-running business transactions and compensation handlers. Although WS-CALCULUS and B*lite* are built on a common set of WS-BPEL activities, there are significant differences between them. Indeed, while WS-CALCULUS aims at formalising the semantics of a fragment of WS-BPEL for enabling the investigation of its relationship with WSDL, B*lite* aims at being an alternative to WS-BPEL as a programming language for web services orchestration. Thus, to facilitate the task of programming orchestration, some B*lite* activities have simplified form and semantics w.r.t. the corresponding WS-BPEL and WS-CALCULUS ones. For example, to be faithful to WS-BPEL, the treatment of partner links and the implementation of the correlation mechanism in WS-CALCULUS are more complex than in B*lite*. The works presented in [38, 55] focus on fault and compensation handling aspects, which are not considered in our WSDL-based types. We expect that WS-CALCULUS can be extended to deal with them without significantly affecting our type theory (see the discussion on fault and compensation handling in Sect. 3).

Some other relevant related works are [10, 14, 15]. In the first two, the authors propose a formal approach to model compensation à la WS-BPEL in transactional calculi and present a detailed comparison with [17]. The third is an extension of the asynchronous π-calculus with long-running (scoped) transactions. As already said, most of these formalisms, however, do not model the different aspects of WS-BPEL in their completeness. One such aspect is represented by *timed activities* that are frequently exploited in service orchestration and are typically used for handling timeouts. For example, in WS-BPEL, activities *wait* and *pick* turn out to be essential for dealing with service transactions or with message losses. Thus, a service process could await a callback message for a certain amount of time after which, if no callback has been received, it invokes another operation or throws a fault. However, only a few process calculi for modelling WS-BPEL programs deal with timed activities. In particular, [50, 51] introduce webπ, a timed extension of the π-calculus tailored to study 'web transactions'. This very expressive language (putting together time aspects with transactional mechanisms) facilitates the modelling of long running transactions and the encoding of transactional constructs such as the *scope activity* of WS-BPEL. Geguang et al. [34, 35] present a timed calculus based on a more general notion of time, and an approach to verify WS-BPEL specifications with compensation/fault activities. Kitchin et al. [44] proposes a general purpose task orchestration language that manages timeouts as signals returned by dedicated services after some specified time intervals. We could integrate the WS-BPEL activity *wait* into WS-CALCULUS as it has been done in [54] for the service-oriented, correlation-based calculus COWS [53]. Since our types, like WSDL interfaces, do not deal with timed aspects, such extension of the calculus would not affect the type discipline introduced in this paper.

Almost all the efforts towards the formal verification of WS-BPEL applications rely on model checking techniques (a wide survey of these approaches is presented in [13]). A largely followed approach is based on (extensions of) Petri Nets. For example, [39] proposes a tool for transforming WS-BPEL specifications into Petri Nets that can be verified using the LoLA analyzer [65]. Similarly, [64] generates Petri Nets from WS-BPEL specifications for enabling a few different kinds of verification (e.g. reachability analysis). Many other works rely on the process meta language Promela. For instance, [31] presents the static analysis tool WSAT that takes as an input a WS-BPEL specification and, after a few translation steps, produces a Promela specification that can be analysed through the model checker SPIN. However, although Petri Nets and Promela can be a natural choice for encoding workflows, they seem not to fit well for such aspects as process instantiation, message correlation, shared variables and partner link-based collaborations that are particularly relevant for WS-BPEL. Moreover, since such approaches focus on the control flow, they ignore the data flow among the services participating to the orchestration. A verification approach that takes into account this issue by exploiting abstraction techniques is presented in [43]. Finally, other different approaches, such as e.g. [30, 48], exploit model checker tools based on Labelled Transition System models. Our work differs from all above for the proposed verification technique, which is indeed based on a type system rather than on a model checker. What is more, it also differs from all others because it formalises the relationship between WS-BPEL and WSDL, which is instead completely overlooked by the other approaches.

7 Concluding remarks

We have set a formal semantics framework for typing web services orchestration languages and, in particular, for clarifying the relationship between WS-BPEL programs and the associated WSDL documents, with special attention to asynchronous interactions. More specifically, we have introduced WS-CALCULUS, a foundational language specifically designed for modelling asynchronous interactions among web services, and a type system for it that forces a neat programming discipline. We have proved that the operational semantics of WS-CALCULUS and the type system are 'sound', and demonstrated feasibility and effectiveness of our approach by means of the specification and the analysis of two illustrative examples.

