Sessionizing BPEL

Jonathan Michaux
Télécom ParisTech
Paris, France
michaux@telecom-paristech.fr

Elie Najm
Télécom ParisTech
Paris, France
najm@telecom-paristech.fr

Alessandro Fantechi
Universita' degli Studi di Firenze
Florence, Italy
fantechi@dsi.unifi.it

We address the general problem of interaction safety between orchestrated web services. By considering an essential subset of the BPEL orchestration language, we define SeB, a session based style of this subset. We discuss the formal semantics of SeB and we present its main properties. We use a new approach to address the formal semantics, based on a translation into so-called control graphs. Our sematics handles control links and addresses the static semantics that prescribes the valid usage of variables. We also provide the semantics of collections of networked services. Relying on these semantics, we define precisely what is meant by interaction safety. Finally, we quickly introduce session types and sketch an approach towards the fullfilment of interaction safety through well typed SeB programs and well typed service assemblies.

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 context, our endeavour is to guarantee that services interact safely. To that end, we investigate means to check, at deployment time, whether or not interacting services are compatible and will not yield interaction errors at run time.

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.

The present paper is a first step towards addressing the problem of behavioural compatibility of web services. To that end, we follow the promising session based approach. Indeed, the session paradigm is now an active area of research with potential to improve the quality and correctness of software. We chose 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

Submitted to:
WWV'12

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. In the present paper, we concentrate on the definition of untyped SeB leaving the discussion of session types for the sequel.

We introduce a formal semantics for SeB (and BPEL like languages) which is novel in the sense that it takes into account both the graph nature of the language and the static semantics that define how variables are declared and used. Indeed, previous approaches either resort to process algebraic simplifications, thus neglecting control links which, in fact, are an essential part of BPEL; or are based on Petri nets and thus do not properly cover the static semantics that regulate the use of variables.

In our approach, the operational semantics is obtained in two steps. The first consists in the creation of what we have called a control graph. This graph takes into account the effect of the evaluation of the control flow and of join-conditions. Control graphs contain symbolic actions and no variables are evaluated in the translation into control graphs. The second step in the operational semantics describes the execution of services when they are part of an assembly made of a client and other web services. Based on this semantics we formalize the concept of interaction error and iteraction safety.

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 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 (this corresponds to a BPEL partnerlink). 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 provide for explicit correlation sets as does BPEL. But, on the other hand, sessions are to be considered as implicit correlation sets and, indeed, they can be emulated by them. But, in this paper, we prefered to have sessions as part of the language so as to better discuss and illustrate their contribution. Hence, in SeB session ids are the only means to identify source and target service instances. This is 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 expressive than correlation sets. At the end of the paper, we will give some ideas as to how this limitation can be lifted.

Structured activities. SeB has the principal structured activities of BPEL, i.e., flow, for running activities in parallel; sequence, for sequential composition of activities, and pick, which allows to wait on multiple messages whereby the continuation behavior is dependent on the received message. SeB also inherits the possibility of having control 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.

Example Service. Figure 1 contains a graphical representation of a service written in SeB. The QuoteComparer service waits for a client to invoke its $s_0?quoteCompare(desc)$ operation with string parameter *desc* containing an item description. Note the use of the special session variable s_0. By accepting the initial request from the client, the service implicitely begins an interaction with a new session called the root session. The service then compares quotes for the item from two different providers (*EZhop* and *QuickBuy*) by opening sessions with each of these providers. Here, the sessions are explicitely opened: $s@EZshop$ is the opening of a session with the service having its address stored in variable *EZshop*. Execution of $s@EZshop$ will result in a root session initiated at *EZshop* side and a fresh session id stored in *s*. Then, the behavior of this service is as follows. Depending on the returns made by the two providers, QuoteComparer either returns the best quote, the only available quote, or indicates to the client that no quote is available. The snapshot given in Figure 1 shows only the control links and the join condition that cause EZShop to be selected as the quote provider. Indeed, the join condition of the bottom left sequence (given by JCD = $\cdots$) indicates that this sequence should begin executing if the code from *Compare* decides that EZShop's quote is favourable (the control link $COMP\_sBUY$ is set to true), or if EZShop returns a quote and QuickBuy returns *noQuote*. In order not to clutter the figure, we have left out the control links and join conditions that correspond to the other cases. The same goes for the implementation details of code in *compare* and *Proceed with payment*. We also assumed that the service only begins with one **REC** operation whereas services may provide multiple initial operations by grouping the receive activities in a **PIC**.

