Highlights

- We give a novel formal semantics of the BPEL orchestration language.
- We extend BPEL to support sessions.
- We provide behavioural typing of services by using typed sessions.
- We provide a solution to check the well-typedness of services and service configurations.
- We prove the property of safe interaction for well-typed service configurations.
Session types for safe Web service orchestration

Jonathan Michaux\textsuperscript{a,1}, Elie Najm\textsuperscript{a}, Alessandro Fantechi\textsuperscript{b}

\textsuperscript{a}Téléscom ParisTech, 46 rue Barrault, 75013 Paris, France
\textsuperscript{b}Università degli Studi di Firenze, via S. Marta 3, I-50139, Florence, Italy

Abstract

We address the general problem of interaction safety in Web service orchestrations. 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 present its main properties. We take a new approach to address the formal semantics which is based on a translation into so-called control graphs. Our semantics accounts for BPEL control links and addresses the static semantics that prescribes the valid usage of variables. We also provide the semantics of service configurations.

During a session, a client and a service can engage in a complex series of interactions. By means of the provided semantics, we define precisely what is meant by interaction safety. We then 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 a service orchestration behaves according to its declared provided and required types.

Using a subtyping relation defined on session types, we show that any configuration of well-typed service partners with compatible session types are interaction safe, i.e., involved partners never receive unexpected messages.

Keywords: Session types, Orchestration, Web Services, BPEL, Interaction-safety, Behavioural compatibility

\textsuperscript{1}Research supported by the MOTELI project - FUI8, Cap Digital
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 this aim, we want to 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. Also, an orchestration typically requires support for concurrency in order to invoke multiple services simultaneously rather than sequentially.

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) [1] provide support for syntactical compatibility analysis by defining message types in a standard way [2]. 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 session paradigm is an active area of research that has the potential to improve the quality and correctness of software, and can arguably contribute to verifying the behavioural compatibility of Web services (e.g. [3, 4]). One approach is the deployment-time detection of certain types of interaction errors in which (using a simplified notation) a service instance is in a waiting state given by: \( ?op() + ?op'() \), but the invocation message residing at the head of the queue targeting the instance is \( op''() \) with \( op'' \notin \{op, op'\} \).

To that end, we follow the promising session based approach by adapting and sessionizing a significant subset of the industry standard orchestration language BPEL [5]. On one hand, 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 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. A SeB service exposes its required and provided session types, and a client wishing to interact with a service begins by opening a session with it. Thus, the interaction that follows the session initiation should behave according to the guidelines declared by the type of the opened session.

In the present paper, we first focus on the definition of untyped SeB which is given a formal semantics that 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 [6] 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 control flow part of a SeB activity, including the evaluation 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 concepts of interaction error and of interaction safety.

We then equip SeB with explicit typing by use of session types and 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 clients’ required session types with the providers’ provided session types. Finally, we 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 organized 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, and the concepts of interaction error and of interaction safety. Section 6 introduces session types and typed SeB. Section 7 shows how we use session types in order to prove interaction safety.

Relevant related work is surveyed in section 8 and the paper is concluded in section 9.

2. Informal introduction to SeB

Session initiation. The main novelty in SeB, compared to BPEL, resides in the addition of the session initiation, a new kind of atomic activity, and in the way sessions impact the invoke and receive activities. The following is a typical sequence of three atomic SeB 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 the endpoint reference of a BPEL partnerlink). The execution of $s@p$ by the client and by the target service (the one whose address is stored in $p$) has the following effects: (i) a fresh session id is stored in $s$, (ii) a new service instance is created on 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 $s$ on the client side. 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.

Sessions in lieu of correlation sets. The mechanism used in BPEL to relate and route distinct messages from one or more clients towards a particular service instance is called correlation. Correlation sets are application-level message fields whose values are used as a basis for correlation. Note that the purpose of both sessions and correlation sets is to maintain long-lasting interactions between process instances, but correlation sets are more implicit by nature. For example, with correlation sets, a service’s lifecycle is hidden from clients, and the initiation of and reference to a specific instance of an interaction cannot be done explicitly and with any certainty. SeB does not feature correlation sets and instead relies on sessions as an explicit language
element. Identifiable sessions are particularly useful as one may then associate types to sessions, which facilitates the fulfilment of our goal which is to check interaction safety. Indeed, analysis of interactions that stem from correlations seems difficult [7]. Hence, in SeB, session identifiers are the only means to refer to the instances at each end of a conversation. This is illustrated in the above example where the session variable $s$ is systematically indicated in the invoke and receive activities.

SeB uses biparty sessions, therefore interactions involve strictly two partners. Correlation sets are more expressive than biparty sessions in the sense that they can, for example, allow multiple clients that are not aware of each others’ existence to communicate with one single service instance if the right correlation data is included in messages. Multiparty sessions have been studied in the literature [7], as well as multiparty session types [8], and offer a solution as to how this limitation can be lifted from biparty sessions (see section 8). Multiparty sessions can in fact be shown to emulate some of the behaviour that can be defined with correlation sets.

**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 (also known as external choice), which waits for multiple messages, the continuation behaviour being dependent on the received message. SeB also inherits from BPEL the possibility of having control links between concurrent 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 so-called dead path elimination (DPE) whereby all links outgoing from a cancelled activity, or from its subactivities, have their values set to false.

**Imperative Computations.** Given that SeB is a language designed as a proof of concept, we limit its features to interaction behaviour. Hence, imperative computation and conditional 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 imperative computation and boolean tests are provided by external sites. Furthermore, we replace BPEL’s original *do until* iteration operator with a structured activity called “repeat”, given by the syntax: $\text{(do } \text{pic}_1 \text{ until } \text{pic}_2)$. 
The informal meaning of repeat is: perform \( \text{pic}_1 \) repeatedly until the arrival of an invocation message awaited for in \( \text{pic}_2 \).

**Example Service.** The QuoteComparer is an example of a service written in SeB that will be used throughout the paper. Given an item description from a client, the purpose of the service is to offer a comparison of quotes from different providers for the item and to reserve the item with the best offer. Figure 1 contains a graphical representation of the QuoteComparer program in SeB. In this figure, each rounded rectangle represents an activity whose kind is indicated in the upper left corner. The service’s first activity, given by \( s_0?\text{searchQuote(desc)} \), is the reception of an invocation of operation \text{searchQuote} from a client with string parameter \( \text{desc} \) containing an item description. Note the use of the special session variable \( s_0 \), called the root session variable. By accepting the initial request from the client, the service implicitly begins an interaction with the client over session \( s_0 \). The service then compares quotes for the item from two different providers (\( \text{EZshop} \) and \( \text{QuickBuy} \)) by opening sessions with each of these providers. Here, the sessions are explicitly opened: \( s@\text{EZshop} \) is the opening of a session with the service whose address is stored in variable \( \text{EZshop} \). The execution of \( s@\text{EZshop} \) will result in a root session being initiated at the \( \text{EZshop} \) service and a fresh session id being stored in \( s \).

The service then behaves in the following way: depending on the returns made by the two providers, the QuoteComparer service either returns the best quote, the only available quote, or indicates to the client that no quote is available. The control links and join conditions illustrated in Figure 1 implement this behaviour. Indeed, note how activity \text{Flo} is composed of 6 subactivities that run in parallel, except that they are not independent: they have to synchronise through control links \( l_1 \) to \( l_{10} \). Control links are represented by labelled arrows, whereas unlabelled arrows represent the flow of activities in a sequence activity (\text{Seq}). For example, control link \( l_1 \) is set to true if \( \text{EZshop} \) returns message \( \text{quote(quote1)} \), meaning \( \text{EZshop} \) has an offer for the item. The value of this link and others will be later used to determine which provider’s quote should be chosen. Indeed, the join condition (given by \( \text{JCD} = (l_1 \text{and } l_2) \text{ or } l_3 \)) of the bottom left sequence, that deals with reserving an item with \( \text{EZshop} \), depends on the value of control link \( l_1 \). The value of this link and others will be later used to determine which provider’s quote should be chosen. Indeed, the join condition (given by \( \text{JCD} = (l_1 \text{and } l_2) \text{ or } l_3 \)) of the bottom left sequence, that deals with reserving an item with \( \text{EZshop} \), depends on the value of control link \( l_1 \). The value of this link and others will be later used to determine which provider’s quote should be chosen. Indeed, the join condition (given by \( \text{JCD} = (l_1 \text{and } l_2) \text{ or } l_3 \)) of the bottom left sequence, that deals with reserving an item with \( \text{EZshop} \), depends on the value of control link \( l_1 \). It also depends on control link \( l_3 \) which comes from the bottom central sequence that decides, in the case where both providers offer a quote, which quote is most advantageous. Link \( l_2 \) will be set to true only if provider \( \text{QuickBuy} \)
does not return a quote. Hence, the join condition of the bottom left sequence indicates that this sequence should begin executing either if EZShop’s quote is favourable (the control link \(l_3\) is set to true), or if EZShop returns a quote and QuickBuy returns \(\text{noQuote}\) (\(l_1\) and \(l_2\) are set to true). The conditions for reserving with the QuickBuy provider are symmetrical.

If neither QuickBuy nor EZShop return a quote for the requested item, then control links \(l_6\) and \(l_7\) are set to true. This results in the execution of the central invocation operation \(s_0!\text{notFound}\), i.e. the client is informed that no quotes were found for the item description \(\text{desc}\).

![Figure 1: The QuoteComparer Service](image)

3. Syntax of SeB

3.1. Basic Sets

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

<table>
<thead>
<tr>
<th>Set</th>
<th>Description</th>
<th>Ranged over by</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatVal</td>
<td>Data Values</td>
<td>$u, u', u_i, \ldots$</td>
</tr>
<tr>
<td>SrvVal</td>
<td>Service Locations</td>
<td>$\pi, \pi', \pi_i \ldots$</td>
</tr>
<tr>
<td>SesVal</td>
<td>Session Ids</td>
<td>$\alpha, \alpha', \alpha_i \ldots \beta \ldots$</td>
</tr>
<tr>
<td>ExVal</td>
<td>Exchangeable Values</td>
<td>$w, w', w_i, \ldots$</td>
</tr>
<tr>
<td>LocVal</td>
<td>All Locations</td>
<td>$\delta, \delta', \delta_i \ldots$</td>
</tr>
<tr>
<td>Val</td>
<td>All Values</td>
<td>$v, v', v_i \ldots$</td>
</tr>
</tbody>
</table>

Table 1: Values

Table 1 presents the various sets of values used in SeB. Data Values (DatVal), Service Locations (SrvVal) and Sessions Ids (SesVal) are ground sets. The set of Exchangeable Values ExVal is given by $ExVal = DatVal \cup SrvVal$, which means that both data values and service locations can be passed between services. Hence, SeB services may dynamically discover other services and then interact with them. The set of all locations LocVal is given by $LocVal = SrvVal \cup SesVal$. Hence, session ids are used to locate service instances. The set of all values Val is given by $Val = ExVal \cup SesVal$.

<table>
<thead>
<tr>
<th>Set</th>
<th>Description</th>
<th>Ranged over by</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatVar</td>
<td>Data Variables</td>
<td>$y$</td>
</tr>
<tr>
<td>SrvVar</td>
<td>Service Location Variables</td>
<td>$p_0, p, p', p_i \ldots$</td>
</tr>
<tr>
<td>ExVar</td>
<td>Variables for Exchangeable Values</td>
<td>$x, x', x_i \ldots$</td>
</tr>
<tr>
<td>SesVar</td>
<td>Session Variables</td>
<td>$s_0, s, s', s_i \ldots q, r, r_0 \ldots$</td>
</tr>
<tr>
<td>Var</td>
<td>All Variables</td>
<td>$z, z', z_i \ldots$</td>
</tr>
</tbody>
</table>

Table 2: Variables

Table 2 presents the various sets of variables. We can note two distinguished variables, $p_0$ and $s_0$: $p_0$, is the service location variable that is dedicated to holding a service’s own location; $s_0$ is a session variable for accepting sessions initiated by clients. The use of $p_0$ and $s_0$ will be described in detail later on in the paper. The set of variables for exchangeable values ExVar and the set of all variables Var are defined by: $ExVar = DatVar \cup SrvVar$ and $Var = ExVar \cup SesVar$.  

