Session Types for BPEL

Jonathan Michaux[1], Elie Najm[1], and Alessandro Fantechi[2]

[1] Télécom Paristech, 46 rue Barrault, 75013, Paris, France
michaux@telecom-paristech.fr, najm@telecom-paristech.fr
[2] Universita' degli Studi di Firenze, via S. Marta 3 I-50139, Firenze, Italy
fantechi@dsi.unifi.it

Abstract. We address the general problem of interaction safety between orchestrated web services. By considering an essential subset of the BPEL orchestration language, we show how the session paradigm with session types can be used to address the problem. During a session, a client and a service can engage in a complex series of interactions. We introduce session types in order to prescribe the correct orderings of these interactions. Service providers must declare their provided and required session types. We define a typing algorithm that checks if an orchestrated service behaves according to its declared provided and required types. Using a compatibility and a subtyping relation defined on session types, we show that any collection of well typed service partners with compatible session types are interaction safe, i.e., involved partners never receive unexpected messages.

Keywords: Session types, Orchestration, BPEL, Formal semantics, Subtyping, Behavioural types

1 Introduction

In service-oriented computing, services are exposed over a network via well defined interfaces and specific communication protocols. The design of software as an orchestration of services is an active topic today. A service orchestration is a local view of a structured set of interactions with remote services. In this paper, we strive to determine whether or not interacting services are compatible. We state that interacting services are compatible if their execution does not lead to the exchange of unexpected messages or arguments.

The elementary construct in a web service interaction is a message exchange between two partner services. The message specifies the name of the operation to be invoked and bears arguments as its payload. An interaction can be long-lasting because multiple messages of different types can be exchanged in both directions before a service is delivered. The set of interactions supported by a service defines its behaviour. We argue that the high levels of concurrency and complex behaviour found in orchestrations make them susceptible to programming errors. Widely adopted standards such as the Web Service Description Language (WSDL) [5] provide support for syntactical compatibility analysis by defining message types in a standard way. However, WSDL defines one-way or request-response exchange patterns and does not support the definition of

more complex behaviour. Relevant behavioural information is exchanged between participants in human-readable forms, if at all. Automated verification of behavioural compatibility is impossible in such cases.

In the present paper, we address the problem of behavioural compatibility of web services by using a session based approach. Indeed, the session paradigm is now an active area of research with potential to improve the quality and correctness of software. The present paper is an exercise in the application of the session paradigm that illustrates some of its benefits. To that end, we choose to adapt and sessionize a significant subset of the industry standard orchestration language BPEL [17]. SeB (Sessionized BPEL) supports the same basic constructs as BPEL, but being a proof of concept, it does not include the non basic BPEL constructs such as, for example, exception handling. These differences are explained in more detail in section 2. On the other hand, SeB extends BPEL by featuring sessions as first class citizens. Sessions are typed in order to describe not only syntactical information but also behaviour. With SeB, a service exposes its required and provided session types. A client wishing to begin an interaction with a service first opens a session with the service. The type of this session defines the type and structure of possible interactions.

We provide an algorithm that verifies if a service written in SeB is well-typed. A well-typed service is one that correctly implements its declared required and provided session types. We also provide an algorithm that determines whether or not a collection of services are able to interact correctly by verifying the compatibility of the client's required session type with the provider's provided session type. Finally, we can prove that a well typed collection of interacting services is interaction-safe, meaning that no unexpected messages or arguments are exchanged.

The rest of this paper is organised as follows. Section 2 provides an informal introduction to the SeB language and contrasts its features with those of BPEL. Sections 3 and 4 give the syntax and semantics of untyped SeB. These sections are self contained manner and do not require any previous knowledge of BPEL. Section 5 presents the semantics of networked service configurations described in SeB. Section 6 introduces session types and typed SeB. It includes a typing algorithm and discusses the properties of well typed configurations. Relevant related work is surveyed in section 7 and the paper is concluded in section 8.

2 Informal introduction to SeB

Session initiation. The main novelty in SeB, compared to BPEL, is the addition of the session initiation, a new kind of atomic activity, and the way sessions impact the invoke and receive activities. The following is a typical sequence of three SeB atomic activities that can be performed by a client (we use a simplified syntax): $s@p$; $s!op_1(x)$; $s?op_2(y)$. This sequence starts by a session initiation activity, $s@p$ where s is a session variable and p a service location variable. The execution of $s@p$ by the client and by the target service (the one whose address is stored in p) have the following effects: (i) a fresh session id is stored in s, (ii) a new service instance is created at the service side and is dedicated to interact

with the client, (iii) another fresh session id is created on the service instance side and is bound to the one stored in s. The second activity, $s!op_1(x)$, is the sending of an invocation operation, op_1, with argument x. The invocation is sent precisely to this newly created service instance. The third activity of the sequence, $s?op_2(y)$, is the reception of an invocation operation op_2 with argument y that comes from this same service instance. Note that invocation messages are all one way and asynchronous: SeB does not provide for synchronous invocation. Furthermore, SeB does not make use of correlation sets, as BPEL does, to designate the instance that is targeted by a message. Instead, it is the session id that plays this role, as illustrated in the above example where the session variable s is systematically indicated in the invoke and receive activities. Moreover, sessions involve two and only two partners and any message sent by one partner over a session is targeted at the other partner. Biparty sessions are less powerful than correlation sets and in particular they do not allow for complex choreography configurations. At the end of the paper, we will give some ideas as to how this limitation can be lifted.

Structured activities. SeB inherits the principal structured activities of BPEL, i.e., flow, sequence and pic. It also inherits the possibility of having links between different subactivities contained in a flow, as well as adding a join condition to any activity. As in BPEL, a join condition requires that all its arguments have a defined value (true or false) and must evaluate to true in order for the activity to be executable. SeB also implements the so-called dead path elimination whereby all links outgoing from a canceled activity, or from its subactivities, have their values set to false.

Sequential Computations. Given that SeB is a language designed as a proof of concept, we wished to limit its main features to interaction behaviour. Hence, sequential computation and branching are not part of the language. Instead, they are assumed to be performed by external services that can be called upon as part of the orchestration. This approach is similar to languages like Orc [9] where the focus is on providing the minimal constructs that allow one to perform service orchestration functions and where sequential computation and boolean tests are provided by external *sites*. In particular, the original *do until* iteration of BPEL is replaced in SeB by a structured activity called "repeat", given by the syntax: $(do\ \mathsf{pic}_1\ until\ \mathsf{pic}_2)$. The informal meaning of repeat is: perform pic_1 repeatedly until the arrival of an invocation message awaited for in pic_2.

3 Syntax of SeB

3.1 Basic Sets

SeB assumes four categories of basic sets: values, variables, type identifiers and others. They are introduced in the following tables where, for each set, a short description is provided as are the names of typical elements. All the sets are pairwise disjoint unless otherwise stated.