In our opinion, the potential of WS-BPEL as an integration platform must still be fully exploited. On the one hand, WSDL specifications are very basic: they only permit to specify names and types of request and callback operations, and how they can be combined. Although there are other technologies that permit to specify more complex interaction patterns (e.g. WSCL [6] provides behavioural descriptions through activity diagrams), they are not currently supported by WS-BPEL. On the other hand, at present, business processes abundantly use synchronous request-response interactions to integrate their complex collaborations. Making interactions asynchronous is beneficial to reduce the time a requestor spends waiting for responses. Therefore, WS-CALCULUS defines a model for describing the behavior of business processes based on asynchronous interactions between the processes and their partners. The collaboration with each partner occurs through port types (i.e. WSDL interfaces) and is described by making use of partner link types. WS-CALCULUS emphasizes the importance of this construct in support of defining peer-to-peer (and not request-response based, client-server) interactions. In practice, by using partner links, the asynchronous message exchange tends to become the primary mechanism with respect to the (still valid) mechanism of request-response.

Our theoretical framework could be the base of an effective software tool for identifying errors during the design phase of WS-BPEL programs. However, implementation issues have not been considered in this paper and are left for future work. Actually, our type system enables a syntax-directed type checking of WS-CALCULUS terms, thus we expect that it can be implemented by a suitable decidable algorithm applying the rules reported in Tables 10–13. In particular, a type checking tool could be written in Ocaml by following the approach developed in [47, 60] for π-calculus-based settings.

References

1. van der Aalst WMP, Lassen KB (2008) Translating unstructured workflow processes to readable BPEL: Theory and implementation. Inf Softw Technol 50(3):131–159
2. Acciai L, Boreale M (2008) Spatial and behavioral types in the pi-calculus. In: CONCUR. LNCS, vol 5201. Springer, Berlin, pp 372–386
3. Acciai L, Boreale M (2008) A type system for client progress in a service-oriented calculus. In: Concurrency, graphs and models. LNCS, vol 5065. Springer, Berlin, pp 642–658
4. Amadio R, Castellani I, Sangiorgi D (1998) On bisimulations for the asynchronous pi-calculus. Theor Comput Sci 195(2):291–324
5. Arkin A, Askary S, Fordin S, Jekeli W, Kawaguchi K, Orchard D, Pogliani S, Riemer D, Struble S, Takacsi-Nagy P, Trickovic I, Zimek S (2002) Web service choreography interface (WSCI) 1.0. Tech rep, W3C. Available at http://www.w3.org/TR/wsci/
6. Banerji A, Bartolini C, Beringer D, Chopella V, Govindarajan K, Karp A, Kuno H, Lemon M, Pogossiants G, Sharma S, Williams S (2002) Web services conversation language (WSCL) 1.0. Tech rep, W3C Note. Available at http://www.w3.org/TR/2002/NOTE-wscl10-20020314
7. Battle S, Bernstein A, Boley H, Grosof B, Gruninger M, Hull R, Kifer M, Martin D, McIlraith S, McGuinness D, Su J, Tabet S (2005) Semantic web services framework (SWSF) 1.0. Tech rep, SWSL Committee. Available at http://www.daml.org/services/swsf/1.0/
8. Bettini L, Coppo M, D'Antoni L, De Luca M, Dezani-Ciancaglini M, Yoshida N (2008) Global progress in dynamically interleaved multiparty sessions. In: CONCUR. LNCS, vol 5201. Springer, Berlin, pp 418–433
9. Boag S, Chamberlin D, Fernández M, Florescu D, Robie J, Siméon J (2007) Xquery 1.0: an XML query language. Tech rep, W3C. Available at http://www.w3.org/TR/xquery/
10. Bocchi L, Laneve C, Zavattaro G (2003) A calculus for long-running transactions. In: FMOODS. LNCS, vol 2884. Springer, Berlin, pp 124–138
11. Bonelli E, Compagnoni AB (2007) Multipoint session types for a distributed calculus. In: TGC. LNCS, vol 4912. Springer, Berlin, pp 240–256
12. Boreale M, Bruni R, De Nicola R, Loreti M (2008) Sessions and pipelines for structured service programming. In: FMOODS. LNCS, vol 5051. Springer, Berlin, pp 19–38
13. van Breugel F, Koshkina M (2006) Models and verification of BPEL. Tech rep, Department of Computer Science and Engineering, York University. Available at http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf
14. Bruni R, Butler M, Ferreira C, Hoare C, Melgratti H, Montanari U (2005) Comparing two approaches to compensable flow composition. In: CONCUR. LNCS, vol 3653. Springer, Berlin, pp 383–397
15. Bruni R, Melgratti H, Montanari U (2005) Theoretical foundations for compensations in flow composition languages. In: POPL. ACM, New York, pp 209–220
16. Busi N, Gorrieri R, Guidi C, Lucchi R, Zavattaro G (2005) Choreography and orchestration: a synergic approach for system design. In: ICSOC. LNCS, vol 3826. Springer, Berlin, pp 228–240
17. Butler M, Ferreira C (2004) An operational semantics for StAC, a language for modelling long-running business transactions. In: COORDINATION. LNCS, vol 2949. Springer, Berlin, pp 87–104
18. Caires L, Vieira H (2009) Conversation types. In: ESOP. LNCS, vol 5502. Springer, Berlin, pp 285–300
19. Carbone M, Honda K, Yoshida N (2007) A calculus of global interaction based on session types. In: DCM. ENTCS, vol 171. Elsevier, Amsterdam, pp 127–151
20. Carbone M, Honda K, Yoshida N (2007) Structured communication-centred programming for web services. In: ESOP. LNCS, vol 4421. Springer, Berlin, pp 2–17
21. Carpineti S, Laneve C (2006) A basic contract language for web services. In: ESOP. LNCS, vol 3924. Springer, Berlin, pp 197–213
22. Carpineti S, Laneve C, Padovani L (2009) `PiDuce`—a project for experimenting Web services technologies. Sci Comput Program 74(10):777–811