8
Table 3: Type Identifiers

<table>
<thead>
<tr>
<th>Set</th>
<th>Description</th>
<th>Ranged over by</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatTyp</td>
<td>Data Type Identifiers</td>
<td>$t, t', t_i, \cdots$</td>
</tr>
<tr>
<td>SrvTyp</td>
<td>Service Type Identifiers</td>
<td>$P, P', P_i, \cdots$</td>
</tr>
<tr>
<td>Typ</td>
<td>Data or Service Type Identifiers</td>
<td>$X, X', X_i, \cdots$</td>
</tr>
</tbody>
</table>

Table 3 presents the various type identifiers of SeB. DatTyp and SrvTyp are ground sets. As it will be shown later, service types are behavioural types. A service type is a special behaviour type that prescribes the behaviour provided by a service.

Table 4: Miscellaneous Sets

<table>
<thead>
<tr>
<th>Set</th>
<th>Description</th>
<th>Ranged over by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Operation Names</td>
<td>$op, op', op_i, \cdots$</td>
</tr>
<tr>
<td>Lnk</td>
<td>Control Links</td>
<td>$l_1, l_2, l_i, \cdots$</td>
</tr>
<tr>
<td>$2^{Lnk}$</td>
<td>subsets of Lnk</td>
<td>$L_1, L_2, l_i, \cdots$</td>
</tr>
<tr>
<td>Jcd</td>
<td>Join Conditions</td>
<td>$e_1, e', e_i, \cdots f, \cdots$</td>
</tr>
</tbody>
</table>

Table 4 presents the other basic sets. Op is the set of operation names (as seen in messages, send and receive activities, and in sessions types), Lnk is the set of all control links, $2^{Lnk}$ are subsets of links (used to identify source links and target links), and Jcd is the set of join conditions, i.e., boolean expressions over control links.

Reconsidering our previous example, EZShop and QuickBuy are service location variables (elements of set SrvVar, with values taken in the set of service locations SrvVal); desc is a data variable (with values in the set DatVal); $s_0, s$ and $q$ are session variables (elements of SesVar, taking their values in the set of session ids SesVal); $l_1, l_2$ and $l_3$ are control links; and finally, the expression $(l_3 \text{ and } l_2)$ or $l_3$ is a join condition.

3.2. Syntax of SeB Activities

BPEL has its textual syntax defined in XML. Hence, XML would have been the most appropriate metalanguage for encoding SeB’s syntax. However, for the purpose of this paper, we have adopted a syntax based on records (in the style of [10]) 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 that have the following predefined fields: knd
which identifies the kind of the activity, \( \text{beh} \), which gives its behaviour, \( \text{src} \) (respectively \( \text{tgt} \)), which contains a declaration of the set of control links for which the activity is the source (respectively target), \( \text{jcd} \) which contains the activity’s join condition, i.e., a boolean expression over control link names (those given in field \( \text{tgt} \)). Moreover, the \( \text{flow} \) activity has an extra field, \( \text{lnk} \), 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 \( \text{act} \) is an activity, then \( \text{act.\text{beh}} \) yields its behaviour. For example: a \( \text{flow} \) activity is given by the record \( \langle \text{knd} = \text{FLO}, \text{beh} = \text{my\_behavior}, \text{tgt} = L_t, \text{src} = L_s, \text{jcd} = e, \text{lnk} = L \rangle \) where \( L_t, L_s \) and \( L \) are sets of control link names, and \( e \) is a boolean expression over control link names. Finally, for the sake of conciseness, we will often drop field names in records and instead we will associate a fixed position to each field. Hence, the \( \text{flow} \) activity given above becomes: \( \langle \text{FLO, my\_behavior, L_t, L_s, e, L} \rangle \).

We let \( \text{ACT} \) be the set of all activities and \( \text{act} \) a running element of \( \text{ACT} \), the syntax of activities is given in table 5.

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

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

Table 5: Syntax of SeB activities

Note that in the production rule for \( \text{flo} \), “|” is to be considered merely as a token separator. It is preferred over comma because it better conveys the intended meaning of the \( \text{flo} \) activity, which is to be 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 \( \text{seq} \), \( \text{pic} \) and \( \text{rep} \) to convey their appropriate intuitive meanings. Note that “|” and “+” are commutative. As a final note, the number \( n \) appearing
in the rules for seq, flo and pic is such that \( n \geq 1 \).

Returning to Figure 1, the syntax of the central INV activity is given by the following expression: \( \langle \text{INV}, s_0!\text{NotFound}, \{l_6, l_7\}, \emptyset, l_6 \text{ and } l_7 \rangle \), and the syntax of the seq activity at the top left of the example is given by the following expression:

\[
\langle \text{SEQ}, \langle \text{SES}, s\@EZshop, \emptyset, \emptyset, \text{true} \rangle; \langle \text{INV}, s!\text{getQuote(desc)}, \emptyset, \emptyset, \text{true} \rangle; \langle \text{PIC}, \langle \text{REC}, s?\text{quote(quote1)}, \emptyset, \{l_1, l_4\}, \text{true} \rangle \\
+ \langle \text{REC}, s?\text{noQuote}, \emptyset, \{l_6, l_8\}, \text{true} \rangle, \emptyset, \emptyset, \text{true} \rangle, \emptyset, \emptyset, \text{true} \rangle
\]

**Note:** Syntax simplification - when an activity has no incoming and no outgoing control links, we will omit its encapsulating record and represent it just by its behaviour. Hence, for example, we will write \( \text{act; act' instead of } \langle \text{SEQ}, \text{act; act'}, \emptyset, \emptyset, \text{true} \rangle \), and we will write \( s!\text{op}(x) \) instead of \( \langle \text{INV}, s!\text{op}(x), \emptyset, \emptyset, \text{true} \rangle \).

### 3.3. Well-formed SeB activities

Not all activities derived from the syntax given in table 5 are valid. Some further constraints need to be fulfilled and they are defined in the present section. For an activity to be well-formed, it should be well-structured and all its rep subactivities should be rep-well-formed. In the sequel of this section, we first formalize the notion of subactivities of an activity, then we define a precedence relation, relation \( \text{pred} \), on subactivities, then we define rep-well-formed and well-structured activities. Finally we define well-formed activities.

**Definition 3.1.** Subactivities - For an activity \( \text{act} \), we define two sets of subactivities, \( \widehat{\text{act}} \) and \( \widehat{\widehat{\text{act}}} \), as follows:

- \( \widehat{\text{act}} \triangleq \{\text{act}\} \) if \( \text{act} \) is an atomic activity, else \( \widehat{\text{act}} \triangleq \{\text{act}\} \cup \widehat{\text{act}.\text{beh}} \)
- \( \widehat{\widehat{\text{act}}} \triangleq \emptyset \) if \( \text{act} \) is an atomic activity, else \( \widehat{\widehat{\text{act}}} \triangleq \widehat{\text{act}.\text{beh}} \)
Informally, \( \hat{\text{act}} \) is the set of activities transitively contained in \( \text{act} \) and \( \tilde{\text{act}} \) is the set of activities strictly and transitively contained in \( \text{act} \).

**Definition 3.2.** Precedence relation between activities - For an activity \( \text{act} \), we define relation \( \text{pred} \) on the set \( \hat{\text{act}} \) as follows:

\[
\text{act}_1 \text{ pred } \text{act}_2 \iff \left( \text{act}_1.\text{src} \cap \text{act}_2.\text{tgt} \neq \emptyset \right) \text{ or } \left( \exists \text{seq} \in \hat{\text{act}} \text{ with seq.beh} = \cdots \text{act}_1; \cdots; \text{act}_2 \cdots \right)
\]

Informally, relation \( \text{pred} \) implies that either \( \text{act}_1 \) precedes \( \text{act}_2 \) in some \( \text{seq} \) activity or \( \text{act}_1 \) has at least one outgoing control link targeting \( \text{act}_2 \).

**Definition 3.3.** \( \text{rep} \)-well-formed activity - A SeB activity \( \text{act}_0 \) is \( \text{rep} \)-well-formed iff any activity \( \text{rep} = (\text{REP}, \text{do pic}_1 \text{ until pic}_2, L_1, L_n, e) \) of \( \hat{\text{act}}_0 \) satisfies the following 3 conditions:

1. \( \text{pic}_1.\text{tgt} = \text{pic}_2.\text{tgt} = \emptyset \),
2. \( \text{pic}_1.\text{src} = \emptyset \),
3. \( \tilde{\text{pic}}_1.\text{src} = \tilde{\text{pic}}_1.\text{tgt} \).

Informally, (1) implies that \( \text{pic}_1 \) and \( \text{pic}_2 \) have no incoming links, (2) states that \( \text{pic}_1 \) has no outgoing links and (3) states that all control links of activities (strictly) contained in \( \text{pic}_1 \) are (strictly) internal to \( \text{pic}_1 \).

![Figure 2](image-url) Figure 2: Example of a not \( \text{rep} \)-well-formed activity

Figure 2 gives a graphical depiction of a \( \text{rep} \) activity that violates the three conditions of \( \text{rep} \)-well-formedness. Links \( l_1 \) and \( l_2 \) violate condition (1), link \( l_5 \) violates condition (2), and links \( l_3 \) and \( l_4 \) violate condition (3).

**Definition 3.4.** Well-structured activity - A SeB activity \( \text{act}_0 \) is well-structured iff the control links occurring in any activity of \( \hat{\text{act}}_0 \) satisfy the unicity, scoping and non cyclicity conditions given below, where \( \text{act}, \text{act}', \text{act}'' \) and \( \text{seq} \) are subactivities of \( \text{act}_0 \).
(i) Control links unicity - For any control link \( l \), and any pair of activities \( \text{act} \) and \( \text{act}' \):

\[
\text{if (} l \in \text{act}.\text{lnk} \cap \text{act}'.\text{lnk} \text{) or (} l \in \text{act}.\text{src} \cap \text{act}'.\text{src} \text{) or (} l \in \text{act}.\text{tgt} \cap \text{act}'.\text{tgt} \text{) then } \text{act} = \text{act}'.
\]

(ii) Control links scoping - If \( l \in \text{act}.\text{src} \) (respectively if \( l \in \text{act}.\text{tgt} \) ) then \( \exists \text{act}', \text{act}'' \) with \( \text{act} \in \widehat{\text{act}''} \), \( \text{act}' \in \widehat{\text{act}''} \), \( l \in \text{act}''.\text{lnk} \) and \( l \in \text{act}'.\text{tgt} \) (respectively \( l \in \text{act}''.\text{src} \)).

(iii) Control links non cyclicity - 1. Relation \( \text{pred} \) is acyclic, and
2. \( \forall \text{act}, \text{act}' \in \widehat{\text{act}} : \text{act}' \in \widehat{\text{act}} \Rightarrow (\text{act}.\text{src} \cap \text{act}'.\text{tgt} = \emptyset) \) and \( (\text{act}'.\text{src} \cap \text{act}.\text{tgt} = \emptyset) \)

Informally, a well-structured activity is such that all of its control links (including those in subactivities) (i) are unique (each target declaration corresponds to one and only one source declaration and vice versa), (ii) are well-scoped (i.e. are within the scope of one flow subactivity and are not “dangling”), and (iii) do not form any causality cycles, either directly (from source to target of activities) or through activities that are chained within some sequence activity or through the containment relation between activities.