Some explanations are in order: (i) the set $ExVal$ of exchangeable values contains the set $SrvVal$ of service locations. As it will be shown later, in SeB

Values

Set	*Description*	*Ranged over by*
$DatVal$	Data Values	$u, u', u_i, \cdots$
$SrvVal$	Service Locations	$\pi, \pi', \pi_i \cdots$
$ExVal = DatVal \uplus SrvVal$	Exchangeable Values	$w, w', w_i, \cdots$
$SesVal$	Session ids	$\alpha, \alpha', \alpha_i \cdots \beta \cdots$
$Val = ExVal \uplus SesVal$	All Values	$v, v', v_i, \cdots$
$LocVal = SrvVal \uplus SesVal$	All Locations	$\delta, \delta', \delta_i, \cdots$

Variables

Set	*Description*	*Ranged over by*
$DatVar$	Data Variables	y
$SrvVar$	Service Location Variables	$p_0, p, p', p_i \cdots q \cdots$
$ExVar = DatVar \uplus SrvVar$	Variables of Exchangeable Values	$x, x', x_i \cdots$
$SesVar$	Session Variables	$s_0, s, s', s_i \cdots r \cdots$
$Var = ExVar \uplus SesVar$	All Variables	$z, z', z_i \cdots$

Type Identifiers

Set	*Description*	*Ranged over by*
$DatTyp$	Data Types	$t, t', t_i, \cdots$
$SrvTyp$	Service Types	$P, P', P_i \cdots$
$ExTyp = DatType \uplus SrvTyp$	Types of Exchangeable Values	$X, X', X_i \cdots Y$
$SesTyp \supset SrvTyp$	Session Types	$T, T', T_i \cdots P, P', P_i$
$Typ = ExType \cup SesTyp$	All Types	$Z, Z', Z_i \cdots$

Others

Set	*Description*	*Ranged over by*
Op	Operation Names	$op, op', op_i \cdots$
Lnk	Control Links	$l, l', l_i \cdots$
Exp	Join Conditions	$e, e', e_i \cdots f \cdots$

Table 1. Basic Sets

services may dynamically learn about the existence of other services and may interact with dynamically discovered services; (ii) the set, $LocVal$, of all locations contains the set of session ids. Hence, session ids also play the role of locations for sending and receiving invocation messages. (iii) p_0 is a distinguished variable name dedicated to hold a service's own location (similar to *self*); (iv) s_0 is a session variable dedicated to hold a service's root session id, i.e., the session id handled by a service instance that is created as a response to a session initiation request from a client. The use of p_0 and s_0 will be described in detail later on in the paper.

3.2 Syntax of Activities

SeB being a dialect of BPEL, XML would be the most appropriate metalanguage for encoding its syntax. However, for the purpose of this paper, we have adopted a syntax based on records (à la Cardelli and Mitchell [3]) as it is better suited for discussing the formal semantics and properties of the language. By virtue of this

syntax, all SeB activities, except **nil**, are records having the following predefined fields: KND which identifies the kind of the activity, BEH, which gives its behaviour, SRC (respectively TGT), which contains the set of control links for which the activity is the source (respectively target), JCD which contains the join condition, i.e., a boolean expression over control link names (those given in field TGT). Moreover, the *flow* activity has an extra field, LNK, which contains the set of links that can be used by the subactivities contained in this activity. Field names are also used to extract the content of a field from an activity, e.g., if act is an activity, then act.BEH yields its behaviour. For example: a *flow* activity is given by the record $\langle \text{KND} = \textbf{FLO}, \text{BEH} = \textit{my\_behaviour},\ \text{SRC} = L_{\text{S}}, \text{TGT} = L_{\text{T}}, \text{JCD} = e, \text{LNK} = L\rangle$ where L_{S}, L_{T} and L are sets of control link names, and e is a boolean expression over link names. Finally, for the sake of conciseness, we will drop field names in records and instead we will associate a fixed position to each field. Hence, the *flow* activity given above becomes: $\langle \textbf{FLO}, \textit{my\_behaviour},\ L_{\text{S}}, L_{\text{T}}, e, L\rangle$.

We let $\mathcal{ACT}$ be the set of all activities and act a running element of $\mathcal{ACT}$, the syntax of activities is given in the following table:

act ::=	**nil**	(* nil activity *)
	ses | inv | rec	(* atomic activities *)
	seq | flo | pic | rep	(* structured activities *)

ses ::=	$\langle \textbf{SES}, s@p,\ L_{\text{S}},\ L_{\text{T}},\ e\rangle$	(* session initiation *)
inv ::=	$\langle \textbf{INV}, s!op(x_1, \cdots, x_n),\ L_{\text{S}},\ L_{\text{T}},\ e\rangle$	(* invocation *)
rec ::=	$\langle \textbf{REC}, s?op(x_1, \cdots, x_n),\ L_{\text{S}},\ L_{\text{T}},\ e\rangle$	(* reception *)
seq ::=	$\langle \textbf{SEQ}, (\mathsf{act}_1; \cdots; \mathsf{act}_n),\ L_{\text{S}},\ L_{\text{T}},\ e\rangle$	(* sequence *)
flo ::=	$\langle \textbf{FLO}, (\mathsf{act}_1 \mid \cdots \mid \mathsf{act}_n),\ L_{\text{S}},\ L_{\text{T}},\ e, L\rangle$	(* flow *)
pic ::=	$\langle \textbf{PIC}, (\mathsf{rec}_1; \mathsf{act}_1) + \cdots + (\mathsf{rec}_n; \mathsf{act}_n),\ L_{\text{S}},\ L_{\text{T}},\ e\rangle$	(* pick *)
rep ::=	$\langle \textbf{REP}, (\textit{do}\ \mathsf{pic}_1\ \textit{until}\ \mathsf{pic}_2),\ L_{\text{S}},\ L_{\text{T}},\ e\rangle$	(* repeat *)

Note that in the production rule for flo, "|" is to be considered just as a token separator. It is preferred over comma because it is more visual and better conveys the intended intuitive meaning of the flo activity being the container of a set of sub activities that run in parallel. The same remark applies to symbols, ";", "+", " *do*" and " *until*", which are used as token separators in the production rules for seq, pic and rep to convey their appropriate intuitive meanings.

Subactivities For an activity act, $\widehat{\mathsf{act}}$ is the set of all subactivities transitively contained in act: $\widehat{\mathsf{act}} \triangleq \{\mathsf{act}\} \cup \widehat{\mathsf{act.BEH}}$.

3.3 Well structured activities

A SeB activity act_0 is well structured iff the control links occuring in any activity of $\widehat{\mathsf{act}_0}$ satisfy the unicity, scoping and non cyclicity conditions given below.

Control links unicity
Given any control link l, and any pair of activities act and act':
if ($l \in$ act.LNK $\cap$ act'.LNK) or ($l \in$ act.SRC $\cap$ act'.SRC) or ($l \in$ act.TGT $\cap$ act'.TGT) then act = act'

Control links scoping
if $l \in \mathsf{act}.\text{SRC}$ (respectively if $l \in \mathsf{act}.\text{TGT}$) then $\exists\ \mathsf{act'}, \mathsf{act''}$ with $\mathsf{act} \in \widehat{\mathsf{act''}}$ and $\mathsf{act'} \in \widehat{\mathsf{act''}}$ and with $l\ \in \mathsf{act''}.\text{LNK}$ and $l\ \in \mathsf{act'}.\text{TGT}$. (respectively $l\ \in \mathsf{act'}.\text{SRC}$).