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 Table 1 where, for each set, a short description is provided as are the names of typical elements.

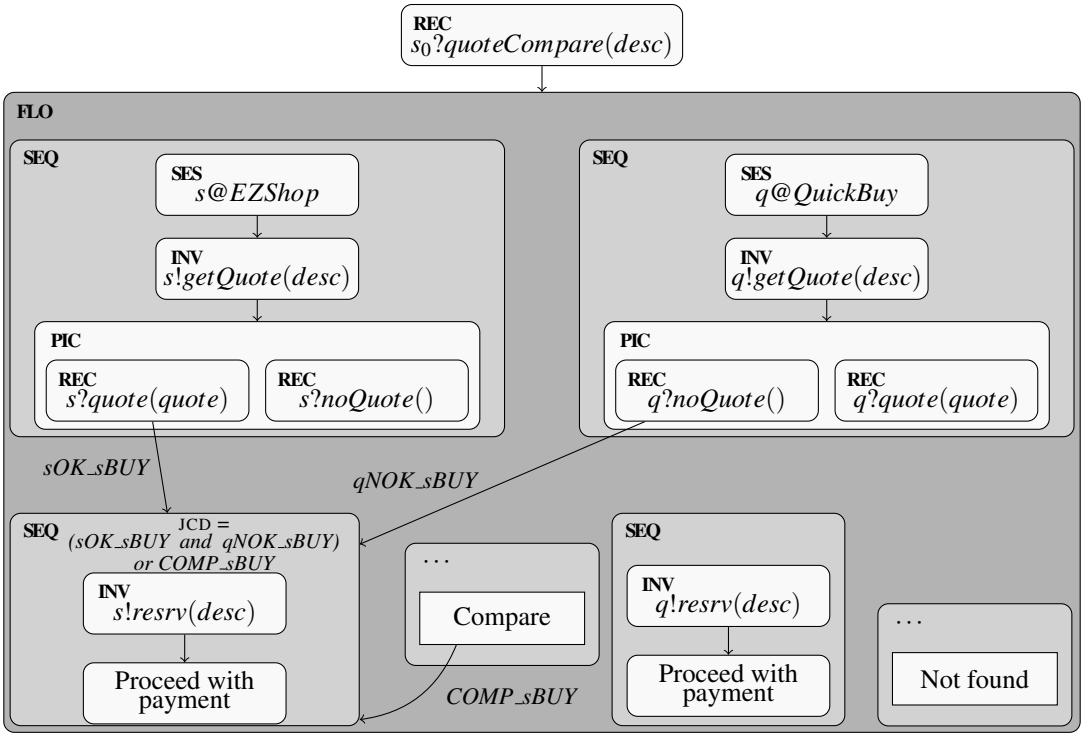

Figure 1: The QuoteComparer Service

All the sets are pairwise disjoint unless otherwise stated.

Some explanations of Table 1 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 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.

Reconsidering our previous example, *EZShop* and *QuickBuy* are service location variables (elements of set *SrvVar*, with values taken in the set *SrvVal* of Service Locations); *desc* is a data variable (with values in set *DatVal*); s_0, s and q are session variables (elements of *SesVar*, taking their values in the set *SesVal* of session ids); *sOK_sBUY*, *qNOK_sBUY* and *COMP_sBUY* are control links; and finally expression *(sOK_sBUY and qNOK_sBUY) or COMP_sBUY* is a join condition.