Figure 3 gives a graphical depiction of an activity that violates the three conditions of well-structuredness. The twin \( l_1 \) links violate condition (i) as they are not unique, link \( l_{2A} \) is dangling and link \( l_{2B}' \)'s target is outside the scope of the flo activity hence they both violate condition (ii), and link \( l_3 \) violates condition (iii) for it creates a causality cycle. Note that “\( \rightarrow \)” is an implicit control link that defines the flow of the sequence activity.

**Definition 3.5.** Well-formed activity - an activity is well-formed iff it is both well-structured and rep-well-formed.
The QuoteComparer example from figure 1 is an example of a well-formed activity (although the control link declarations in the flo activity were not represented for brevity).

4. Semantics of SeB: from activities to control graphs

Introducing SeB’s operational semantics. The nature of BPEL is such that there is an additional layer of control over concurrent activities within a local process. This layer of control manifests itself in the form of control links and complementary join conditions (boolean expressions over an activity’s incoming control links). Compared to traditional theoretical concurrent languages such as those based on the π-calculus, this adds a layer of complexity when giving BPEL-style languages a formal semantics. Indeed, attempting to combine the evaluation of local control flow while simultaneously studying the value and message passing semantics of distributed processes will result in a very complicated semantics.

Hence, before looking at how distributed services interact, we transform (or compile) SeB activities into what we call control graphs. Control graphs are labelled transition systems that reflect the evaluation of the control flow of SeB activities but do not address values and message passing. By isolating this first step, we can study different properties and transformations of control graphs and distinguish between binding, usage, and free occurrences of variables.

In the second step of the semantics (section 5), we run service definitions, service instances, and client instances alongside each other (based on the control graphs obtained in the previous step) in what we call a service configuration and we enable communication through FIFO message queues. At this stage, the values of variables need to be taken into account as they are exchanged in messages. Hence, each local service and client definition (written in the SeB language) declares a static set of variables whose values are kept in a memory map. Variables are considered global within each local process, as SeB does not implement the BPEL scope feature that allows the definition of local variable scopes. In this second step, by adjoining a memory map to each SeB activity (represented by its control graph), we provide the value and message passing semantics of SeB activities. This semantics shows
how a dynamic configuration of services can evolve by instantiating sessions and exchanging messages.

We have found that this two-stepped approach yields a more concise semantics that facilitates the definition and proof of the property of interaction safety. Hence, below we begin with the first step in the semantics.

4.1. Definitions

4.1.1. Control Graphs

**Definition 4.1.** Observable Actions - The set, $\text{ACTIONS}$, of observable actions is defined by:

$$\text{ACTIONS} = \{ a \mid a \text{ is any action of the form: } s@p, \ ?s, \ s\circ p(x_1, \ldots, x_n) \text{ or } s?op(x_1, \ldots, x_n) \}$$

Actions $s@p$, $s\circ p(x_1, \ldots, x_n)$ and $s?op(x_1, \ldots, x_n)$ derive directly from the syntax. Action $?s$ appears only in the semantics. It represents the reception of an acknowledgement for a session initiation request $\alpha$.

**Definition 4.2.** All actions - We define the set $\text{ACTIONS}_\tau$ of all actions (ranged over by $\sigma$): $\text{ACTIONS}_\tau = \text{ACTIONS} \cup \{ \tau \}$ where $\tau$ denotes the unobservable (or silent) action.

**Definition 4.3.** Control Graphs - A control graph, $\Gamma = < G, g_0, \mathcal{A}, \rightarrow >$, is a labelled transition system where:
- $G$ is a set of states, called control states
- $g_0$ is the initial control state
- $\mathcal{A}$ is a set of actions ($\mathcal{A} \subseteq \text{ACTIONS}_\tau$)
- $\rightarrow \subseteq G \times \mathcal{A} \times G$

4.1.2. Control Link Maps

We now define the control part of activities where we consider only the values of control links. Activities are given a map that stores the running values of control links, which are initially undefined values.

**Definition 4.4.** - Control Link Maps - A Control link map $c$ is a partial function from the set of control links, $\text{Lnk}$, to the set of boolean values extended with the undefined value. $c : \text{Lnk} \rightarrow \{ \text{true, false, } \bot \}$

**Definition 4.5.** Initial Control Links Map - For an activity $\text{act}$ we define $c_{\text{act}}$, the initial control links map: $\text{dom}(c_{\text{act}}) = \{ l \in \text{Lnk} \mid l \text{ occurs in } \text{\hat{act}} \}$ and $\forall l \in \text{dom}(c_{\text{act}}) : c_{\text{act}}(l) = \bot$
Definition 4.6. Evaluation of a Join Condition - If $L$ is a set of control links, $e$ is a boolean expression over $L$ and $c$ is a control links map, then the evaluation of $e$ in the context of $c$ is written: $c ⊘ e(L)$. Furthermore, we consider that this evaluation is defined only when $\forall l \in L, c(l) \neq \bot$.

4.2. From Activities to Control Graphs: Structured Operational Semantics Rules

In Table 6, we provide the structured operational semantics (SOS) rules defining a translation from activities to control graphs. Some comments are in order concerning this table:

- the notation for value substitution in control link maps requires some explanation: $c[\text{true} / l] =_{def} c'$ where $c'(l') = c(l)$ for $l \neq l'$ and $c'(l) = \text{true}$. By abuse of notation, we also apply value substitution to sets of control links. Hence, if $\Pi$ is a set of activities, then, e.g., $c[\text{false} / \hat{\Pi}.\text{src}]$ is the substitution whereby any control link occurring as source of an activity in $\hat{\Pi}$ has its value set to $\text{false}$.

- in rule $\text{ses}$, note how a session initiation activity is in fact converted into a reception activity $\text{rec}$ that waits for message $\text{ack}()$. This way a service waits for a message confirming that the session initiation $s@p$ was accepted. This behaviour is used to make the session initiation activity atomic, and as such a session is guaranteed to be initiated at both ends before any messages are exchanged over it.

- in the rules for the repeat activity $\text{rep}$, we have introduced the implicit activity $\text{unf}$ (unfold). Its syntax is: $\text{unf} := (\text{UNF}, \text{do act then pic}_1 \text{ until pic}_2, L_s, L_t, e)$. Activity $\text{unf}$ is introduced as a result of the execution of rule $\text{REP1}$. This mirrors the unfolding of an iteration by activity $\text{pic}_1$. Rule $\text{UNF1}$ represents the execution of an action by the unfolded activity and rule $\text{UNF2}$ represents the termination of the iteration whereby $\text{unf}$ is transformed back into a $\text{rep}$ activity identical to the original one. Note how, in this transformation, all links (strictly) contained in $\text{pic}_1$ are reset to the undefined value. The $\text{rep}$ activity as a whole terminates by means of rule $\text{REP2}$ representing the activation of activity $\text{pic}_2$.

- in the rules for activity $\text{flo}$ we have dropped the field $\text{lnk}$ since its value is constant ($\text{lnk}$ is used to define a scope for control link variables).

- as a notational convention, we have used “∗” to denote a wildcard activity, as seen in the rule for dead-path elimination (DPE).
<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLO2</td>
<td>(c \triangleright e(L_i) = \text{true} ) ( (c, (\text{FLO}, \text{act}; \text{act}; \ldots; \text{act}<em>n, L_i, L_e, e)) ) ( c' \in (\text{true}</em>{\text{L}_i}; \text{nil}) )</td>
<td>( (c, (\text{FLO}, \text{act}; \text{act}; \ldots; \text{act}<em>n, L_i, L_e, e)) ) ( c' \in (\text{true}</em>{\text{L}_i}; \text{nil}) )</td>
<td>( c' \in (\text{true}_{\text{L}_i}; \text{nil}) )</td>
</tr>
<tr>
<td>UNF2</td>
<td>(c \triangleright e(L_i) = \text{true} ) ( (c, (\text{UNF}, \text{do} \cdot \text{pic} \cdot \text{until} \cdot \text{pic} \cdot \text{L}<em>i, L_i, e)) ) ( c' \in (\text{true}</em>{\text{L}_i}; \text{nil}) )</td>
<td>( (c, (\text{UNF}, \text{do} \cdot \text{pic} \cdot \text{until} \cdot \text{pic} \cdot \text{L}<em>i, L_i, e)) ) ( c' \in (\text{true}</em>{\text{L}_i}; \text{nil}) )</td>
<td>( c' \in (\text{true}_{\text{L}_i}; \text{nil}) )</td>
</tr>
</tbody>
</table>

Table 6: Structured Operational Semantics
When applied to the initial control state \((c_{\text{act}}, \text{act})\) of a well-formed activity \(\text{act}\), the SOS rules defined in Table 6 yield a first control graph, the raw control graph, that we note \(r-cg(\text{act})\). A state \(g\) in \(r-cg(\text{act})\) is a couple \((c', \text{act}')\). A transition between two such states in this control graph will be denoted by \((c', \text{act}') \xrightarrow{\text{act}} (c'', \text{act}'')\).

### 4.3. Properties of Raw Control Graphs

**Property 1.** The set of states of \(r-cg(\text{act})\) is finite.

**Proof.** We can structurally define function \(#_{ub}(\text{act})\) that gives an upper bound for the number of states generated from an activity \(\text{act}\). To obtain this upper bound we consider an empty set of control links so as to allow the maximum possible interleaving and hence generating the maximum number of states. This upper bound is structurally defined as follows:

\[
\begin{align*}
\#_{ub}(\text{nil}) &= 1 \\
\#_{ub}(\text{inv}) &= \#_{ub}(\text{rec}) = 2 \\
\#_{ub}(\text{REp, do pic}_{1} \text{ until pic}_{2}, \ldots, ) &= \#_{ub}(\text{pic}_{1}) + \#_{ub}(\text{pic}_{2}) + 1 \\
\#_{ub}(\text{flo, act}_{1} | \ldots | \text{act}_{n}, \ldots) &= (\#_{ub}(\text{act}_{1}) + 1) \times \ldots \times (\#_{ub}(\text{act}_{n}) + 1) + 1 \\
\#_{ub}(\text{pic, rec}_{1}, \text{act}_{1} + \ldots + \text{rec}_{n}, \text{act}_{n}, \ldots) &= \#_{ub}(\text{act}_{1}) + 2 + \ldots + \#_{ub}(\text{act}_{n}) + 2 + 1
\end{align*}
\]

**Property 2.** \(r-cg(\text{act})\) is free of \(\tau\) loops.

**Proof.** The only case in which a \(\tau\) loop could appear is in the \(\text{pic}_{1}\) of a repeat activity. But since \(\text{pic}\) necessarily contains an initial receive activity, any potential \(\tau\) loop is broken by this receive activity.

**Property 3.** \(r-cg(\text{act})\) is \(\tau\)-confluent, i.e.:

\[
g \xrightarrow{\text{act}} g_{1} \text{ and } g \xrightarrow{\sigma_{\text{act}}} g_{2} \Rightarrow \exists g' \text{ where } g_{1} \xrightarrow{\sigma_{\text{act}}} g' \text{ and } g_{2} \xrightarrow{\tau_{\text{act}}} g'
\]

**Proof.** (sketch) Let us consider the situation where from some state \((c, \text{act}_{0})\) we can fire two transitions:

(1) \((c, \text{act}_{0}) \xrightarrow{\tau_{\text{act}}} (c_{1}, \text{act}_{1})\) and (2) \((c, \text{act}_{0}) \xrightarrow{\sigma_{\text{act}}} (c_{2}, \text{act}_{2})\). There are two cases in which two transitions are fireable from a given state, they are in the context of a \(\text{flo}\) or of a \(\text{pic}\) activity. In the case of \(\text{pic}\), none of the two transitions can be a \(\tau\)-transition since the first actions of a \(\text{pic}\) activity must be \(\text{receive}\) actions. Since one of the actions here is \(\tau\) then there is necessarily a subactivity \(\text{flo} \in \overline{\text{act}_{0}}\) with \(\text{flo} = \langle \text{flo}, \cdots, \text{act}_{1}', \cdots | \text{act}_{2}, \cdots, L_{r}, L_{s}, e \rangle\).
where transition (1) is produced by act′_1 and transition (2) by act′_2. Since transitions (1) and (2) are not conflicting, then both sequences (1) then (2) and (2) then (1) are possible and reach the same state.