Control links non cyclicity
Relation *pred* defined by: act *pred* act' iff $\mathsf{act}.\text{SRC} \cap \mathsf{act'}.\text{TGT} \neq \emptyset$, is acyclic.

4 Semantics of SeB

The semantics of SeB will be given in two steps. First, we show how SeB activities translate into control graphs, then we use control graphs to provide the semantics of networked services. In this section, we start by presenting control graphs and then provide the SOS rules that define a translation of well structured activities, then we present configurations of networked services and provide the reduction rules defining their operational semantics.

4.1 Control Graphs

Observable Actions The set ACTIONS of *observable* actions is defined by:
$\mathsf{ACTIONS} =_{\text{def}} \{\ a \mid a$ is any action of the form: $s@p,\ s!op(\tilde{x})$ or $s?op(\tilde{x})\ \}$

All actions We define the set $\mathsf{ACTIONS}_\tau$ of all actions (ranged over by σ):
$\mathsf{ACTIONS}_\tau =_{\text{def}} \mathsf{ACTIONS} \cup \{\tau\}$ where τ denotes the unobservable action.

Control Graphs A control graph, Γ, is a labeled transition system with the following structure: $\Gamma =< G, g_0, \mathcal{A}, \rightarrow >$ where
- G is a set of states, called control states
- g_0 is the initial control state
- $\mathcal{A}$ is a set of actions ($\mathcal{A} \subset \mathsf{ACTIONS}_\tau$)
- $\rightarrow \subset\ G$ x $\mathcal{A}$ x G

4.2 Semantics of activities

Control Links Maps: A Control link map c is a partial function from control links to booleans extended with the undefined value. $c : Lnk \rightarrow \{\mathbf{true}, \mathbf{false}, \bot\}$

Initial Control Links Map: For an activity act we define c_{act}, the initial control links map: $dom(c_{\mathsf{act}}) = \{\ l \mid l$ occurs in $\widehat{\mathsf{act}}\}$ and $\forall l \in dom(c_{\mathsf{act}}) : c_{\mathsf{act}}(l) = \bot$

Evaluation of a join condition: If L is a set of control links, e a boolean expression over L and c a control links map, then the evaluation of e in the context of c is written: $c \triangleright e(L)$. Furthermore we consider that this evaluation is defined only when $\forall l \in L, c(l) \neq \bot$.

Control states of activities: A couple (c, act) is said to be a valid activity control state iff for any control link, l, occuring in $\widehat{\mathsf{act}}$: $l \in dom(c)$.

In table 2, we provide the SOS rules defining a translation from activities to control graphs. The rules for the seq activity have been skipped as for any seq one can construct a behaviourally equivalent flo activity that defines the same

ordered list of subactivities by defining control links between consecutive activities. The rules for the repeat rep are given by first substituting the behaviour part of the rep record, (do pic_1 $until$ pic_2), with a triple noted (pic_1 [pic_1>pic_2) where the second occurence of pic_1 encodes the current state of the repeat, while the first occurence is the activity to be repeated when the current state reaches activity **nil**. Also, in order to have an escape from the repeat, a static rule enforces that pic_2 is not equal to **nil**. Finally, in activity flo we dropped field LNK since its value is constant (LNK is used to define a scope for control link variables).

The notation for value substitution in control link maps used in the rules of Table 2 needs an explanation: $c[^{\mathbf{true}}/_{l}] =_{def} c'$ where $c'(l') = c(l)$ for $l \neq l'$ and $c'(l) = \mathbf{true}$. By abuse of notation, we also apply value substitution to sets of control links. Hence, if Π is a set of activities, then, e.g., $c[^{\mathbf{false}}/\widehat{\Pi}.\text{SRC}]$ is the subsitution whereby any control link occuring as source of an activity in $\widehat{\Pi}$ has its value set to **false**.

$$\boxed{\text{SES}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true}}{(c, \langle \mathbf{SES}, s@p, L_{\text{T}}, L_{\text{S}}, e\rangle) \downarrow s@p (c[^{\mathbf{true}}/_{L_{\text{S}}}], \mathbf{nil})}$$

$$\boxed{\text{INV}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true}}{(c, \langle \mathbf{INV}, s!op(\tilde{x}), L_{\text{T}}, L_{\text{S}}, e\rangle) \downarrow s!op(\tilde{x}) (c[^{\mathbf{true}}/_{L_{\text{S}}}], \mathbf{nil})}$$

$$\boxed{\text{REC}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true}}{(c, \langle \mathbf{REC}, s?op(\tilde{x}), L_{\text{T}}, L_{\text{S}}, e\rangle) \downarrow s?op(\tilde{x}) (c[^{\mathbf{true}}/_{L_{\text{S}}}], \mathbf{nil})}$$

$$\boxed{\text{FLO1}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true} \quad (c, \mathsf{act}_i) \xrightarrow{\sigma} (c', \mathsf{act}')}{(c, \langle \mathbf{FLO},\ \mathsf{act}_1|\cdots|\mathsf{act}_i|\cdots|\mathsf{act}_n\ ,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle) \downarrow\sigma (c', \langle \mathbf{FLO},\ \mathsf{act}_1|\cdots|\mathsf{act}'|\cdots|\mathsf{act}_n\ ,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle)}$$

$$\boxed{\text{FLO2}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true}}{(\ c,\ \langle \mathbf{FLO}, \mathsf{act}_1|\cdots|\mathbf{nil}|\cdots|\mathsf{act}_n\ ,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle\ ) \downarrow\tau (\ c',\ \langle \mathbf{FLO},\ \mathsf{act}_1|\cdots|\cdots|\mathsf{act}_n\ ,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle\ )}$$

$$\boxed{\text{FLO3}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true}}{(c, \langle \mathbf{FLO},\ \mathbf{nil},\ L_{\text{T}}, L_{\text{S}},\ e\rangle) \xrightarrow{\tau} (c[^{\mathbf{true}}/_{L_{\text{S}}}], \mathbf{nil})}$$

$$\boxed{\text{PIC}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true} \quad (c, \mathsf{rec}) \xrightarrow{\sigma} (c', \mathbf{nil})}{(\ c,\ \langle \mathbf{PIC}, (\mathsf{rec}; \mathsf{act}) + \Pi\ ,\ L_{\text{T}},\ L_{\text{S}},\ e\ \rangle\ ) \downarrow\sigma (\ c'',\ \langle \mathbf{FLO},\ \mathsf{act}\ ,\ L_{\text{T}},\ L_{\text{S}},\ e\ \rangle\ )}$$

where $\Pi = \sum(\mathsf{rec}_i\ ;\ \mathsf{act}_i)$

and $c'' = c'[^{\mathbf{false}}/\widehat{\Pi}.\text{SRC}]$

$$\boxed{\text{REP1}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true} \quad (c, \mathsf{act}) \xrightarrow{\sigma} (c', \mathsf{act}')}{(c,\ \langle \mathbf{REP}\ ,\ \mathsf{pic}_1[\,\mathsf{act}>\mathsf{pic}_2,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle\ ) \downarrow\sigma (c',\ \langle \mathbf{REP}\ ,\ \mathsf{pic}_1[\,\mathsf{act}'>\mathsf{pic}_2,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle\ )}$$