23. Castagna G, Gesbert N, Padovani L (2009) A theory of contracts for Web services. ACM Trans Program Lang Syst 31:5
24. Chaki S, Rajamani SK, Rehof J (2002) Types as models: model checking message-passing programs. In: POPL. ACM, New York, pp 45–57
25. Christensen E, Curbera F, Meredith G, Weerawarana S (2001) Web services description language (WSDL) 1.1. Tech rep, W3C. Available at http://www.w3.org/TR/wsdl/
26. Clark J, DeRose S (1999) XML path language (XPath) Version 1.0. Tech rep, W3C. Available at http://www.w3.org/TR/xpath/
27. Cook W, Patwardhan S, Misra J (2006) Workflow patterns in ORC. In: COORDINATION. LNCS, vol 4038. Springer, Berlin, pp 82–96
28. Dragoni N, Mazzara M (2010) A formal semantics for the WS-BPEL recovery framework: the pi-calculus way. In: WS-FM. LNCS, vol 6194. Springer, Berlin, pp 92–109
29. Erl T (2009) SOA design patterns. Prentice Hall, New York
30. Foster H, Uchitel S, Magee J, Kramer J (2006) LTSA-WS: a tool for model-based verification of web service compositions and choreography. In: ICSE. ACM, New York, pp 771–774
31. Fu X, Bultan T, Su J (2005) Synchronizability of conversations among web services. IEEE Trans Softw Eng 31(12):1042–1055
32. Gay S, Vasconcelos V (2010) Linear type theory for asynchronous session types. J Funct Program 20(1):19–50
33. Geguang P, Xiangpeng Z, Shuling W, Zongyan Q (2005) Semantics of BPEL4WS-like fault and compensation handling. In: FM. LNCS, vol 3582. Springer, Berlin, pp 350–365
34. Geguang P, Huibiao Z, Zongyan Q, Shuling W, Xiangpeng Z, Jifeng H (2006) Theoretical foundations of scope-based compensable flow language for web service. In: FMOODS. LNCS, vol 4037. Springer, Berlin, pp 251–266
35. Geguang P, Xiangpeng Z, Shuling W, Zongyan Q (2006) Towards the semantics and verification of BPEL4WS. In: DCM. ENTCS, vol 151. Elsevier, Amsterdam, pp 33–52
36. Gudgin M, Hadley M, Rogers T (2006) Web services addressing 1.0—Core. Tech rep, W3C. Available at http://www.w3.org/TR/ws-addr-core
37. Guidi C, Lucchi R, Gorrieri R, Busi N, Zavattaro G (2006) SOCK: a calculus for service oriented computing. In: ICSOC. LNCS, vol 4294. Springer, Berlin, pp 327–338
38. Guidi C, Lanese I, Montesi F, Zavattaro G (2008) On the interplay between fault handling and request-response service invocations. In: ACSD. IEEE, New York, pp 190–198
39. Hinz S, Schmidt K, Stahl C (2005) Transforming BPEL to Petri nets. In: BPM. LNCS, vol 3649. Springer, Berlin, pp 220–235
40. Honda K, Vasconcelos VT, Kubo M (1998) Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol 1381. Springer, Berlin, pp 122–138
41. Honda K, Yoshida N, Carbone M (2008) Multiparty asynchronous session types. In: POPL. ACM, New York, pp 273–284
42. Igarashi A, Kobayashi N (2004) A generic type system for the pi-calculus. Theor Comput Sci 311(1–3):121–163
43. Kazhamiakin R, Pistore M (2006) Static verification of control and data inWeb service compositions. In: ICWS. IEEE, New York, pp 83–90
44. Kitchin D, Cook W, Misra J (2006) A language for task orchestration and its semantic properties. In: CONCUR. LNCS, vol 4137. Springer, Berlin, pp 477–491
45. Kobayashi N (2003) Type systems for concurrent programs. In: UNU/IIST. LNCS, vol 2757. Springer, Berlin, pp 439–453
46. Kobayashi N, Suenaga K, Wischik L (2006) Resource usage analysis for the π-calculus. In: VMCAI. LNCS, vol 3855. Springer, Berlin, pp 298–312
47. Kobayashi N (2008) Typical: type-based static analyzer for the picalculus. Tool available at http://www.kb.ecei.tohoku.ac.jp/koba/typical
48. Kovács M, Gönczy L, Varró D (2008) Formal analysis of BPEL workflows with compensation by model checking. J Comput Syst Sci Eng 23(5):35–49
49. Lanese I, Martins F, Ravara A, Vasconcelos V (2007) Disciplining orchestration and conversation in service-oriented computing. In: SEFM. IEEE, New York, pp 305–314
50. Laneve C, Zavattaro G (2005) Foundations of web transactions. In: FoSSaCS. LNCS, vol 3441. Springer, Berlin, pp 282–298
51. Laneve C, Zavattaro G (2005) Web-pi at work. In: TGC. LNCS, vol 3705. Springer, Berlin, pp 182–194
52. Lapadula A, Pugliese R, Tiezzi F (2006) A WSDL-based type system for WS-BPEL. In: COORDINATION. LNCS, vol 4038. Springer, Berlin, pp 145–163
53. Lapadula A, Pugliese R, Tiezzi F (2007) A calculus for orchestration of web services. In: ESOP. LNCS, vol 4421. Springer, Berlin, pp 33–47