Property 4.

(i) All sink states of r-cg(act) (i.e. states with no outgoing transitions) are of the form (c, nil). (Hence sink states may differ only by their control link maps),

(ii) for any state of r-cg(act), there exists a path that leads to a sink state.

Proof. (sketch)

(i) Let us consider the non trivial case where act is not an atomic activity, and a state (c, act_0) reachable from the initial state (c_{act}, act). If act_0.beh = nil then (c, act_0) cannot be a sink state since it can make a τ-transition to (c, nil). If act_0.beh ≠ nil then surely there is an activity act′ ∈ act_0.beh which is next in line and ready for execution in the context of control link map c. This stems from the non cyclicity property which, it can be proven, is preserved along the execution path from the initial state. This means that all the activities in act that precede act′ have been either executed or cancelled and act′ is ready to be executed. Hence, (c, act_0) can make a transition (the one derived from act′), and thus cannot be a sink state.

(ii) the inspection of all SOS rules shows that for all states (c, act_0), where act_0 is not a repeat activity, there is a transition to some other state (c′, act′) where act′ is strictly (syntactically) simpler (i.e., smaller in size) that act_0. In the case of repeat, there is also always a path leading to a syntactically simpler activity, which is through the pre-empting activity pic_2. Hence, any activity has a transition to a simpler activity. nil is the simplest activity, thus all activities must reduce to nil.

4.4. Control Graph Transformations

We apply a series of transformations on raw control graphs, some are semantics preserving simplifications, while others denote a design choice in the way control graphs must execute.

The first transformation applied to r-cg(act) is τ-prioritization and results in control graph τp-cg(act). Then we apply τ-compression, resulting in control graph τc-cg(act) that is free of τ transitions. Next comes a transformation into run-to-completion semantics resulting in rtc-cg(act). Finally, strong
equivalence minimization is applied resulting in $\text{cg}(\text{act})$, a graph with a single sink state.

4.4.1. $\tau$-Priorisation and $\tau$-Compression

In [11], the authors give an efficient algorithm that, given a graph in which the $\tau$ transitions are confluent and in which there are no $\tau$ loops, prioritises $\tau$ transitions in the graph while preserving branching equivalence. Given Property 3, i.e. that the $\tau$ transitions of any $\text{r-cg}(\text{act})$ are confluent, and Property 2, i.e. there are no $\tau$ loops in any $\text{r-cg}(\text{act})$, we can apply the algorithm given in [11] and obtain a $\tau$-prioritised control graph that we call $\text{tp-cg}(\text{act})$. In such a graph, the outgoing transitions from a state are either all $\tau$ transitions, or they are all observable actions. In [11], an algorithm for $\tau$-compression is also given that leads to a $\tau$-free branching equivalent transition system. Hence, we can apply $\tau$-prioritization and then $\tau$-compression to a raw control graph $\text{r-cg}(\text{act})$ and obtain a $\tau$-free branching equivalent control graph that we call the $\tau$-compressed graph, $\text{tc-cg}(\text{act})$.

4.4.2. Applying run-to-completion semantics

Here we process control graphs so as to enforce the run-to-completion property. This property implies the following behaviour of control graphs: after a receive, all possible invocations and session initiations are executed before another message can be received.

In order to give SeB a run-to-completion semantics, we give a priority order to the transitions in a $\tau$-compressed control graph $\text{tc-cg}(\text{act})$. We consider the outgoing transitions from a state and we define the following priority order between the actions that label these transitions:

$$s@p = s!op() > s?op().$$

This means that when two or more transitions labelled with actions with different priorities are possible from a given state, then the ones having a lower priority are pruned. All states that have become unreachable from the initial state are also pruned. The graph obtained from this transformation is the run-to-completion graph, $\text{rtc-cg}(\text{act})$.

4.5. Semantics of activities

When strong equivalence minimisation is applied to control graph $\text{rtc-cg}(\text{act})$, a minimal control graph $\text{cg}(\text{act})$ is produced that has one and only sink state (because all sink states are equivalent). Henceforth, when we write $\text{cg}(\text{act})$ for any well-formed activity $\text{act}$, we will consider that we
are dealing with the $\tau$-prioritised, $\tau$-compressed, run-to-completion and minimised control graph, and we name its unique sink state $\text{term}(\text{act})$, to be referred to as the terminal state. We also adopt the following notations: $\text{init}(\text{act})$ denotes the initial state of $\text{cg}(\text{act})$; $\text{states}(\text{act})$ is the set of states of $\text{cg}(\text{act})$; $\text{trans}(\text{act})$ is the set of transitions of $\text{cg}(\text{act})$. A transition in this final control graph $\text{cg}(\text{act})$ will be denoted by $g \xrightarrow{\sigma_{\text{act}}} g'$.

4.6. Application to the QuoteComparer example

Figure 4 shows part of the control graph generated for the QuoteComparer SeB service shown in Figure 1 in which the $\tau$-prioritisation and run-to-completion semantics transformations have been applied. The computation paths between states 0 and 17 represents an example of run-to-completion. For the purpose of clarity and brevity, this control graph diverges from the SeB semantics on a few points. Firstly, $\tau$-compression was not applied. Also, the session initiation activity is not fully represented, i.e. the confirmation step is not shown, as it adds complexity to the graph and does not contribute to the comprehension of the example. Finally, the priority scheme that is illustrated is $\text{slop()} > \text{s@p} > \text{s?op()}$ (as opposed to the scheme presented in 4.4.2) because this stricter priority order generates less possibilities and hence produces a clearer illustration.

Note in the control graph of figure 4 how the paths between states 0 and 17 represent an example of run-to-completion where only immediate actions are executed before reaching state 17 which is a waiting for receptions state. Note also how this control graph contains 5 terminal states. They all correspond to couples of the form $(c_i, \text{nil})$, in accordance with Property 4. They differ only by their control link maps. For example, state 48 corresponds to the couple $(c_{48}, \text{nil})$ where $c_{48}$ is given by $c_{48}(l_6) = c_{48}(l_7) = c_{48}(l_4) = c_{48}(l_5) = \text{true}$ and where $c_{48}$ is false for the remaining links.

Let us use this control graph to illustrate the derivation of some transitions. Let us consider the transition between states 17 and 19 which is labelled with action $q?\text{noQuote()}$. State 17 is the state in which both sessions $s$ and $q$ have been started and the two invocations $s!\text{getQuote(desc)}$ and $q!\text{getQuote(desc)}$ have been sent, in whatever order (again note that states 2 to 17 clearly show the interleaving semantics of the $\text{flo}$ activity). At state 17, the value of all control links is undefined ($\bot$). Transition $17 \rightarrow 19$ corresponds to the reception of $\text{noQuote()}$ on session $q$. This transition transforms the receive activity $q?\text{noQuote()}$ to $\text{nil}$ and sets control links $l_7$
and \( l_2 \) to true; and also sets links \( l_5 \) and \( l_9 \) to false. Indeed the source activity of links \( l_5 \) and \( l_9 \) is the reception \( q?\text{Quote}(\text{quote2}) \) which is competing with \( q?\text{noQuote}() \) and is discarded as a result of the execution of the of reception of \( \text{noQuote}() \). The \( \tau \) transition \( 19 \rightarrow 24 \) represents the termination of the encapsulating \( \text{pic} \) activity. Activity \( \text{pic} \) is terminated because its behaviour
has become nil.

Figure 5 shows the minimised control graph of the QuoteComparer example. We used the CADP [12] toolset to perform some of the transformations defined in this section on the QuoteComparer example.

4.7. Free, bound, usage and forbidden occurrences of variables

Definition 4.7. Variables of an activity - For an activity \( \text{act} \) we define the set of variables occurring in \( \text{act} \): 
\[
V(\text{act}) = \{ z \mid z \text{ occurs in } \text{act} \}.
\]

Definition 4.8. 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 \( \text{act} \): 
\[
s_p, s_{?op}(\cdots, y, \cdots) \quad \text{and} \quad s_{?op}(\cdots, p, \cdots).\]

We denote \( BV(\text{act}) \) the set of variables having a binding occurrence in \( \text{act} \).
Definition 4.9. 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 $\text{act}$: $s@p$, $s@op(\cdots)$, $s@op(\cdots, p, \cdots)$ and $s@op(\cdots, y, \cdots)$. We denote $UV(\text{act})$ the set of variables having at least one usage occurrence in $\text{act}$.

Definition 4.10. Free occurrences - A variable $z \in V(\text{act})$ is said to occur free in $\text{act}$, iff there is a path in $\text{cg}(\text{act})$: $\text{init}(\text{act})$$\xrightarrow{\sigma_1} g_1, \cdots, g_{n-1}$$\xrightarrow{\sigma_n} g_n$ where $z$ has a usage occurrence in $\sigma_n$ 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 $\text{act}$.

Definition 4.11. Forbidden occurrences - $\text{op}(\cdots, p_0, \cdots)$ and $s_0@p$ are forbidden occurrences. The two distinguished names $p_0$ and $s_0$ are reserved: $p_0$ is a service’s own location, $s_0$ is implicitly attributed a session id at service instantiation time, as detailed in section 5.

5. Syntax and Semantics of Service Configurations

In this section, we describe how SeB activities can be turned into deployable services and clients. To that end we first introduce the memory maps mentioned in section 4 that are coupled to SeB activities and which contain variable declarations and initial values. We then introduce the concept of configurations of services and clients, and we give the dynamic semantics of such configurations. Finally, we use this semantics to state the property of safe interaction.

Let $M$ be a partial map from variables $\text{Var}$ to $\text{Val} \cup \{\perp\}$, the set of values augmented with the undefined value. Henceforth, we consider couples $(M, \text{act})$ where $\text{dom}(M) = V(\text{act})$. We now turn to defining the conditions that allow an activity $\text{act}$ endowed with a map $M$ to be ready for deployment.

5.1. Deployable Services and Clients

Definition 5.1. Deployable services - The couple $(M, \text{pic})$ is a deployable service iff:

- $p_0 \in FV(\text{pic})$, $s_0 \in FV(\text{pic})$
• $FV(pic) \cap SesVar = \{s_0\}$
• $pic.BEH$ is of the form $\sum s_0?op_i(\bar{x}_i); act_i$
• $dom(M) = V(pic)$
• $\forall z \in V(pic) \setminus FV(pic) : M(z) = \perp$
• $\forall z \in FV(pic) \setminus \{s_0\} : M(z) \neq \perp$

Informally, in order for a couple $(M, act)$ to be a deployable service, activity $act$ must be a $pic$ activity, $s_0$ must be its only free session variable, the initial receptions of this activity must be on session $s_0$, its location must be defined (i.e., variable $p_0$ must be set in $M$), and all its free variables, except $s_0$, must have defined values in $M$.

**Definition 5.2.** Running service instances - The running state of a service instance derived from the deployable service $(M, pic)$ is the triple $(m, pic\cdot g)$ where:

• $g \in states(pic)$
• $dom(m) = dom(M)$

Informally, a deployed service, $(M, pic)$, behaves like a factory creating a new running service instance $(m, pic\cdot g)$ each time it receives a session initiation request. $g$ is the current control state of the instance, $m$ is its current map.

**Definition 5.3.** Initial state of a service instance - The initial state of a running service instance spawned from the deployable service $(M, pic)$ is the triple $(M[\beta/s_0], pic\cdot init(pic))$ with $\beta$ fresh.

Informally, the initial state of a newly created instance is given by the triple $(M[\beta/s_0], pic\cdot init(pic))$ where $s_0$ is assigned the initial value $\beta$, a freshly created session id, and where $init(pic)$ is the initial control state.