$$\boxed{\text{REP2}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true} \quad (c, \mathsf{pic}_2) \xrightarrow{\sigma} (c', \mathsf{act}')}{(c,\ \langle \mathbf{REP}\ ,\ \mathsf{pic}_1[\,\mathsf{pic}_1>\mathsf{pic}_2,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle\ ) \downarrow\sigma (c',\ \langle \mathbf{FLO}\ ,\ \mathsf{act}'\ ,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle\ )}$$

$$\boxed{\text{REP3}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{true} \quad \mathsf{pic}_1 \neq \mathbf{nil}}{(c,\ \langle \mathbf{REP}\ ,\ \mathsf{pic}_1[\,\mathbf{nil}>\mathsf{pic}_2,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle\ ) \downarrow\tau (c',\ \langle \mathbf{REP}\ ,\ \mathsf{pic}_1[\,\mathsf{pic}_1>\mathsf{pic}_2,\ L_{\text{T}},\ L_{\text{S}},\ e\rangle\ )}$$

where $c' = c[^{\perp}/_{\widehat{\mathsf{pic}_1}.\text{SRC}}, ^{\perp}/_{\widehat{\mathsf{pic}_1}.\text{TGT}}]$

$$\boxed{\text{DPE}}\ \frac{c \triangleright e(L_{\text{T}}) = \mathbf{false}}{(c, \langle *,\ \mathsf{act},\ L_{\text{T}},\ L_{\text{S}},\ e\rangle) \downarrow\tau (c[^{\mathbf{false}}/_{\widehat{\mathsf{act}}.\text{SRC}}],\ \mathbf{nil}\ )}$$

Table 2. SOS Rules for Activities

Rules priorities. On the ground SOS rules, i.e., those having no transitions in their premise, we define a priority order as follows: FLO2 > FLO3 > REP3 > DPE > SES = INV > REC. This means that when two transitions are possible from a given state, each deriving from a ground rule such that the two corresponding ground rules have differing priorities, then the one having a lower priority is pruned. We will discuss the properties of control graphs obtained from such prioritised SOS rules in the a subsequent section.

4.3 Control Graphs of SeB Activities

When applied to the initial control state $(c_{\mathsf{act}}, \mathsf{act})$ of a well structured activity act, the SOS rules with priorities defined above yield a control graph that we note $\mathbf{cg}(\mathsf{act})$.

Properties of Control Graphs of Activities The following properties can be proven about control graphs of activities: In $\mathbf{cg}(\mathsf{act})$, the set of control states is finite and is partitioned into four categories: *silent* states, *receiving* states, *transient* states, and *terminal* states. All terminal states are of the form (c, **nil**). Hence they differ only by their control link map. The transitions leaving a *silent* state are all labeled with the silent action τ. The transitions leaving a *receiving* state are all labeled with reception actions, while the transitions leaving a *transient* state are labeled with either invocation or session initiation actions. Moreover, the silent τ transitions are confluent since no two τ transitions can be mutually conflicting (in fact, the same applies to transient transitions). Hence, using observation equivalence minimisation, a control graph can be reduced to a minimal control graph with no τ actions and one and only terminal state (because all terminal states are observationally equivalent). The graph resulting of such a reduction is said to verify the *run to completion* property. This property implies the following behaviour of control graphs: when a message is received, it is only after all other possible transient actions have taken place that another message can be received.
Henceforth, when we write $\mathbf{cg}(\mathsf{act})$ we will consider that we are dealing with the minimised control graph, and we will name its unique terminal state $\mathbf{term}(\mathsf{act})$.

Considering a well structured activity act, we will adopt the following notations: $\mathbf{init}(\mathsf{act})$ denotes the initial state of $\mathbf{cg}(\mathsf{act})$; $\mathbf{states}(\mathsf{act})$ is the set of control states of $\mathbf{cg}(\mathsf{act})$; $\mathbf{trans}(\mathsf{act})$ is the set of transitions of $\mathbf{cg}(\mathsf{act})$. For $g \in \mathbf{states}(\mathsf{act})$ and $g' \in \mathbf{states}(\mathsf{act})$: $g \xrightarrow[\mathsf{act}]{\sigma} g' \Leftrightarrow (g, \sigma, g') \in \mathbf{trans}(\mathsf{act})$

Open for reception. A state, g, of $\mathbf{cg}(\mathsf{act})$ is said to be open on session s and we note $open(\mathsf{act}, g, s)$, iff state g has at least one outgoing transition labeled with a receive action on session s. More formally: $open(\mathsf{pic}, g, s) =_{def} \exists\ op,\ x_1\ \ldots\ x_n,\ g'$ such that $g \xrightarrow[\mathsf{act}]{s?op(x_1,\cdots,x_n)} g'$

4.4 Free, bound, usage and forbidden occurrences of variables

Thanks to control graphs of (well structured) activities, we can define the notions of free, usage, bound and forbidden occurences of variables. For an activity act we define the set of variables occurring in act: $V(\mathsf{act}) =_{\text{def}} \{\ z \mid z \text{ occurs in } \widehat{\mathsf{act}}\ \}$

Binding occurrences. For variables $y \in V(\mathsf{act})$, $s \in V(\mathsf{act})$ and $p \in V(\mathsf{act})$, the following underlined occurrences are said to be binding occurrences in act: $\underline{s}@p$, $s?op(\cdots, \underline{y}, \cdots)$ and $s?op(\cdots, \underline{p}, \cdots)$. We denote $BV(\mathsf{act})$ the set of variables having a binding occurrence in act.

Usage occurrences. For variables $y \in V(\mathsf{act})$, $s \in V(\mathsf{act})$ and $p \in V(\mathsf{act})$, the following underlined occurrences are said to be usage occurrences in act: $s@\underline{p}$, $\underline{s}?op(\cdots)$, $\underline{s}!op(\cdots)$, $s!op(\cdots, \underline{p}, \cdots)$ and $s!op(\cdots, \underline{y}, \cdots)$. We denote $UV(\mathsf{act})$ the set of variables having at least one usage occurrence in act.

Free occurrences. A variable $z \in V(\mathsf{act})$ is said to occur free in act, iff there is a path in $\mathbf{cg}(\mathsf{act})$: $g_0 \xrightarrow[\mathsf{act}]{\sigma_1} g_1, \cdots, g_{n-1} \xrightarrow[\mathsf{act}]{\sigma_n} g_n$ where z has a usage occurrence in σ_n and has no binding occurrence in any of $\sigma_1, \cdots, \sigma_{n-1}$. We denote $FV(\mathsf{act})$ the set of variables having at least one free occurrence in act.

Forbidden occurences. $op?(\cdots p_0 \cdots)$ and $s_0@p$ are forbidden occurences. As we shall explain later, p_0 is reserved for the own location of the service, while s_0 is a reserved session variable that receives a session id implicitly at service instantiation time.