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

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	$P, P', P_i \cdots$
$ExTyp = DatTyp \uplus SrvTyp$	Types of Exchangeable Values	$X, X', X_i \cdots Y$
$SesTyp \supset SrvTyp$	Session Types	$T, T', T_i \cdots P, P', P_i$
$Typ = ExTyp \cup SesTyp$	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**

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 $\langle \text{KND} = \textbf{FLO}, \text{BEH} = \mathit{my\_behaviour},\ \text{TGT} = L_{\text{T}}, \text{SRC} = L_{\text{S}}, \text{JCD} = e, \text{LNK} = L \rangle$ where L_{T}, L_{S} 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: $\langle \textbf{FLO}, \mathit{my\_behaviour},\ L_{\text{T}}, L_{\text{S}}, e, L \rangle$.

We let ACT be the set of all activities and act a running element of 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	::=	$\langle \textbf{SES}, s@p,\ L_{\text{T}},\ L_{\text{S}},\ e \rangle$	(* session initiation *)
inv	::=	$\langle \textbf{INV}, s!op(x_1, \cdots, x_n),\ L_{\text{T}},\ L_{\text{S}},\ e \rangle$	(* invocation *)
rec	::=	$\langle \textbf{REC}, s?op(x_1, \cdots, x_n),\ L_{\text{T}},\ L_{\text{S}},\ e \rangle$	(* reception *)
seq	::=	$\langle \textbf{SEQ}, (\mathsf{act}_1; \cdots; \mathsf{act}_n),\ L_{\text{T}},\ L_{\text{S}},\ e \rangle$	(* sequence *)
flo	::=	$\langle \textbf{FLO}, (\mathsf{act}_1 \mid \cdots \mid \mathsf{act}_n),\ L_{\text{T}},\ L_{\text{S}},\ e, L \rangle$	(* flow *)
pic	::=	$\langle \textbf{PIC}, (\mathsf{rec}_1; \mathsf{act}_1) + \cdots + (\mathsf{rec}_n; \mathsf{act}_n),\ L_{\text{T}},\ L_{\text{S}},\ e \rangle$	(* pick *)
rep	::=	$\langle \textbf{REP}, (\mathit{do}\ \mathsf{pic}_1\ \mathit{until}\ \mathsf{pic}_2),\ L_{\text{T}},\ L_{\text{S}},\ e \rangle$	(* repeat *)

Note that in the production rule for flo, "|" is to be considered merely 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. Note that "|" and "+" are commutative.

Returning to our example in Figure 1 and considering the activity seq at the bottom left, its syntax is given by the following expression:

$\langle \textbf{SEQ},$
$\quad (\langle \textbf{INV}, s!resrv(desc),\ \emptyset,\ \emptyset,\ true \rangle\ ;\ \langle \cdots \rangle),$
$\quad \{sOK\_sBUY, qNOK\_sBUY, COMP\_sBUY\},$
$\quad \emptyset,$
$\quad$ *(sOK_sBUY and qNOK_sBUY) or COMP_sBUY*
$\rangle$

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

Control links non cyclicity
Relation *pred* defined by: act *pred* act' iff act.SRC $\cap$ act'.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. We have found that this approach yields more concise semantics than those that can be found in similar work in Petri net based approaches, notably when it comes to capturing the behavior of dead-path elimination. Also, this approach allows us to take into account control links while at the same time considering static semantics. This is important because control-flow and data-flow are interdependent in BPEL and therefore need to be addressed jointly. 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:
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 ACTIONS_τ of all actions (ranged over by σ):
$\text{ACTIONS}_\tau =_{\text{def}} \text{ACTIONS} \cup \{\tau\}$ where τ denotes the unobservable action.