**Definition 5.4.** Deployable pure clients - The couple $(m, act)$ is a deployable pure client iff:

• $p_0 \notin V(act)$
• $r_0 \in V(act)$ is the unique session variable used in $act$
• **act.BEH** is of the form \( r_0@p; r_0!op(x_1, \cdots, x_n); \text{act}' \)

• there are no further session initiation activities in \( \text{act}' \)

• \( \text{dom}(m) = V(\text{act}) \)

• \( \forall z \in V(\text{act}) \setminus FV(\text{act}) : m(z) = \bot \)

• \( \forall z \in FV(\text{act}) : m(z) \neq \bot \)

Informally, a deployable pure client \((m, \text{act})\) starts with a request for a new session sent to the service located at \( m(p) \), then invokes this service with operation \( r_0!op(x_1, \cdots, x_n) \), and then it engages in other interactions with this service on the sole session \( r_0 \). Hence a deployable pure client the client of just one service (\( r_0@p \) is the only session initiation action) and is not itself a service. Pure clients are needed as bootstraps for discussions about the semantics and properties of configurations of services.

**Definition 5.5.** Running client instances - At deployment, a pure client, \((m, \text{act})\), creates one unique instance given by \((m, \text{act} \cdot \text{init}(\text{act}))\). The current state of a running client instance is given by the triple \((m', \text{act} \cdot g)\) where \( g \in \text{states}(\text{act}) \) is the current control state of the instance and \( m' \), with \( \text{dom}(m') = \text{dom}(m) \), is its current map.

**Definition 5.6.** Characteristic Functions - We define a characteristic function *service* on activities by: \( \text{service}(\text{act}) = \text{true} \) iff \( \text{act} = \text{pic} \) with \( \text{pic} \) satisfying the conditions of deployable services. We can similarly also define function *client*.

### 5.2. Service configurations and running configurations

Now we turn to defining how configurations can be built from deployable services and clients. Our aim is to provide an abstraction of the communication bus that is necessary to formalize and prove the desirable properties of service configurations. In this section, we consider that service instances and their clients exchange messages through FIFO queues. We will show how session bindings can be set up in order to establish the corresponding queues.

**Definition 5.7.** Service configurations - When deployed, a set of deployable services yields a configuration noted: \((M_1, \text{pic}_1) \diamond \cdots \diamond (M_k, \text{pic}_k)\) where the symbol \( \diamond \) denotes the associative and commutative deployment operator.
which merely means that services are deployed together and share the same address space.

**Definition 5.8.** Well partnered service configuration - A service configuration \((M_1, p_{i_1}) \odot \cdots \odot (M_k, p_{i_k})\) is said to be well partnered iff:

- \(\forall i, j : i \neq j \Rightarrow M_i(p_0) \neq M_j(p_0)\)
- \(\forall i, p : M_i(p) \neq \perp \Rightarrow \exists j \text{ with } M_i(p) = M_j(p_0)\)

That is, any two services have different location addresses, and any partner required by one service is present in the set of services.

**Definition 5.9.** Message queues - \(Q\) is a set made of message queues with \(Q := \zeta \mid \zeta \odot Q\), where \(\zeta\) is an individual FIFO message queue of the form \(\zeta := \delta \leftarrow \tilde{\text{Mes}}\) with \(\tilde{\text{Mes}}\) a possibly empty list of ordered messages and \(\delta\) the destination of the messages in the queue. The contents of \(\tilde{\text{Mes}}\) depend on the kind of the destination \(\delta\). If \(\delta\) is a service location, \(\tilde{\text{Mes}}\) contains only session initiation requests of the form \(\text{new}(\alpha)\). However if \(\delta\) is a session id, then \(\tilde{\text{Mes}}\) contains only operation messages of the form \(\text{op}(\tilde{w})\).

**Definition 5.10.** Session bindings - A session binding is an unordered pair of session ids \((\alpha, \beta)\). A running set of session bindings is noted \(B\) and has the syntax \(B := (\alpha, \beta) \mid (\alpha, \beta) \odot B\). If \((\alpha, \beta) \in B\) then \(\alpha\) and \(\beta\) are said to be bound and messages sent on local session id \(\alpha\) are routed to a partner holding local session id \(\beta\), and vice-versa.

**Definition 5.11.** Active sessions - A session, identified by the session variable \(s\), is said to be active in an instance \((m, \text{act} \triangleright g)\) if \(s \in \text{dom}(m)\) and \(m(s) = \alpha \neq \perp\).

**Definition 5.12.** Pairs of session queues - The couple \((\zeta_1, \zeta_2)\) is a pair of session queues if \(\zeta_1\) and \(\zeta_2\) are queues such that:

\[
\zeta_1 = \alpha_1 \leftarrow \tilde{\text{Mes}}_1, \quad \zeta_2 = \alpha_2 \leftarrow \tilde{\text{Mes}}_2 \text{ with } (\alpha_1, \alpha_2) \in B.
\]

Informally, pairs of session queues are such that each queue targets a different end of the same active session, i.e. the two targets are bound in \(B\).

**Definition 5.13.** Exclusively active pairs of session queues - A pair of session queues is said to be exclusively active if at most one of them is active. A queue is active if it contains messages.
Definition 5.14. Running configurations - A running configuration, \( \mathcal{C} \), is a configuration made of services, service instances, client instances, queues and bindings all running in parallel and sharing the same address space:

\[
\mathcal{C} = \mathcal{C}_{\text{serv}} \circ (m_1, \text{act}_1, g_1) \circ \cdots \circ (m_k, \text{act}_k, g_k) \circ Q \circ B
\]

where \( \mathcal{C}_{\text{serv}} = (M_1, \text{pic}_1) \circ \cdots \circ (M_n, \text{pic}_n) \) is a well-partnered service configuration and \((m_1, \text{act}_1, g_1) \circ \cdots \circ (m_k, \text{act}_k, g_k) \) are service and client instances.

In other words, a service configuration is a static composition of services, while a running configuration is a possible run-time state of the former.

Note again that the operator \( \circ \) is associative and commutative therefore the order of services, instances, bindings and queues is irrelevant. Furthermore, if the sets of bindings or queues are empty, they are omitted and need not appear in the notation.

A service configuration on its own has no way to bootstrap itself. For this to happen, the configuration requires at least one pure client instance. Thus, we give the definition of initial running configurations in Def. 5.15.

Definition 5.15. Initial Running configuration - The minimal initial running configuration is:

\[
\mathcal{C} = (M_1, \text{pic}_1) \circ \cdots \circ (M_n, \text{pic}_n) \circ (m, \text{act}, g) \text{ where } \text{act}.\text{beh} = r_0 @ p; \text{act}' \text{ and } \exists i \text{ such that } m(p) = M_i(p_0).
\]

5.3. Semantics of Service Configurations

We now turn to the semantics of service configurations. Our aim is twofold: to provide a full semantics for SeB and to allow us to formalize the property that we want to assess in SeB programs. The service configuration semantics is defined using the five SOS rules in Figure 6.

Rule SES1 applies when some service instance \((m_i, \text{act}_i, g_i)\) has a session initiation transition, \(s@p\), where \(m_i(p)\) is the address of a remote service \(M_j\) such that \(m_i(p) = M_j(p_0)\). The result is that two fresh session ids are created \((\alpha, \beta)\), the first being for the local session end, and the second for the remote session end. The queue targeting the initiator is created and message \texttt{new}(\alpha) is placed in the FIFO queue targeting \(m_i(p)\).

Rule SES2 describes how a service reacts to the reception of a \texttt{new}(\alpha) message with \((\alpha, \beta) \in \mathcal{B}\). A new service instance is created with root session \(s_0\) set to the value \(\beta\). An \texttt{ack} message is sent to the session initiator at \(\alpha\) who is expecting it (see the semantics of \texttt{ses}, Section 4), and the queue \((\beta \leftarrow \varepsilon)\) is created thus allowing the initiator of the session to place its messages.
Rule INV states that when a service instance \((m, \text{act} \triangleright g)\) is ready to send an invocation message over session \(s\), then the message is appended to the queue whose target is \(\beta\) which is bound to \(m(s)\). Rule REC is symmetrical to rule INV, and applies when a service can receive a message.

5.4. Interaction-safe Service Configurations

We can now shift our focus to the definition of a desirable property for service orchestrations called interaction safety. Informally, interaction safety is verified when the following situation never occurs: a service instance 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 no matching pick or receive activity and cannot remove the message from the queue.

We first need to characterise the state of a service instance that is ready to receive on a session \(s\).
Definition 5.16. Open for reception - A state $g$ of $cg(\text{act})$ 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 $g \xrightarrow{s?op(x_1, \ldots, x_n)}_{act} g'$.

We break down the definition of interaction safety into two stages. The first addresses running configurations, and the second (which depends on the first), addresses service configurations.

Definition 5.17. One-step interaction safety - A running configuration $C = (M_1, \text{pic}_1) \cdots (M_n, \text{pic}_n) \circ (m_1, \text{act}_1 \rightarrow g_1) \cdots (m_k, \text{act}_k \rightarrow g_k) \circ Q \circ B$ is one-step interaction-safe iff for any (service or client) instance $j$, any session variable $s$, and any operation $op$ the following implication holds:

(i) $m_j(s) \leftarrow op(w_1, \ldots, w_l) \cdot \overline{Mes} \in Q$ for some values $w_1, \ldots, w_l$, and

(ii) $open(\text{act}_j, g_j, s)$

then $g_j \xrightarrow{s?op(x'_1, \ldots, x'_l)}_{act_j}$ for some variables $x'_1, \ldots, x'_l$.

Definition 5.18. Interaction-safe Service Configuration - A service configuration $C_{serv}$ is interaction-safe iff for any client instance $(m, \text{act}, g)$ and any running configuration $C$ reachable from $C_{serv} \circ (m, \text{act}, g)$, $C$ is one-step interaction-safe.

In the following section, we will introduce a typing system for SeB. We will then use typed SeB as a basis for the verification of the property of safe interaction.

6. Typed SeB

6.1. Session Types

Behavioural types associated to sessions are used to abstract the interactions between the client and the service occurring within a session; typically they take the form of finite-state automata, or finite state labelled transition systems [13]. In order to introduce session types in SeB we need the following definitions.
Definition 6.1. Action types - An action type, $\eta$, is either a send action type, written $\eta = ! op(X_1, \cdots, X_n)$, or a receive action type, written $\eta = ? op(X_1, \cdots, X_n)$, where $X_i$ is a (data or service) type identifier.

We let $\text{ACTION TYPES}$ denote the set of all action types, and will adopt the notation $\bar{X}$ to denote a vector of dimension 0 or more: $X_1, \cdots, X_n$.

6.1.1. Syntax of session types

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

$$ST ::= \Delta \quad (* \text{Terminal State} *)$$

$$| \sum_I ? op_i(\bar{X}_i); T_i \quad (* \text{Input Choice} *)$$

$$| \oplus_I ! op_i(\bar{X}_i); T_i \quad (* \text{Output Choice} *)$$

$$| \nabla \quad (* \text{Error State} *)$$

where $T_1, \cdots, T_n$ are session type identifiers that are defined using a set of equations: $E = \{ T_1 = ST_1, \cdots, T_n = ST_n \}$ and we adopt the notation $E(T_i) =_{\text{def}} ST_i$. Moreover, we assume that the operations, $op_i$, appearing in the input and output choice are all different, i.e., session types are deterministic. Hence, one can define a next function (see definition 6.2).