5 Syntax and Semantics of Networked Services

5.1 Service Configurations

Let m be a partial map from variables Var to $Val \cup \{\bot\}$, the set of values augmented with undefined value. Henceforth, we consider couples (m, act) where $dom(m) = V(\mathsf{act})$.

Deployable services. The couple (m, pic) is a deployable service iff:
- $m(p_0) \neq \bot$ (the service has a defined location address recorded in p_0),
- $\mathsf{pic}.\text{BEH} = \sum s_0?op_i(\tilde{x}_i)\ ;\ \mathsf{act}_i$ (pic has all its receive actions on session s_0),
- $FV(\mathsf{act}) \cap SesVar = \{s_0\}$ (s_0 is the only free session variable),
- $\forall z \in FV(\mathsf{act}) \setminus \{s_0\} : m(z) \neq \bot$ (free variables, except s_0, have defined values)

Running service instances. Informally, a deployed service behaves like a factory creating a new running service instance each time it receives a session initiation request. The initial state of the instance is given by a triple $(m[^{\beta}/_{s_0}], \mathsf{pic} \triangleright \mathbf{init}(\mathsf{pic}))$ where s_0 has received an initial value, β, a freshly created session id, and where $\mathbf{init}(\mathsf{pic})$ is the initial control state. The new instance will then run and interact with the client that initiated the session and with other services that it *orchestrates*. The running state of a service instance derived from the deployable service (m, pic) is the triple $(m', \mathsf{pic} \triangleright g)$ where $g \in \mathbf{states}(\mathsf{pic})$ is the current control state of the instance and m', with $dom(m') = dom(m)$, is its current map.

Deployable clients. We can define similarly the concepts of deployable "pure" client and client instance. A deployable client is a couple (m, act) where $\mathsf{act}.\text{BEH} =$

$s@p;\ s!op(x_1, \cdots, x_n);\ \mathsf{act}'$ where s is the only session variable occurring in act and where $s@p$ is the only session initiation action present in $\widehat{\mathsf{act}}$. The deployment of a deployable client (m, act) yields the running client instance $(m, \mathsf{act} \blacktriangleright \mathbf{init}(\mathsf{act}))$.

Well partnered sets of services. A set, $\{(m_1, \mathsf{pic}_1), \cdots, (m_k, \mathsf{pic}_k)\}$, of deployable services is said to be well partnered iff:

- $\forall i, j : i \neq j \Rightarrow m_i(p_0) \neq m_j(p_0)$ (any two services have different location addresses),
- $\forall i, p : m_i(p) \neq \perp \Rightarrow \exists j$ with $m_i(p) = m_j(p_0)$ (any partner required by one service is present in the set of services).

Running configurations. A running configuration, C, is a collection made of a well partnered set of services, a set of service instances and a set of client instances, all running in parallel. We use the symbol $\diamond$ to denote the associative and commutative parallel operator:

$$\begin{array}{lll} C ::= & (m, \mathsf{pic}) & (* \text{ service } *) \\ \mid & (m, \mathsf{pic} \blacktriangleright g) & (* \text{ service instance } *) \\ \mid & (m, \mathsf{act} \blacktriangleright g) & (* \text{ client instance } *) \\ \mid & C \diamond C & (* \text{ running configuration } *) \end{array}$$

Networked configurations. A networked configuration is a triple $\lfloor\!\lfloor\ C\ \rceil\!\rfloor\ \lfloor\!\lfloor\ Q\ \rceil\!\rfloor\ \lfloor\!\lfloor\ \mathcal{B}\ \rceil\!\rfloor$ where C, is a running configuration, Q is a set of message queues and $\mathcal{B}$ a set of session bindings. Q and $\mathcal{B}$ are introduced hereafter.

Message queues. Q is a set made of message queues with $Q ::= q \mid q \diamond Q$, where q is an individual FIFO message queue of the form $q ::= \delta \hookleftarrow \tilde{M}$ with $\tilde{M}$ a possibly empty list of ordered messages and δ the destination of the messages in the queue. The contents of $\tilde{M}$ depend on the destination type. If δ is a service location, $\tilde{M}$ contains only session initiation requests of the form $new(\alpha)$. However if δ is a session id, then $\tilde{M}$ contains only operation messages of the form $op(\tilde{w})$.

Session bindings. A session binding is an unordered pair of session ids (α, β). A running set of session bindings is noted $\mathcal{B}$ and has the syntax $\mathcal{B} ::= (\alpha, \beta) \mid (\alpha, \beta) \diamond \mathcal{B}$. If $(\alpha, \beta) \in \mathcal{B}$ then α and β are said to be bound and messages sent on local session id α are routed to a partner holding local session id β, and vice-versa.

5.2 Semantics of Networked Services

$$\frac{\boxed{\text{SES1}} \qquad g \xrightarrow[\mathsf{act}]{s@p} g' \qquad m(p) = \pi}{\begin{array}{l} \lfloor\!\lfloor\ \cdots\ (m, \mathsf{act} \blacktriangleright g)\ \cdots \rceil\!\rfloor \cdots (\pi \hookleftarrow \widetilde{M}) \cdots \rceil\!\rfloor\ \mathcal{B}\ \rceil\!\rfloor \longrightarrow \\ \lfloor\!\lfloor\ \cdots\ (m[^{\alpha}/_{s}], \mathsf{act} \blacktriangleright g')\ \cdots\ \rceil\!\rfloor \cdots (\pi \hookleftarrow \widetilde{M} \cdot new(\beta)\ ) \cdots \rceil\!\rfloor\ (\alpha, \beta) \diamond \mathcal{B}\ \rceil\!\rfloor \end{array}}\ \alpha, \beta \text{ fresh}$$

$$\frac{\boxed{\text{SES2}} \qquad m(p_0) = \pi}{\begin{array}{l} \lfloor\!\lfloor\ \cdots\ (m, \mathsf{pic})\ \cdots \rceil\!\rfloor \quad \cdots (\pi \hookleftarrow new(\beta) \cdot \widetilde{M})\ \cdots \rceil\!\rfloor\ \mathcal{B}\ \rceil\!\rfloor \longrightarrow \\ \lfloor\!\lfloor \cdots (m, \mathsf{pic}) \diamond (m[^{\beta}/_{s_0}], \mathsf{pic} \blacktriangleright g_0) \cdots \rceil\!\rfloor \cdots (\pi \hookleftarrow \widetilde{M}) \cdots \rceil\!\rfloor\ \mathcal{B}\ \rceil\!\rfloor \end{array}}\ g_0 = \text{init}(\mathsf{pic})$$