Control Graphs A control graph, Γ, is a labeled transition system with the following structure:
$\Gamma =< G, g_0, \mathscr{A}, \rightarrow >$ where

- G is a set of states, called control states
- g_0 is the initial control state
- $\mathscr{A}$ is a set of actions ($\mathscr{A} \subset \text{ACTIONS}_\tau$)
- $\rightarrow \subset G \times \mathscr{A} \times G$

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. $c : Lnk \rightarrow \{\textbf{true}, \textbf{false}, \perp\}$

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

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. Also, in activity flo we dropped field LNK since its value is constant (LNK is used to define a scope for control link variables). The rules for the repeat activity rep are given by first replacing the rep record, $(do\ \mathsf{pic}_1\ until\ \mathsf{pic}_2)$ with a triple noted $(\mathsf{pic}_1\ [\mathsf{pic}_1\mathsf{>pic}_2)$. The meaning of each of the constituents of the triple $(\mathsf{pic}_1\ [\mathsf{act}\mathsf{>pic}_2)$ is as follows: act encodes the current state of the repeat, pic_1 is the activity to be restarted when act becomes activity **nil**, pic_2 is the activity that when started will terminate the repeat. Also, in order to have an escape from the repeat, a static rule enforces that pic_2 is not equal to **nil**. A final notational convention is that "$*$" denotes a wildcard activity, as seen in the rule for dead-path elimination (DPE).

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., $c[^{\mathbf{false}}/\widehat{\Pi}.\mathrm{SRC}]$ is the substitution whereby any control link occuring as source of an activity in $\widehat{\Pi}$ has its value set to **false**.

Rule 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 > INV > SES > 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 later in the paper.

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 $\mathbf{cg}(\mathsf{act})$. Figure 2 shows the control graph generated for the QuoteComparer SeB service (Figure 1) based on the SOS rules defined in subsection 4.2.

In this control graph, let us consider the transition between states 9 and 11 which is labelled with action q?noQuote(). State 9 is the state when both sessions s and q have been started and the two invocations s!getQuote(desc) and q!getQuote(desc) have been sent. At state 9, the value of all control links is undefined ($\bot$). Transition $9 \rightarrow 11$ transforms the receive activity q?noQuote() to nil and sets the control link *qNOK_sBUY* to true. The τ transition $9 \rightarrow 11$ represents the handling of the dead path elimination whereby all control links outgoing from the receive activity q?quote(quote) are set to false.

Properties of Control Graphs of Activities: The following properties can be proven about control graphs of activities:

- For any act the set of control states of $\mathbf{cg}(\mathsf{act})$ is finite. This is true because, on the one hand, control links are a fixed vector which have a finite combination, and on the other hand, the set of activities that can be derived from act is finite (all activities, except repeat, are reductions, and, for repeat, all possible derivatives of $(\mathsf{pic}_1\ [\mathsf{act}\mathsf{>pic}_2)$ are syntactically smaller than $(\mathsf{pic}_1\ [\mathsf{pic}_1\mathsf{>pic}_2)$.