6.1.2. Semantics of Session Types

The semantics of session types is given through a translation to labelled transition systems with labels in $\text{ACTION TYPES}$, and defined with the 3 SOS rules given below. For a given $ST$, we denote $\text{lt}(ST)$ its associated labelled transition system.

$$\begin{align*}
\oplus_I ! op_i(\bar{X}_i); T_i & \xrightarrow{\text{Send}} T_i \\
\sum_i ? op_i(\bar{X}_i); T_i & \xrightarrow{\text{Receive}} T_i
\end{align*}$$

$$E(T) = ST, \quad ST \xrightarrow{\eta} T'$$

Session Type Identifier
It is worth noting that for any given $ST$ and for any sink state $T_s$ of $\text{lts}(ST)$ (that is, a state having no outgoing transitions), we know that by definition either $E(T_s) = \triangle$ or $E(T_s) = \nabla$.

**Definition 6.2.** Function next (recall that session types are deterministic):

$$\text{next}(T, op) = \text{def}$$

$$\text{if } (\exists \tilde{X}, T' \text{ with } T \xrightarrow{\text{op}(\tilde{X})} T' \text{ or } T \xrightarrow{?\text{op}(\tilde{X})} T') :$$

- then $T'$
- else $\nabla$

**Definition 6.3.** Service Type - The session type $ST$ is said to be a service type iff $ST$ starts with a reception and $\text{lts}(ST)$ contains at least one terminal state and contains no error state. More formally, $ST$ is a service type iff the following three conditions are met:

- $ST = \sum_i ?\text{op}_i(\tilde{X}_i); T_i$,
- $\text{lts}(ST)$ contains at least one sink state
- Any $T'$ that is a sink state of $\text{lts}(ST)$ is such that $E(T) = \triangle$.

To each service type identifier $P$ we associate, in $E$, a service type, i.e., $E(P) = ST$ where $ST$ is a service type. Figure 7 shows two examples of service types, the first is the type of both EZshop and QuickBuy, the second defines the type of the QuoteComparer service.

![Diagram](image.png)

Figure 7: Examples of Service Types
A session has two ends: one at the client side, and one at the service side. The session types declared and used at each end of a session differ in that an input on one side corresponds to an output on the other side. This is called type duality in [13].

**Definition 6.4.** Duality function - On session types, we define the duality function as follows. For a session type, $ST$, its dual $\overline{ST}$ is given by:

- $\overline{\Delta} = \Delta$, $\overline{\nabla} = \nabla$, and

- $\sum_i ? op_i (\tilde{X}_i); T_i = \bigoplus_i ! op_i (\tilde{X}_i); T_i$, $\bigoplus_i ! op_i (\tilde{X}_i); T_i = \sum_i ? op_i (\tilde{X}_i); T_i$

where for each $T_i$, we associate $\overline{T_i}$, a new session type identifier defined by $E(\overline{T_i}) = E(T_i)$. Note that $\overline{ST} = ST$.

**Definition 6.5.** Client Type - $ST$ is said to be a client type iff its dual $\overline{ST}$ is a service type.

6.2. Subtyping

We define a subtyping relation on session types that is identical to the one defined in previous work [14, 13]. Hence, our subtyping relation can be proven to be a preorder [13].

**Definition 6.6.** Subtyping relation - Relation $\text{Sub}$ is a subtyping relation iff for all $ST_1$ and $ST_2$ 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:

$$
\begin{array}{c}
ST_1 \xrightarrow{\text{Sub}} ST_2 \\
\downarrow \text{Sub} \hspace{2cm} \downarrow \text{Sub}
\end{array}
\qquad
\begin{array}{c}
ST_1 \xrightarrow{\text{Sub}} ST_2 \\
\downarrow \text{Sub} \hspace{2cm} \downarrow \text{Sub}
\end{array}
$$

where $\forall i : X_i \text{ sub } Y_i$

and where, if $X_i$ and $Y_i$ are data type identifiers, then $X_i = Y_i \iff X_i \text{ sub } Y_i$.

**Definition 6.7.** Subtyping ($\preceq$) - A session type $ST_1$ is said to be a subtype of a session type $ST_2$, written $ST_1 \preceq ST_2$, iff there exists a subtyping relation, $\text{Sub}$, such that $ST_1 \text{ sub } ST_2$.

Notice that, since our focus is on behavioural typing, we have taken a simplified approach to data typing by assimilating subtyping and type equality.
6.3. Adding types to SeB

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

$$\hat{M} : (\text{DatVar} \rightarrow \text{DatTyp}) \cup (\text{SrvVar} \rightarrow \text{SrvTyp}) \cup (\text{SesVar} \rightarrow \text{SesTyp} \cup \{\bot\}).$$

We need to redefine the notions of deployable services and of well partnered configurations, in the context of typed services.

**Definition 6.8.** Deployable typed service - A couple $(\hat{M}, \text{pic})$ is a deployable typed service iff:

- $(\hat{M}, \text{pic})$ is a deployable service
- $\text{dom}(\hat{M}) = V(\text{pic})$
- $\hat{M}(s_0) = \hat{M}(p_0)$
- $\forall s \in V(\text{pic}) \setminus \{s_0\}: \hat{M}(s) = \bot$

Informally, $(\hat{M}, \text{pic})$ is a deployable typed service if all its variables have initial types, if its session variable $s_0$ is initiated with the dual type of the provider service type, and if the initial type of all session variables other than $s_0$ is $\bot$.

**Definition 6.9.** Deployable typed client - A couple $(\hat{m}, \text{act})$ is a deployable typed client iff:

- $(\hat{m}, \text{act})$ is a deployable client with $\text{act}.\text{BEH} = r_0@p; \text{act}'$
- $\text{dom}(\hat{m}) = V(\text{act})$
- $\hat{m}(r_0) = \bot$

6.4. Well-typedness

6.4.1. Notations for typing rules

Before tackling the typing rules, we need to introduce the following notations, where $\hat{M}$ is used as a typing environment (with $\ast \in \{!, ?\}$):

$$\hat{M} \vdash z : Z \iff \hat{M}(z) = Z$$

where $z$ is a data, session or service location variable.
\[
\hat{M} \vdash g \xrightarrow{s \text{op}(X_1, \ldots, X_n)} T_{\text{types}} \text{act} g' \iff \\
\exists \text{op}, x_1, \ldots, x_n \text{ with } \hat{M} \vdash x_1 : X_1, \ldots, x_n : X_n \text{ and } g \xrightarrow{s \text{op}(x_1, \ldots, x_n)} T_{\text{act}} g'
\]

\[
\hat{M} \vdash g \xrightarrow{s \text{op}(\check{X})} T_{\text{types}} \text{act} g' \iff \exists g' \text{ with } \hat{M} \vdash g \xrightarrow{s \text{op}(\check{X})} T_{\text{act}} g'
\]

\[
\hat{M} \vdash g \xrightarrow{s \text{op}(\check{X})} T_{\text{act}} g' \iff \neg (\hat{M} \vdash g \xrightarrow{s \text{op}(\check{X})} T_{\text{act}})
\]

\[
T \xrightarrow{\eta} \iff \exists T' : T \xrightarrow{\eta} T' \quad T \xrightarrow{\eta'} \iff \neg (T \xrightarrow{\eta'})
\]

\[
T \xrightarrow{\parallel} \iff \exists \text{op}, \check{X} : T \xrightarrow{\check{\text{op}}(\check{X})} \quad T \xrightarrow{\parallel'} \iff \neg (T \xrightarrow{\parallel'})
\]

6.4.2. Typing procedure for SeB

We now consider a deployable typed service \((\hat{M}, \text{pic})\). The SOS typing rules provided in Figure 8, when applied to the typed part of this service starting from its initial state, \((\hat{M}, \text{pic} \xrightarrow{\parallel} \text{init}(\text{pic}))\), define a translation into a finite labelled transition system, called the typing graph of \((\hat{M}, \text{pic})\) and noted \(\text{tg}(\hat{M}, \text{pic})\). A running state of \(\text{tg}(\hat{M}, \text{pic})\) is given by a triple \((\hat{m}, \text{pic} \xrightarrow{\parallel} g)\) where \(\hat{m}\) is the current typing map and \(g\) the current control state of the service. We call \(\text{states}(\hat{M}, \text{pic})\) the set of states belonging to the typing graph of service \((\hat{M}, \text{pic})\).

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 corresponding session have matching transitions. In case of mismatch, the state of the session type is set to \(\nabla\), the error state, and the transition is labelled with the keyword \textit{error}. Also, the rules enforce that a session cannot be re-initiated if its current state is not \(\Delta\), the terminal state. It is possible to define a strong and a weak well-typedness property, depending on whether or not we allow sessions other than \(s_0\) to be stopped by the service before reaching the terminal state \(\Delta\). Hereafter we provide the definition of strong well-typedness.

\textbf{Definition 6.10.} Strong well-typed - A typed service, \((\hat{M}, \text{pic})\), is strongly well-typed iff:

\[
\]
### Type-safe rules

\[
\begin{align*}
g \xrightarrow{\text{s} \hspace{1cm} \text{top}_p} g' & \quad \hat{m} \vdash p : P & (\hat{m} \vdash s : \Delta \text{ or } \hat{m} \vdash s : \perp) \\
\hat{m}, \text{pic} \triangleright g \xrightarrow{\text{s} \hspace{1cm} \text{top}_p} (\hat{m} [\gamma], \text{pic} \triangleright g') \\
g \xrightarrow{\text{?} \hspace{1cm} s} g' & \quad ?s \\
\hat{m}, \text{pic} \triangleright g \xrightarrow{\text{?} \hspace{1cm} s} (\hat{m}, \text{pic} \triangleright g') \\
\hat{m} \vdash g \xrightarrow{\text{s} \hspace{1cm} \text{top}(\bar{X})} g' & \quad \hat{m} \vdash s : T \quad T \xrightarrow{\text{?} \hspace{1cm} \text{op}(\bar{X})} T' & ?? \\
\hat{m}, \text{pic} \triangleright g \xrightarrow{\text{s} \hspace{1cm} \text{top}(\bar{X})} (\hat{m} [\gamma], \text{pic} \triangleright g') \\
\hat{m} \vdash g \xrightarrow{\text{s} \hspace{1cm} \text{top}(\bar{X})} g' & \quad \hat{m} \vdash s : T \quad T \xrightarrow{\text{?} \hspace{1cm} \text{op}(\bar{X})} T' & ?? \\
\hat{m}, \text{pic} \triangleright g \xrightarrow{\text{s} \hspace{1cm} \text{top}(\bar{X})} (\hat{m} [\gamma], \text{pic} \triangleright g') \\
\end{align*}
\]

### Type-unsafe rules

\[
\begin{align*}
g \xrightarrow{\text{s} \hspace{1cm} \text{top}_p} g' & \quad \hat{m} \vdash p : P & (\hat{m} \vdash s : \Delta \text{ or } \hat{m} \vdash s : \perp) \\
\hat{m}, \text{pic} \triangleright g \xrightarrow{\text{?} \hspace{1cm} \text{top}_p} (\hat{m} [\gamma], \text{pic} \triangleright g') \\
\hat{m} \vdash g \xrightarrow{\text{?} \hspace{1cm} \text{top}(\bar{X})} g' & \quad \hat{m} \vdash s : T \quad T \xrightarrow{\text{?} \hspace{1cm} \text{op}(\bar{X})} T' & ?? \\
\hat{m}, \text{pic} \triangleright g \xrightarrow{\text{?} \hspace{1cm} \text{top}(\bar{X})} (\hat{m} [\gamma], \text{pic} \triangleright g') \\
\hat{m} \vdash \text{open}(\text{pic}, g, s) \vdash T & \quad T \xrightarrow{\text{?} \hspace{1cm} \text{op}(\bar{X})} T' & ?? \\
\hat{m}, \text{pic} \triangleright g \xrightarrow{\text{?} \hspace{1cm} \text{top}(\bar{X})} (\hat{m} [\gamma], \text{pic} \triangleright g') \\
\hat{m} \vdash \text{open}(\text{pic}, g, s) \vdash T & \quad T \xrightarrow{\text{?} \hspace{1cm} \text{op}(\bar{X})} T' & ?? \\
\hat{m}, \text{pic} \triangleright g \xrightarrow{\text{?} \hspace{1cm} \text{top}(\bar{X})} (\hat{m} [\gamma], \text{pic} \triangleright g') \\
\end{align*}
\]
\[ \forall (m, \text{pic} \triangleright g) \in \text{states}(\hat{M}, \text{pic}), \forall s \in \text{dom}(\hat{M}) : \hat{m}(s) \neq \nabla \]

\[ \forall (m, \text{pic} \triangleright g) \in \text{states}(\hat{M}, \text{pic}) : \text{if } (m, \text{act} \triangleright g) \text{ is a sink state of } t\hat{g}(\hat{M}, \text{pic}) \text{ then } g = \text{term}(\text{pic}) \text{ and } \forall s \in \text{dom}(\hat{M}) : \hat{m}(s) \in \{\nabla, \perp\} \]

In the following section we prove the property of interaction safety for well-typed service configurations.