$$\frac{\boxed{\text{INV}} \qquad g \xrightarrow[\text{act}]{s!op(x_1,\cdots,x_n)} g' \qquad m(s)=\alpha \qquad (\alpha,\beta)\in\mathcal{B}}{\begin{array}{l}\lfloor\ \cdots\ (m,\mathsf{act}\blacktriangleright g)\ \cdots\ \rceil\!\lfloor\cdots(\beta \hookleftarrow \widetilde{M})\cdots\ \rceil\!\lfloor\ \mathcal{B}\ \rceil \longrightarrow \\ \lfloor\ \cdots\ (m,\mathsf{act}\blacktriangleright g')\ \cdots\ \rceil\!\lfloor\cdots\ (\ \beta \hookleftarrow \widetilde{M}\cdot op(m(x_1),\cdots,m(x_n))\ )\ \cdots\ \rceil\!\lfloor\ \mathcal{B}\ \rceil\end{array}}$$

$$\frac{\boxed{\text{REC}} \qquad g \xrightarrow[\text{act}]{s?op(x_1,\cdots,x_n)} g' \qquad m(s)=\beta}{\begin{array}{l}\lfloor\ \cdots\ (m,\mathsf{act}\blacktriangleright g)\ \cdots\rceil\!\lfloor\cdots(\beta \hookleftarrow op(w_1,\cdots,w_n)\cdot\widetilde{M})\cdots\ \rceil\!\lfloor\ \mathcal{B}\ \rceil \longrightarrow \\ \lfloor\ \cdots\ (m[{}^{w_1}\!/_{x_1},\cdots,{}^{w_n}\!/_{x_n}],\mathsf{act}\blacktriangleright g')\ \cdots\ \rceil\!\lfloor\cdots(\beta \hookleftarrow \widetilde{M})\ \cdots\ \rceil\!\lfloor\ \mathcal{B}\ \rceil\end{array}}$$

6 Typed SeB

6.1 Session Types

Action types. An action type, η, is either a send action type, written $\eta = !op(X_1,\cdots,X_n)$, or a receive action type, $\eta = ?op(X_1,\cdots,X_n)$, where X_i is either a session type name, T, or a data type identifier, t. We let ACTION_TYPES denote the set of all action types, and will adopt the notation $\tilde{X}$ to denote a vector of dimension 0 or more: $X_1,\cdots,X_n$.

Syntax of session types. We let ST run over session types. ST is as follows:

$$\begin{array}{rcll} ST ::= & \triangle & \text{(* Terminal State *)} \\ \mid & T, P & \text{(* Session Type name *)} \\ \mid & \sum_I\ ?op_i(\tilde{X}_i);\ T_i & \text{(* Input Choice *)} \\ \mid & \oplus_I\ !op_i(\tilde{X}_i);\ T_i & \text{(* Output Choice *)} \\ \mid & \bigtriangledown & \text{(* Error State *)} \end{array}$$

where we assume a set of defining equations associating names to session types: $E = \{T_1 = ST_1,\ \cdots\ , T_n = ST_n\}$ and we adopt the notation $E(T_i) = ST_i$. Moreover, we assume that the operations, op_i, appearing in the input and output choice are all different, i.e., session types are deterministic.

Semantics of Session Types. The semantics of session types is given through a translation to labelled transition systems with labels in ACTION_TYPES, and defined with the 3 SOS rules below. Moreover, we will consider this labelled transition system endowed with its natural bisimulation relation, noted $\equiv$.

$$\frac{}{\oplus_I !op_i(\tilde{X}_i);\ T_i \xrightarrow{!op_i(\tilde{X}_i)} T_i}\ \textit{Send} \qquad \frac{}{\sum_I ?op_i(\tilde{X}_i);\ T_i \xrightarrow{?op_i(\tilde{X}_i)} T_i}\ \textit{Receive}$$

$$\frac{E(T)=ST,\ ST \xrightarrow{\eta} ST',\ \ E(T')=ST'}{T \xrightarrow{\eta} T'}\ \textit{Session Type Name}$$

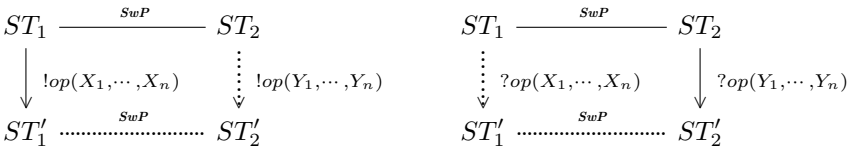
Service Type. The session type ST is said to be a service type iff the following three conditions are met: (i) $ST = \sum_I ?op_i(\tilde{X}_i);\ T_i$, (ii) $\triangle$ is a sink state of ST and (iii) $\bigtriangledown$ is not a sink state of ST. We will use P to run over service type names, i.e., $E(P) = ST$ where ST is a service type.

Dual Session Types. For a session type, ST, its dual $\overline{ST}$ is obtained by swaping sends and receives. The dual of a session type name, T, is another name $\overline{T}$ with $E(\overline{T}) = \overline{E(T)}$. Of course we have $\overline{\overline{ST}} = ST$.

Client Type. ST is said to be a client type iff its dual $\overline{ST}$ is a service type.

6.2 Subtyping with Progress

Subtyping relation. We need an adpated version of the classical subtyping relation and define subtyping with progress on session types as follows: $\mathcal{SwP}$ is a subtyping with progress relation iff for all ST_1 and ST_2: (i) $(ST_1\ \mathcal{SwP}\ ST_2) \Rightarrow (ST_1 \equiv \triangle \Leftrightarrow ST_2 \equiv \triangle)$ and (ii) the following two diagram conditions hold where continuous lines and arrows represent the "if exists" part and dotted lines and arrows represent the "then exists" part:



In the above digrams, we consider that if X_i and Y_i are data types, then $X_i\ \mathcal{SwP}\ Y_i \Leftrightarrow X_i = Y_i$. A session type ST_1 is said to be a sub-progress type of a session type $ST2$, written $ST_1 \preceq ST_2$, iff there exists a subtyping with progress relation, $\mathcal{SwP}$, such that $ST_1\ \mathcal{SwP}\ ST_2$.

6.3 Adding types to SeB

In this section we extend SeB with explicit types. We consider now the map $\hat{m}$ as composed of two maps: $\hat{m} = (\acute{m}, \grave{m})$, where $\acute{m}$ is the value map (previously noted m) and $\grave{m}$ the type map, mapping variables to types:

$\grave{m} : (DatVar \to DatTyp) \cup (SrvVar \to SrvTyp)) \cup (SesVar \to SesTyp \cup \{\bot\})$.

We need to redefine the notions of deployable service and of well partenered configuration.

Deployable typed service. A couple $(\hat{m}, \mathsf{pic})$ is a deployable typed service iff:

- $(\acute{m}, \mathsf{pic})$ is a deployable service,
- $dom(\grave{m}) = V(\mathsf{act})$ (all variables must have initial types),
- $\grave{m}(s_0) = \overline{\grave{m}(p_0)}$ (session variable s_0 is initiated with the dual type of the the service type),
- $\forall s \in V(\mathsf{act}) \backslash \{s_0\} : \grave{m}(s) = \bot$ (the initial type of all other session variables is $\bot$)

The initial running instance of a deployable typed service $(\hat{m}, \mathsf{pic})$ is given by the triple $(\acute{m}, \mathsf{pic} \blacktriangleright \mathbf{init}(\mathsf{pic}))$ and its well typedness is assessed by considering the typing part, i.e., the triple $(\grave{m}, \mathsf{pic} \blacktriangleright \mathbf{init}(\mathsf{pic}))$.