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

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

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

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

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

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

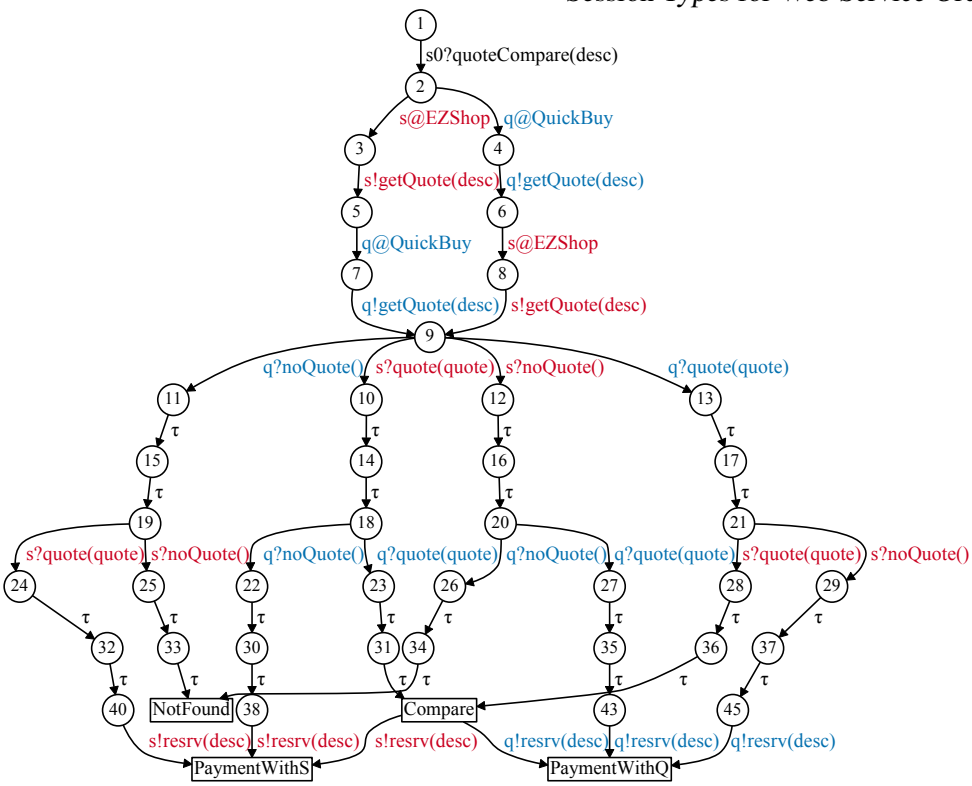

Figure 2: The QuoteComparer Control Graph

- The set of control states of **cg**(act) is partitioned into four categories: *silent* states, *receiving* states, *transient* states, and *terminal* states. 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. This stems directly from the prioritized order of application of the SOS rules. For instance, if in a given state both a "tau" and a "receive" transition are firable then only the "tau" transition will be taken because it has higher priority over the "receive" transition. Moreover, the silent τ transitions are confluent since no two τ transitions can be mutually conflicting. This can be proven by visiting all SOS rules and verifying that if two τ transitions are possible then if one is taken the other is not discarded. (in fact, we can also show that transient transitions are also confluent).

- All terminal states are of the form (c, **nil**). Hence they differ only by their control link map, and, for any state, there exists a path that leads to a terminal state. This is true because any activity that is head (i.e a candidate for execution) is either fireable (when the join condition yields true) or cancellable (when the join condition yields false). In both cases, there is a possible transition that removes the activity. Hence successor activities in turn become head and this proceeds until **nil** is reached.

A consequence of the above properties is when applying 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: after a receive, all possible transient actions are executed before another message can be received. Henceforth, when we write **cg**(act) we will consider that we are dealing with the minimised control graph, and we will name its unique terminal state **term**(act).

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

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

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: $V(\text{act}) =_{\text{def}} \{\ z \mid z \text{ occurs in } \widehat{\text{act}}\ \}$

Binding occurrences. For variables $y \in V(\text{act})$, $s \in V(\text{act})$ and $p \in V(\text{act})$, 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 $BV(\text{act})$ the set of variables having a binding occurrence in act.

Usage occurrences. For variables $y \in V(\text{act})$, $s \in V(\text{act})$ and $p \in V(\text{act})$, 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 $UV(\text{act})$ the set of variables having at least one usage occurrence in act.

Free occurrences. A variable $z \in V(\text{act})$ is said to occur free in act, iff there is a path in **cg**(act): $g_0 \xrightarrow[\text{act}]{\sigma_1} g_1, \cdots, g_{n-1} \xrightarrow[\text{act}]{\sigma_n} g_n$ 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 $FV(\text{act})$ 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 $Val \cup \{\perp\}$, the set of values augmented with the undefined value. Henceforth, we consider couples (m, act) where $dom(m) = V(\text{act})$.