7. Well-typed service configurations and their properties

This section addresses the property of interaction safety. We first define typed and well-typed service configurations. Then we formalise the theorem according to which well-typed service configurations (Def 7.2) are interaction safe (Def 5.18). Then, in order to specify how well-typed configurations evolve, we give a new operational semantics for typed service configurations.

We then prove the theorem of interaction safety in two steps. First, we extend the notion of well-typedness to running configurations of services. We then prove the property of subject reduction, i.e that well-typedness is maintained when we derive a transition from a well-typed running configuration of services. Secondly, we show that the property of one-step interaction safety (Def 5.17) is valid for any well-typed running configuration. We conclude by proving our theorem, i.e that well-typed service configurations are interaction-safe.

7.1. Typed and well-typed service configurations

**Definition 7.1.** Typed Service Configuration - A typed service configuration is a configuration made of deployable typed services: \((\hat{M}_1, \text{pic}_1) \circ \ldots \circ (\hat{M}_n, \text{pic}_n)\).

**Notation.** If \(\hat{C} = (\hat{M}_1, \text{pic}_1) \circ \ldots \circ (\hat{M}_n, \text{pic}_n)\) is a typed service configuration, then \(\hat{C}\) denotes its untyped projection, i.e. \(\hat{C} = (M_1, \text{pic}_1) \circ \ldots \circ (M_n, \text{pic}_n)\)

**Definition 7.2.** Well-typed Service Configuration - A typed service configuration \((\hat{M}_1, \text{pic}_1) \circ \ldots \circ (\hat{M}_n, \text{pic}_n)\) is well-typed iff:

- \((M_1, \text{pic}_1) \circ \ldots \circ (M_n, \text{pic}_n)\) is a well-partnered service configuration
- \(\forall i, (\hat{M}_i, \text{pic}_i)\) is strongly well typed,
- \(\forall i, j, p : \hat{M}_i(p) = M_j(p_0) \Rightarrow \hat{M}_j(p_0) \preceq \hat{M}_i(p)\) (each provided type is a subtype of the corresponding required type).
7.2. Interaction Safety

We can now state the theorem of interaction safety.

**Theorem.** If $\hat{C}_{\text{serv}}$ is a well-typed service configuration then $\hat{C}_{\text{serv}}$ is an interaction-safe service configuration.

7.3. Semantics of typed service configurations

We now define typed message queues, which are similar to the queues defined in Def. 5.9, except that values are decorated with their types.

**Definition 7.3.** Typed message queues - A typed message queue is a queue in which each element is of the form $\text{op}(w_1:X_1, \cdots, w_n:X_n)$.

**Definition 7.4.** Typed running configurations - A typed running configuration is a configuration made of deployable typed services, typed service instances, typed client instances, typed queues, and bindings: $\hat{C} = (\hat{M}_1, \text{pic}_1) \cdots (\hat{M}_n, \text{pic}_n) \circ (\hat{m}_1, \text{act}_1 \triangleright g_1) \cdots (\hat{m}_k, \text{act}_k \triangleright g_k) \circ Q \circ B$.

Figure 9 presents the rules that define how a running configuration of services and instances can evolve, and also specifies how the type decorations can evolve based on the semantics of their untyped counterparts. These rules are a type-decorated version of the rules from figure 6, and do not add additional constraints to the execution of SeB programs.

7.4. Subject reduction and one-step interaction safety of running configurations

We define the recursive function $\text{consume}$ that describes how the state of the type of a session evolves when messages from a FIFO queue targeting the session end are consumed.

**Definition 7.5.** Function $\text{consume}$ -

\[
\text{consume}(T, \widetilde{\text{Mes}}) =_{\text{def}} \begin{cases} 
\text{if } T = \nabla : \nabla \\
\text{if } \widetilde{\text{Mes}} = \varepsilon : T \\
\text{if } \widetilde{\text{Mes}} = \text{op}(w_1:X_1, \cdots, w_n:X_n) \cdot \widetilde{\text{Mes}}' : \\
\quad - \text{ if } T \xrightarrow{\text{op}(X_1, \cdots, X_n)} : \text{consume}(\text{next}(T, \text{op}), \widetilde{\text{Mes}}') \\
\quad - \text{ else } : \nabla
\end{cases}
\]
Figure 9: SOS rules for typed running configurations
This function is used to study the state of types in running configurations by emptying the queues targeting each session end. Indeed, to evaluate the well-typedness of a configuration at any point in time, messages that are buffered in queues must be consumed in order to check the compatibility of the resulting types at each end of a session. Hence the function is used in condition (6) of well-typed running configurations (Def 7.6).

7.4.1. Well-typed running configurations

We now generalise the notion of well-typedness from service configurations to running configurations.

**Definition 7.6.** Well-typed running configurations -

Let $\hat{C} = \hat{C}_{\text{serv}} \diamond (\hat{m}_1, \text{act}_1 \triangleright g_1) \cdot \cdots (\hat{m}_k, \text{act}_k \triangleright g_k) \diamond Q \diamond B$

with $\hat{C}_{\text{serv}} = (M_1, \text{pic}_1) \cdots (M_n, \text{pic}_n)$ a typed running configuration. $\hat{C}$ is a well-typed running configuration iff it satisfies the following conditions.

1. $\hat{C}_{\text{serv}}$ is a well-typed service configuration.

2. For any instance $(\hat{m}_i, \text{act}_i \triangleright g_i)$ if $\text{service}(\text{act}_i)$, then there is a unique service $(M_j, \text{act}_j)$ with $\hat{m}_i(p_0) = M_j(p_0)$ and we define $j = \text{def parent}(i, \hat{C})$.

3. For any instance $(\hat{m}_i, \text{act}_i \triangleright g_i)$ and any location variable $p$ of that instance with $\hat{m}_i(p) \neq \bot$, there exists a unique service $(M_j, \text{pic}_j)$ with $\hat{m}_i(p) = M_j(p_0)$ and we define $j = \text{def provider}(i, p, \hat{C})$.

4. For any instance $(\hat{m}_i, \text{act}_i \triangleright g_i)$ and for any active session $s$ (Def. 5.11):
   - $\exists! \beta$ with $(\hat{m}_i(s), \beta) \in B$
   - $s \neq s_0 \Rightarrow$ there exists a unique service instance $(\hat{m}_j, \text{act}_j \triangleright g_j)$ with $\hat{m}_j(s_0) = \beta$ and we define $j = \text{def server}(i, s, \hat{C})$
   - $s = s_0 \Rightarrow$ There exists a unique instance $(\hat{m}_j, \text{act}_j \triangleright g_j)$, and $\exists! q$ with $q \in \text{dom}(\hat{m}_j)$ and $\hat{m}_j(q) = \beta$

Informally, every active session is tied at one end to a single instance for which the session is the root session, and is tied at the other end to a single instance for which $s$ is not the root session.

5. For any instance $(\hat{m}_i, \text{act}_i \triangleright g_i)$:
• if service(act) with \((\hat{M}_j, \hat{p}c_j)\) the parent of \(i\), (and so \(act = \hat{p}c_j\)), then \((\hat{m}_i, act_i \triangleright g_i) \in states(\hat{M}_j, \hat{p}c_j)\).

• if client(act) then \((\hat{m}_i, act_i \triangleright g_i) \in states(\hat{m}_i[\hat{p}c_0], act_i)\)

(6) For any instance \((\hat{m}_i, act_i \triangleright g_i)\), for any active session \(s \in dom(\hat{m}_i)\) such that \(j = \text{server}(i, s, \hat{C})\), \(\hat{m}_j(s_0) = \beta\) and \(\hat{m}_i(s) = \alpha\), \((\beta \leftrightarrow \hat{M}_{\hat{m}1}) \in Q\) and \((\alpha \leftrightarrow \hat{M}_{\hat{m}2}) \in Q\), \(\exists \ T_1, T_2\) such that:

\[\text{consume}(\hat{m}_j(s_0), \hat{M}_{\hat{m}2}) = T_2 \ (T_2 \neq \nabla),\]
\[\text{consume}(\hat{m}_i(s), \hat{M}_{\hat{m}1}) = T_1 \ (T_1 \neq \nabla),\]
with \(T_2 \preceq T_1\)

Informally, if any pending messages in queues in either direction are consumed by the target session type, then the resulting pair of required and provided types remain compatible.

(7) For any instance \((\hat{m}_i, act_i \triangleright g_i)\), \(\forall p \in dom(\hat{m}_i)\) : \(j = \text{provider}(i, p, \hat{C}) \Rightarrow \hat{M}_j(p_0) \preceq \hat{m}_i(p)\)

(8) All the pairs of session queues in \(\hat{C}\) are exclusively active (Def. 5.13).

**Definition 7.7.** Well-typed initial running configurations - A service configuration on its own has no way to bootstrap itself. For this to happen, the configuration requires at least one pure client instance. The minimal well-typed initial running configuration is: \(\hat{C} = \hat{C}_{\text{serv}} \circ Q \circ B \circ (\hat{m}, act, g)\) where:

- \((\hat{m}, act, g)\) is a pure client instance such that \(act.beh = r_0@p; act'\) and \(\exists i\) such that \(\hat{m}(p) = \hat{M}_i(p_0)\) and \(\hat{M}_i(p_0) \preceq \hat{m}(p)\)
- \(\hat{C}_{\text{serv}}\) is a well-typed service configuration.
- The sets \(B\) and \(Q\) are both empty (i.e. there are no active sessions).

**7.4.2. Subject Reduction**

**Lemma 1.** If \(\hat{C}\) is a well-typed running configuration and \(\hat{C} \rightarrow \hat{C}'\), then \(\hat{C}'\) is also a well-typed running configuration.

**Proof.** (sketch)

The proof is done by starting off with a well-typed running configuration \(\hat{C}\). We then individually study the outcomes of the execution of each rule.
from figure 9. We call the new configuration reached \( \hat{C}' \). We then check \( \hat{C}' \) against conditions (1) through to (8) from the definition of well-typed running configurations and show that they all remain true, hence proving the property of subject reduction for well-typed running configurations. The detailed proof is available in Appendix A.

7.4.3. One-step interaction-safety of well-typed running configurations

Lemma 2. If \( \hat{C} \) is a well-typed running configuration then \( \hat{C} \) is one-step interaction-safe.

Proof. Let \( \hat{C} = (\hat{M}_1, \text{pic}_1) \cdots (\hat{M}_n, \text{pic}_n) \diamond \cdots (\hat{m}_j, \text{act}_j \cdot g_j) \cdots \diamond Q \diamond B \) be a well-typed running configuration.