6.4 Well typedness

Notations for typing rules. Before tackling the typing rules, we need to introduce the following notations, where $\grave{m}$ is used as a typing environment (with $* \in \{!, ?\}$):

$\grave{m} \vdash z : Z \Leftrightarrow \grave{m}(z) = Z$ where $(z : Z)$ can be any of $\{(s : T), (y : t), (p : P), (x : X)\}$

$\grave{m} \vdash g \xrightarrow[\mathit{Types}\ \ \mathsf{act}]{s*op(X_1,\ldots,X_n)} g' \Leftrightarrow$
$\exists\ op, x_1, \ldots x_n$ with $\grave{m} \vdash x_1 : X_1, \ \ldots\ x_n : X_n$ and $g \xrightarrow[\mathsf{act}]{s*op(x_1,\ldots,x_n)} g'$

$\grave{m} \vdash g \xrightarrow[\mathit{Types}\ \ \mathsf{act}]{s*op(\tilde{X})} \Leftrightarrow \exists g'$ with $\grave{m} \vdash g \xrightarrow[\mathit{Types}\ \ \mathsf{act}]{s*op(\tilde{X})} g' \quad \grave{m} \vdash g \xrightarrow[\mathit{Types}\ \ \mathsf{act}]{s*op(\tilde{X})} \Leftrightarrow \neg(\grave{m} \vdash g \xrightarrow[\mathit{Types}\ \ \mathsf{act}]{s*op(\tilde{X})})$

Typing procedure for SeB We now consider a deployable typed service $(\hat{m}_0, \mathsf{pic})$. The SOS typing rules (provided in the appendix), when applied to this typed service starting from its initial state, $(\grave{m}_0, \mathsf{act} \blacktriangleright \mathbf{init}(\mathsf{pic}))$, define a translation into a finite non-labelled transition system, called the typing graph of $(\grave{m}_0, \mathsf{pic})$ and noted $\mathbf{tg}(\grave{m}_0, \mathsf{pic})$. A running state of $\mathbf{tg}(\grave{m}_0, \mathsf{pic})$ is given by a triple $(\grave{m}, \mathsf{act} \blacktriangleright g))$ where $\grave{m}$ is the current typing map and g the current control state of the service.

The typing rules explore the joint behaviour of the service and the session types whereby an input/output transition can be taken only if both the service and the coresponding session have matching transitions. In case of mismatch, the state of the session type is set to ∇, the error state. Also, the rules enforce that a session cannot be re-initiated if its current state is not $\triangle$, the terminal state. It is possible to define a strong and a weak typedeness, depending on whether we allow sessions other than s_0 to be stopped by the service before reaching the terminal state $\triangle$. We provide herafter the definition of the strong version.

Strong well typedeness. A typed service, $(\grave{m}_0, \mathsf{pic})$, is strongly well typed iff:

- $\forall(\grave{m}, \mathsf{pic} \blacktriangleright g) \in \mathbf{states}(\grave{m}_0, \mathsf{pic}), \forall s \in dom(\grave{m}_0) : \grave{m}_0(s) \neq \nabla$
- $\forall(\grave{m}, \mathsf{pic} \blacktriangleright g) \in \mathbf{states}(\grave{m}_0, \mathsf{pic})$: if $(\grave{m}, \mathsf{act} \blacktriangleright g)$ is a sink state of $\mathbf{tg}(\grave{m}_0, \mathsf{pic})$ then $g = \mathbf{term}(\mathsf{pic})$ and $\forall s \in dom(\grave{m}_0) : \grave{m}_0(s) \in \{\triangle, \bot\}$

Well typed collection of services A set $\{(\hat{m}_1, \mathsf{pic}_1), \cdots, (\hat{m}_k, \mathsf{pic}_k)\}$ of well partnered services is said to be a well typed collection iff:

- $\forall i, (\grave{m}_i, \mathsf{pic}_i)$ is strongly well typed,
- $\forall i, j, p : \acute{m}_i(p) = \acute{m}_j(p_0) \Rightarrow \grave{m}_j(p_0) \preceq \grave{m}_i(p)$ (each provided type is a subprogress type of a required type).

Property of strong well-typedness
In a running configuration made of a well typed collection of services, a service instance never blocks. More formally, considering a state, Ω, reached by a well typed collection, with $\Omega = \lfloor \cdots (\acute{m}, \mathsf{pic} \blacktriangleright g) \cdots \rceil\!\lfloor \cdots (\beta \hookleftarrow op(\tilde{w}) \cdot \widetilde{M}) \cdots \rceil\!\lfloor\ \mathcal{B}\ \rceil$ where $\acute{m}(s) = \beta$ and $\mathit{open}(\mathsf{pic}, g, s)$ then $\Omega \rightarrow \lfloor \cdots (\acute{m}', \mathsf{pic} \blacktriangleright g') \cdots \rceil\!\lfloor \cdots (\beta \hookleftarrow \cdot \widetilde{M}) \cdots \rceil\!\lfloor\ \mathcal{B}\ \rceil$

7 Related work

Behavioural types associated to sessions have been studied for protocols [7] and software components [19]. Service orchestration calculi including the notion of sessions have also been defined [1, 12], as well as session-based graphical orchestration languages [6]. Correlation is a different approach in which messages that are logically related are identified as sharing the same *correlation data* [13] as it occurs, for example, when a unique id related to a client is passed in any message referring to that client. Notably BPEL [17] features correlations sets, and we could have defined a "session oriented" style in BPEL using correlation sets. However, this approach would not lend itself easily to typing. On the other hand, BPEL correlation sets allow for multi-party choreographies to be defined. We argue that similar expressivity is attainable with session types by extending them to support multi-party sessions. This is a challenge for future work, particularly considering that another layer of complexity is added with the concept of multi-party session types [8, 2] . In both session-based and correlation based approaches, defining behavioural types has often proved difficult: while sessions make simpler, with respect to correlation approaches, the identification of interaction patterns that are to be typed, session based calculi with higher order session communication, defined in π-calculus style, have also been studied, but make typing non-trivial and difficult to support automatic verification [11, 16].

Session types usually take the form of finite-state automata, sometimes borrowing process algebra concepts. The distinction between external and internal choice corresponding to input and output messages is the major source of difficulty for determining compatibility between components or services [15, 4].

The work on BPEL described in [10, 18, 14] is most related to ours. In [10] the authors provide a full static semantics for data flow in BPEL, covering also xpath data types. In [18] the authors provide a full semantics of control flow of a comprehensive part of BPEL. Their approach is based on a translation on Petri Nets. We have shown that using a direct semantics based on records and SOS rules can be also elegant and concise. Perhaps the work that is closest is the one given in [14]. The authors present an interesting typing system on a process calculus inspired from BPEL, but it does not cover behavioural typing and does not tackle BPEL control links.