Deployable services. The couple (m, pic) is a deployable service iff:

- $m(p_0) \neq \perp$ (the service has a defined location address recorded in p_0),
- $\text{pic.BEH} = \sum s_0?op_i(\tilde{x_i})\ ;\ \text{act}_i$ (pic has all its receive actions on session s_0),
- $FV(\text{act}) \cap SesVar = \{s_0\}$ (s_0 is the only free session variable),

- $\forall z \in FV(\mathsf{act}) \setminus \{s_0\} : m(z) \neq \perp$ (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 **init**(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 similarly define 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 $(m, \mathsf{act} \triangleright \mathbf{init}(\mathsf{act}))$.

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:

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

Networked configurations. A networked configuration is a triple $\lfloor\!\lceil\ C\ \rceil\!\rfloor\ \lfloor\!\lceil\ Q\ \rceil\!\rfloor\ \lfloor\!\lceil\ \mathscr{B}\ \rceil\!\rfloor$ where C is a running configuration, Q is a set of message queues and $\mathscr{B}$ a set of session bindings. Q and $\mathscr{B}$ 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 $\mathscr{B}$ and has the syntax $\mathscr{B} ::= (\alpha, \beta) \mid (\alpha, \beta) \diamond \mathscr{B}$. If $(\alpha, \beta) \in \mathscr{B}$ 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

We now turn to the semantics of networked services. Our aim is twofold: to provide a full semantics to SeB and to formalize the property that we want to assess in SeB programs. The semantics is again defined using the following 4 SOS rules.

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

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

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

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

Rule SES1 applies when one of the service instances has a session initiation transition, $s@p$, with $m(p)=\pi$, i.e., the address of the remote service is π. In that case, two fresh session ids are created (α,β), the first is for the local session end, and the second for the distant session end, and a message $new(\beta)$ is put at the tail of the queue towards service π.
Rule SES2 applies precisely when message $new(\beta)$ is the head of the input queue of the service located at π (this service verifies $m(p_0)=\pi$). The effect of the rule is that the message is consumed and a new service instance is created with root session s_0 set to β.
Rules INV states that when a service instance is ready to send an invocation message over session s whose session id is α, then the message is appended to the queue whose target is β which is bound to α. Rule REC is symmetrical to rule INV.

Property of safe interaction. We now turn to the property that we wish to verify through session types and well typedness algorithm. In a running configuration, interaction safety is verified when the following situation never occurs: a service reaches a state where it waits for an input on a session, and the message that is at the head of the queue for that session is not expected. i.e., the service has a no matching pick or receive activity. Stated formally, the property is as follows: In a state, Ω, reached by a well typed collection, with $\Omega = \lfloor \cdots (m,\mathsf{pic} \triangleright g) \cdots \rceil \lfloor \cdots (\beta \hookleftarrow op(\tilde{w})\cdot\tilde{M}) \cdots \rceil \lfloor \mathscr{B} \rceil$ where $m(s)=\beta$ and $open(\mathsf{pic},g,s)$ then $\Omega \rightarrow \lfloor \cdots (m',\mathsf{pic} \triangleright g') \cdots \rceil \lfloor \cdots (\beta \hookleftarrow \cdot\tilde{M}) \cdots \rceil \lfloor \mathscr{B} \rceil$.