54. Lapadula A, Pugliese R, Tiezzi F (2007) C⌚WS: a timed service-oriented calculus. In: ICTAC. LNCS, vol 4711. Springer, Berlin, pp 275–290
55. Lapadula A, Pugliese R, Tiezzi F (2008) A formal account of WS-BPEL. In: COORDINATION. LNCS, vol 5052. Springer, Berlin, pp 199–215
56. Leymann F (2001) Web services flow language (WSFL 1.0). Tech rep, IBM. Available at http://xml.coverpages.org/wsfl.html
57. Lohmann N (2008) A feature-complete Petri net semantics for WS-BPEL 2.0. In: WSFM. LNCS, vol 4937. Springer, Berlin, pp 77–91
58. Mazzara M, Lucchi R (2007) A pi-calculus based semantics for WS-BPEL. J Log Algebr Program 70(1):96–118
59. Meredith L, Bjorg S (2003) Contracts and types. Commun ACM 46(10):41–47
60. Mezzina LG (2008) Typses: a tool for type checking session types. Tool available at http://www.di.unipi.it/mezzina
61. OASIS WSBPEL TC (2007) Web services business process execution language version 2.0. Tech rep, OASIS. Available at http://docs.oasis-open.org/wsbpel/2.0/OS/
62. Object Management Group (1996) The common object request broker: architecture and specification (CORBA 2.0). Tech rep, OMG. Available at http://www.corba.org/
63. Ouyang C, Dumas M, ter Hofstede AHM, van der Aalst WMP (2006) From BPMN process models to BPEL web services. In: ICWS. IEEE, New York, pp 285–292
64. Ouyang C, Verbeek E, van der Aalst W, Breutel S, Dumas M, ter Hofstede A (2007) Formal semantics and analysis of control flow in WS-BPEL. Sci Comput Program 67(2–3):162–198
65. Schmidt K (2000) LoLA—a low level analyser. In: ICATPN. LNCS, vol 1825. Springer, Berlin, pp 465–474
66. Thatte S (2001) Xlang: Web services for business process design. Tech rep, Microsoft. Available at http://xml.coverpages.org/xlang.html
67. Viroli M (2004) Towards a formal foundational to orchestration languages. In: WSFM. ENTCS, vol 105. Elsevier, Amsterdam, pp 51–71
68. Wright A, Felleisen M (1994) A syntactic approach to type soundness. Inf Comput 115(1):38–94
69. XMethods (2010). http://www.xmethods.com
70. Yoshida N, Vasconcelos VT (2007) Language primitives and type discipline for structured communication-based programming revisited: two systems for higher-order session communication. In: SecReT. ENTCS, vol 171/4. Elsevier, Amsterdam, pp 73–93