8 Conclusion

We have shown that one approach to verify the behavioural compatibility of web services is to introduce a behavioural typing system. In order to prove this concept, we have adapted and formalised a subset of the widely adopted orchestration language BPEL to support typed sessions. We call the resulting formalism SeB, for Sessionized BPEL. A typed session is used to prescribe the correct structure of an interaction between two partner services during the fulfilment of a service. A SeB service declares the session types that it can provide to prospective partners, while also declaring its required session types. Based on these declarations, we can verify whether or not a service is well-typed, hence answering the question of whether or not the service respects its required and provided types. We are also able to verify the compatibility of two sessions types,

which allows us to determine whether or not two partners that correctly implement their own declared required and provided sessions types are able to interact together.
When a set of interacting services are well typed in this way, we call it a well-typed service configuration. We have shown that a well-typed service configuration is interaction-safe. The formal approach taken with SeB as presented in this paper opens up the possibility of defining and proving other properties of web service interactions. These include but are not limited to controllability and progress properties, which we hope to tackle in future work.

References

1. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. A service centered calculus. In M. Bravetti, M. Nez, and G. Zavattaro, editors, *Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer Berlin / Heidelberg, 2006. 10.1007/11841197_3.
2. R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty sessions in soc. In D. Lea and G. Zavattaro, editors, *Coordination Models and Languages*, volume 5052 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-68265-3_5.
3. L. Cardelli and J. Mitchell. Operations on records. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 22–52. Springer Berlin / Heidelberg, 1990. 10.1007/BFb0040253.
4. G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundations of Session Types. In *PPDP'09*, pages 219–230. ACM Press, 2009. Full version:http://www.di.unito.it/ dezani/papers/cdgpFull.pdf.
5. R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Service Definition Language (WSDL) Version 2.0. Technical report, June 2007.
6. A. Fantechi and E. Najm. Session types for orchestration charts. In D. Lea and G. Zavattaro, editors, *Coordination Models and Languages*, volume 5052 of *Lecture Notes in Computer Science*, pages 117–134. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-68265-3_8.
7. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0053567.
8. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43:273–284, January 2008.
9. D. Kitchin, A. Quark, W. Cook, and J. Misra. The orc programming language. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02138-1_1.
10. O. Kopp, R. Khalaf, and F. Leymann. Deriving explicit data links in ws-bpel processes. In *IEEE SCC (2)*, pages 367–376, 2008.
11. R. P. A. R. L. Caires, G. Ferrari. Behavioural types for service composition. Technical report, Sensoria project, 2006.
12. I. Lanese, V. T. Vasconcelos, F. Martins, C. Gr, I. Lanese, V. T. Vasconcelos, and F. Martins. Disciplining orchestration and conversation. In *in Service-Oriented*

Computing. In 5th IEEE International Conference on Software Engineering and Formal Methods, 2007.
13. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In R. De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-71316-6_4.
14. A. Lapadula, R. Pugliese, and F. Tiezzi. A wsdl-based type system for asynchronous ws-bpel processes. *Formal Methods in System Design*, 38(2):119–157, 2011.
15. A. Martens. Analyzing web service based business processes. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-31984-9_3.
16. D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In S. Della Rocca, editor, *Typed Lambda Calculi and Applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 321–335. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73228-0_23.
17. Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, Apr. 2007.
18. C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.
19. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types, 2003.

Appendices

A Typing Rules

$$\frac{g \xrightarrow[\mathsf{pic}]{s@p} g' \quad \grave{m} \vdash p : P \quad (\grave{m} \vdash s : \Delta \text{ or } \grave{m} \vdash s : \bot)}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{P}/_{s}], \mathsf{pic} \blacktriangleright g')} \boxed{@}$$

$$\frac{g \xrightarrow[\mathsf{pic}]{s@p} g' \quad \grave{m} \vdash p : P \quad \neg(\grave{m} \vdash s : \Delta \text{ or } \grave{m} \vdash s : \bot)}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{\nabla}/_{s}], \mathsf{pic} \blacktriangleright g')} \boxed{\overline{@}}$$

$$\frac{\grave{m} \vdash g \xrightarrow[\mathit{Types}\ \mathsf{pic}]{s!op(\tilde{X})} g' \quad \grave{m} \vdash s : T \quad T \xrightarrow{?op(\tilde{X})} T'}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{T'}/_{s}], \mathsf{pic} \blacktriangleright g')} \boxed{!?}$$

$$\frac{\grave{m} \vdash g \xrightarrow[\mathit{Types}\ \mathsf{pic}]{s!op(\tilde{X})} g' \quad \grave{m} \vdash s : T \quad T \not\xrightarrow{?op(\tilde{X})}}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{\nabla}/_{s}], \mathsf{pic} \blacktriangleright g')} \boxed{!\overline{?}}$$

$$\frac{\grave{m} \vdash g \xrightarrow[\mathit{Types}\ \mathsf{pic}]{s?op(\tilde{X})} g' \quad \grave{m} \vdash s : T \quad T \xrightarrow{!op(\tilde{X})} T'}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{T'}/_{s}], \mathsf{pic} \blacktriangleright g')} \boxed{?!}$$

$$\frac{open(\mathsf{pic}, g, s) \quad \grave{m} \vdash s : T \quad T \xrightarrow{!op(\tilde{X})} T' \quad \grave{m} \vdash g \not\xrightarrow[\mathit{Types}\ \mathsf{pic}]{s?op(\tilde{X})}}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{\nabla}/_{s}], \mathsf{pic} \blacktriangleright g)} \boxed{\overline{?}!}$$

$$\frac{open(\mathsf{pic}, g, s) \quad \grave{m} \vdash s : T \quad T \not\xrightarrow{!}}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{\nabla}/_{s}], \mathsf{pic} \blacktriangleright g)} \boxed{?\bar{!}}$$

$$\frac{g = \mathrm{term}(\mathsf{pic}) \quad \grave{m} \vdash s : T \quad T \xrightarrow{?}}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{\nabla}/_{s}], \mathsf{pic} \blacktriangleright g)} \boxed{\bar{!}?}$$

$$\frac{g = \mathrm{term}(\mathsf{pic}) \quad \grave{m} \vdash s : T \quad T \xrightarrow{!}}{(\grave{m}, \mathsf{pic} \blacktriangleright g) \rightarrow (\grave{m}[^{\nabla}/_{s}], \mathsf{pic} \blacktriangleright g)} \boxed{\overline{?}!_{\mathrm{bis}}}$$

Notations for session types transitions

$$(T \xrightarrow{\eta}) \Leftrightarrow (\exists T' : T \xrightarrow{\eta} T') \qquad (T \not\xrightarrow{\eta}) = \neg(T \xrightarrow{\eta})$$
$$(T \xrightarrow{!}) \Leftrightarrow (\exists op, \tilde{X} : T \xrightarrow{!op(\tilde{X})}) \qquad (T \not\xrightarrow{!}) = \neg(T \xrightarrow{!})$$
$$(T \xrightarrow{?}) \Leftrightarrow (\exists op, \tilde{X} : T \xrightarrow{?op(\tilde{X})}) \qquad (T \not\xrightarrow{?}) = \neg(T \xrightarrow{?})$$