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:

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

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) $\triangledown$ 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 adapted 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:

$$\begin{array}{ccc} ST_1 & \overset{\mathcal{SwP}}{\text{———}} & ST_2 \\ \Big\downarrow {\scriptstyle !op(X_1,\cdots,X_n)} & & \vdots\Big\downarrow {\scriptstyle !op(Y_1,\cdots,Y_n)} \\ ST_1' & \overset{\mathcal{SwP}}{\cdots\cdots} & ST_2' \end{array} \qquad \text{where } \forall i : X_i\ \mathcal{SwP}\ Y_i$$

$$\begin{array}{ccc} ST_1 & \overset{\mathcal{SwP}}{\text{———}} & ST_2 \\ \vdots\Big\downarrow {\scriptstyle ?op(X_1,\cdots,X_n)} & & \Big\downarrow {\scriptstyle ?op(Y_1,\cdots,Y_n)} \\ ST_1' & \overset{\mathcal{SwP}}{\cdots\cdots} & ST_2' \end{array} \qquad \text{where } \forall i : X_i\ \mathcal{SwP}\ Y_i$$

In the above digrams, we consider that if X_i and Y_i are data types, then $X_i \; \mathit{SwP} \; Y_i \Leftrightarrow X_i = Y_i$. 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, SwP, such that $ST_1 \; \mathit{SwP} \; ST_2$.

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 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 $(\grave{m}, \mathsf{pic} \blacktriangleright \mathbf{init}(\mathsf{pic}))$.

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

A Web service orchestration that is "interaction safe" does not lead to interaction errors, such as unexpected messages being sent or received from one partner to another. In order to provide a basis for the verification of such a property, we have adapted and formalised a subset of the widely adopted orchestration language BPEL. The resulting formalism, that we call SeB for Sessionized BPEL, supports typed sessions as first class citizens of the language. The formalisation we have given is novel because the operational semantics are broken into two steps. This seperation has allowed us to define relatively consise semantics compared to other approaches. Furthermore our semantics take into account the effect of BPEL control links, which are an essential and often neglected part of the language.

We briefly introduce session types as a means of prescribing 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 show how session types could be used to 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 and is therefore 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 & G. Zavattaro (2006): *A Service Centered Calculus*. In Mario Bravetti, Manuel Nez & Gianluigi Zavattaro, editors: *Web Services and Formal Methods*, *Lecture Notes in Computer Science* 4184, Springer Berlin / Heidelberg, pp. 38–57. Available at `http://dx.doi.org/10.1007/11841197_3`. 10.1007/11841197_3.

[2] Roberto Bruni, Ivan Lanese, Hernán Melgratti & Emilio Tuosto (2008): *Multiparty Sessions in SOC*. In Doug Lea & Gianluigi Zavattaro, editors: *Coordination Models and Languages*, *Lecture Notes in Computer Science* 5052, Springer Berlin / Heidelberg, pp. 67–82. Available at `http://dx.doi.org/10.1007/978-3-540-68265-3_5`. 10.1007/978-3-540-68265-3_5.

[3] Luca Cardelli & John Mitchell (1990): *Operations on records*. In M. Main, A. Melton, M. Mislove & D. Schmidt, editors: *Mathematical Foundations of Programming Semantics*, *Lecture Notes in Computer Science* 442, Springer Berlin / Heidelberg, pp. 22–52. Available at `http://dx.doi.org/10.1007/BFb0040253`. 10.1007/BFb0040253.

[4] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino & Luca Padovani (2009): *Foundations of Session Types*. In: *PPDP'09*, ACM Press, pp. 219–230. Available at `http://www.di.unito.it/~dezani/papers/cdgp.pdf`. Full version:http://www.di.unito.it/ dezani/papers/cdgpFull.pdf.

[5] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman & Sanjiva Weerawarana (2007): *Web Service Definition Language (WSDL) Version 2.0*. Technical Report. Available at `http://www.w3.org/TR/wsdl20`.

[6] Alessandro Fantechi & Elie Najm (2008): *Session Types for Orchestration Charts*. In Doug Lea & Gianluigi Zavattaro, editors: *Coordination Models and Languages*, *Lecture Notes in Computer Science* 5052, Springer Berlin / Heidelberg, pp. 117–134. Available at `http://dx.doi.org/10.1007/978-3-540-68265-3_8`. 10.1007/978-3-540-68265-3_8.

[7] Kohei Honda, Vasco Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In Chris Hankin, editor: *Programming Languages and Systems*, *Lecture Notes in Computer Science* 1381, Springer Berlin / Heidelberg, pp. 122–138. Available at http://dx.doi.org/10.1007/BFb0053567. 10.1007/BFb0053567.

[8] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. *SIGPLAN Not.* 43, pp. 273–284, doi:http://doi.acm.org/10.1145/1328897.1328472. Available at http://doi.acm.org/10.1145/1328897.1328472.

[9] David Kitchin, Adrian Quark, William Cook & Jayadev Misra (2009): *The Orc Programming Language*. In David Lee, Antnia Lopes & Arnd Poetzsch-Heffter, editors: *Formal Techniques for Distributed Systems*, *Lecture Notes in Computer Science* 5522, Springer Berlin / Heidelberg, pp. 1–25. Available at http://dx.doi.org/10.1007/978-3-642-02138-1_1. 10.1007/978-3-642-02138-1_1.

[10] Oliver Kopp, Rania Khalaf & Frank Leymann (2008): *Deriving Explicit Data Links in WS-BPEL Processes*. In: *IEEE SCC (2)*, pp. 367–376. Available at http://doi.ieeecomputersociety.org/10.1109/SCC.2008.122.

[11] R. Pugliese A. Ravara L. Caires, G. Ferrari (2006): *Behavioural Types for Service Composition*. Technical Report, Sensoria project.

[12] Ivan Lanese, Vasco T. Vasconcelos, Francisco Martins, Campo Gr, Ivan Lanese, Vasco T. Vasconcelos & Francisco Martins (2007): *Disciplining Orchestration and Conversation*. In: *in Service-Oriented Computing. In 5th IEEE International Conference on Software Engineering and Formal Methods*.

[13] Alessandro Lapadula, Rosario Pugliese & Francesco Tiezzi (2007): *A Calculus for Orchestration of Web Services*. In Rocco De Nicola, editor: *Programming Languages and Systems*, *Lecture Notes in Computer Science* 4421, Springer Berlin / Heidelberg, pp. 33–47. Available at http://dx.doi.org/10.1007/978-3-540-71316-6_4. 10.1007/978-3-540-71316-6_4.

[14] Alessandro Lapadula, Rosario Pugliese & Francesco Tiezzi (2011): *A WSDL-based type system for asynchronous WS-BPEL processes*. *Formal Methods in System Design* 38(2), pp. 119–157. Available at http://dx.doi.org/10.1007/s10703-010-0110-0.

[15] Axel Martens (2005): *Analyzing Web Service Based Business Processes*. In Maura Cerioli, editor: *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science* 3442, Springer Berlin / Heidelberg, pp. 19–33. Available at http://dx.doi.org/10.1007/978-3-540-31984-9_3. 10.1007/978-3-540-31984-9_3.

[16] Dimitris Mostrous & Nobuko Yoshida (2007): *Two Session Typing Systems for Higher-Order Mobile Processes*. In Simona Della Rocca, editor: *Typed Lambda Calculi and Applications*, *Lecture Notes in Computer Science* 4583, Springer Berlin / Heidelberg, pp. 321–335. Available at http://dx.doi.org/10.1007/978-3-540-73228-0_23. 10.1007/978-3-540-73228-0_23.

[17] Organization for the Advancement of Structured Information Standards (OASIS) (2007): *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Available at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[18] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas & Arthur H. M. ter Hofstede (2007): *Formal semantics and analysis of control flow in WS-BPEL*. *Sci. Comput. Program.* 67(2-3), pp. 162–198. Available at http://dx.doi.org/10.1016/j.scico.2007.03.002.

[19] Antonio Vallecillo, Vasco T. Vasconcelos & Antnio Ravara (2003): *Typing the Behavior of Objects and Components using Session Types*.