Let us consider the situation in which conditions (i) and (ii) from definition 5.17 hold, augmented with types based on the equivalent transition in the typed configuration semantics (Figure 9):

(i) \( \hat{m}_j(s) \leftarrow \text{op}(w_1:X_1, \cdots, w_n:X_n) \cdot \tilde{M}es \in Q \) for some values \( w_1, \ldots, w_l \),

(ii) \( \text{open(\text{act}_j, g_j, s)} \)

- property (6) of well-typed running configurations:
  - \( \exists T' \) such that \( \text{consume}(\hat{m}_j(s), \text{op}(w_1:X_1, \cdots, w_n:X_n) \cdot \tilde{M}es) = T' \)
  - therefore \( \exists T \) such that \( \text{consume}(\hat{m}_j(s), \text{op}(w_1:X_1, \cdots, w_n:X_n)) = T \)
    - and \( \text{consume}(T, \tilde{M}es) = T' \)
  - \( \hat{m}_j(s) \xrightarrow{\text{op}(X_1, \cdots, X_n)} \)
  - instance \( j \) is well-typed so:
    - \( g_j \xrightarrow{s \text{op}(x_1, \cdots, x_n)} \)

Informally, property (6) of well-typed running configurations implies that it is possible for \( \hat{m}_j(s) \) to evolve by consuming message \( \text{op} \), and that because instance \( j \) is well-typed according to \( \hat{m}_j(s) \), it must be in a state in which it can receive message \( \text{op} \).

Hence, we have shown that the message can be correctly received and therefore we have verified the property of one-step safe-interaction for well-typed running configurations.

\( \square \)
7.5. Proof of the theorem of interaction safety

Proof. The proof of the theorem itself is straightforward: starting from a well-typed initial configuration, Lemma 1 establishes that all reachable running configurations are well-typed and Lemma 2 ensures that each reachable running configuration is one-step interaction safe.

8. Related work

The potential of sessions in programming languages has gained recognition recently, and languages such as Java [15] and Erlang [16] have been extended in order to support them. In the service orchestration community, a significant body of work looks at formal models that support sessions for services as a first-class element of the language, such as in the Service-Centered Calculus (SCC) [17], SSCC [4], CaSPiS [18], and Orcharts [19], among others. In particular, SSCC and CaSPis are process calculi inspired by the \( \pi \)-calculus and Orc. On the other hand, SeB (the language presented in this paper) is not a process calculus because it is based on a different paradigm. Indeed, SeB has its roots in the business process modelling community, namely it is a formal model of the BPEL [5] orchestration language. BPEL features control-links as an additional mechanism for specifying control-flow in concurrent orchestrations, which calculi like SSCC and CaSPis do not. On the other hand, BPEL does not feature sessions nor any form of behavioural typing. To the best of our knowledge, our work on SeB is the first that has introduced sessions into the BPEL orchestration language.

Other formalizations of BPEL have been suggested. For instance, in [20] the authors defined an algorithm to derive data-links from BPEL code by evaluating the control flow of processes as described by their control links. In our work, we have taken a step further and given an overall static semantics of variables in BPEL, which has allowed us to define the notion of a well-structured activity. BPEL models are quite often based on Petri nets [21, 22], which lend themselves well to the task of formalizing control links. By separating our semantics into two steps, we were able to propose a formal specification of BPEL with control links more concisely than with Petri nets. Other work suggests a formalization that takes into account typing of BPEL processes with WSDL descriptors [2], but does not cover behavioural typing and does not tackle BPEL control links. The session types defined in the present paper can model WSDL descriptors, and they add the possibility to model a service’s behavior. Blite [23] (a language that features a translation
to executable BPEL) uses a WSDL-based typing system, but it does not feature BPEL control links.

As explained in section 2, BPEL [5] uses correlation sets (not sessions) to relate messages belonging to a particular instance of an interaction, but we have opted to use sessions as they lend themselves well to typing. Still, there are works that have integrated correlation in formal approaches [24]. In [25], a calculus based on process algebra and enhanced with a system for correlating operations is presented and is shown to be able to reach a certain degree of BPEL-like expressiveness. Blite [23] is a formalized subset of BPEL with operational semantics that take into account correlation sets. It is worth noting that BPEL correlation sets support the execution of certain types of multiparty interactions, whereas the biparty sessions used in SeB do not. Multiparty sessions have been addressed outside the context of BPEL [7]. Work on multiparty session types [8] indicates two major difficulties: they lack the duality between each end of a session that makes biparty sessions relatively straightforward, and the effects of non-linear use of communication channels is error-prone. As advocated in [8] (albeit for the π-calculus), it is possible to project global types (for multiparty sessions) onto local processes, and these local type projections can be used for local type verification. We believe that our work on biparty sessions for BPEL could be extended to embrace multiparty sessions, and the results of previous approaches could serve as a basis for a BPEL-style solution.

Session types usually take the form of finite-state automata, sometimes borrowing concepts from process algebra. The distinction between external and internal choice corresponding to input and output messages is a source of difficulty when it comes to determining compatibility between components or services [22, 26]. Recent work expresses similar concepts concerning session types by means of a linear logic [27]. Behavioural types associated to sessions have also been studied for protocols [28] and software components [13].

The duality of a conversation (or session) has been addressed via the notion of contracts, that is, the behaviour exhibited by a service to which a client must conform [29, 30]. Duality between contracts and session types is discussed in [31]. The Petri Nets community working on BPEL semantics [32] developed a similar concept, that of operating guidelines that a service user is bound to follow in order to guarantee compatibility [33]. Several works have addressed the kinds of properties that can be guaranteed when a service is properly operated by the user [6, 34, 35] following this approach. Looking for independence from the formalism in which service orchestration
is modeled, [36] proposes instead to use temporal logic to define the provided and required interfaces of services, and to prove the consistency of service composition within this mathematical framework.

9. Conclusion and Future Work

In order to provide a basis for formal reasoning and verification of service orchestrations, we have adapted and formalized a subset of the widely adopted orchestration language BPEL. The resulting formalism, that we call SeB for Sessionized BPEL, supports sessions as first class citizens of the language. The separation of the proposed operational semantics in two steps has allowed a relatively concise semantics to be provided, compared to previous 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 have then introduced a behavioural typing system, where session types are used as a means of prescribing the correct structure of an interaction between two partner services during the fulfillment of a service. A typed 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 session 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. We benefit from this because no sent messages are misunderstood or neglected, and no deadlocks occur as there is always a path that leads to service fulfillment and service termination.

We believe that session types for BPEL-style languages could be implemented to illustrate the usefulness of our approach in real life applications. Such a tool could detect interaction errors in service orchestrations prior to deployment. Specific issues relating to implementation were not addressed in this paper and are left for future work.

The formal approach taken with SeB as presented in this paper also opens up the possibility of defining and proving other properties of Web service interactions, such as controllability and progress properties, the study of which
is left to future work.


[16] D. Mostrous, V. Vasconcelos, Session typing for a featherweight erlang, in: W. De Meuter, G.-C. Roman (Eds.), Coordination Models and Lan-


49


Appendix A. Proof of Subject Reduction

Lemma 1. If $\hat{\mathcal{C}}$ is a well-typed running configuration and $\hat{\mathcal{C}} \rightarrow \hat{\mathcal{C}}'$, then $\hat{\mathcal{C}}'$ is also a well-typed running configuration.

Proof.
Consider the following well-typed running configuration of services

\[ \hat{\mathcal{C}} = \hat{\mathcal{C}}_{\text{serv}} \odot (\hat{m}_1, \text{act}_1 \bullet g_1) \cdots (\hat{m}_k, \text{act}_k \bullet g_k) \odot Q \odot B \]

with $\hat{\mathcal{C}}_{\text{serv}} = ((\hat{M}_1, \text{pic}_1) \cdots (\hat{M}_n, \text{pic}_n))$. We will now show that if $\hat{\mathcal{C}} \rightarrow \hat{\mathcal{C}}'$ as a result of executing either of the rules for typed running configurations (figure 9), then $\hat{\mathcal{C}}'$ satisfies conditions (1) to (8) of well-typed service configurations (def. 7.6).

T-SES1: We start by looking at the case in which T-SES1 : $\hat{\mathcal{C}} \rightarrow \hat{\mathcal{C}}'$.
Let us consider the changes made in the conclusion of rule T-SES1:

- The instance $\hat{m}, \text{act} \triangleright g$ is modified to $\hat{m}, \text{act} \triangleright g'$ where $\hat{m}' = (\hat{m}[\alpha/s], \hat{m}[\hat{m}(p)/s])$
- Queue $\alpha \leftarrow \varepsilon$ is created and is targeting $\hat{m}', \text{act} \triangleright g'$
- Message $\text{new}(\beta)$ was added to the queue $(\hat{m}(p) \leftarrow \tilde{\text{Mes}})$
- Binding $(\alpha, \beta)$ is created

Now we shall evaluate the validity of the enumerated properties of well-typed running configurations against the new configuration $\hat{C}'$:

1. True for $\hat{C}'$ because $\hat{C}_{\text{serv}}$ is unchanged.
2. True since $\hat{m}(p_0)$ did not change.
3. True since no service location variable $p \in \text{dom}(\hat{m})$ has changed.
4. True because
   - The new binding $(\alpha, \beta)$ is added to $\mathcal{B}$
   - Because the message $\text{new}(\beta)$ was added to the queue $(\hat{m}(p) \leftarrow \tilde{\text{Mes}})$ and because property (3) tells us that $p$ corresponds to an existing service provider’s address.
   - There were no other changes to session variables.
5. True because
   - If $(\hat{m}, \text{act} \triangleright g)$ is a service instance, we let its parent be $(\hat{M}_j, \text{pic}_j)$ with $\text{act} = \text{pic}_j$, and we have $(\hat{m}, \text{act} \triangleright g) \in \text{states}((\hat{M}_j, \text{pic}_j))$. Because $(\hat{M}_j, \text{pic}_j)$ is well-typed, we have $g \xrightarrow{s_{\text{sp}}^p} g' \Rightarrow (\hat{m}, \text{pic}_j \triangleright g) \xrightarrow{s_{\text{sp}}^p} (\hat{m}[\hat{m}(p)/s], \text{pic}_j \triangleright g)$ (rule @) and $(\hat{m}[\hat{m}(p)/s], g') \in \text{states}(\hat{M}_j, \text{pic}_j)$. Since $\hat{m}[\hat{m}(p)/s] = \hat{m}'$ (rule $t$-$\text{SES1}$), we therefore have $(\hat{m}', \text{act} \triangleright g) \in \text{states}(\hat{M}_j, \text{pic}_j)$. 


If \((\hat{m}, \text{act} \triangleright g)\) is a pure client instance, this is the instance’s initial action and \(s = r_0\) is the unique session variable of the instance. \(\hat{C}\) is well-typed. The state of instance \((\hat{m}, \text{act} \triangleright g)\) is \(\text{states}[\hat{m}[\{r_0\}], \text{act}]\). The type of \(r_0\) changes according to rule \(\@\) from the typing semantics, as follows: \(\hat{m}(r_0) = \hat{m}(p)\). This is in concordance with its type in the conclusion of rule \(T-\text{SES}_1\).

6. True because no new sessions were completely initialised at both ends, and no existing active sessions evolved.

7. True by construction, because no service addresses are changed by rule \(T-\text{SES}_1\).

8. True by construction, because no pairs of session queues were changed by rule \(T-\text{SES}_1\) (and only one side of a session queue pair was created).

\(T-\text{SES}_2\): We now look at the case in which \(T-\text{SES}_2 : \hat{C} \rightarrow \hat{C}'\).

\[
\begin{array}{c}
\text{T-SES2} \\
\hline
\hat{M}(p_0) = \pi \\
\phantom{\text{T-SES2}} (\alpha, \beta) \in \mathcal{B}
\end{array}
\]

\[
\begin{array}{c}
\hat{C}_\text{sen} \circ (\hat{M}, \text{pic}) \circ (\hat{m}_1, \text{act}_1 \triangleright g_1) \circ \cdots \circ (\hat{m}_k, \text{act}_k \triangleright g_k) \circ \cdots \circ (\pi \leftarrow \text{new}(\beta) \cdot \hat{m}_\text{os}) \circ \mathcal{B} \rightarrow \\
\hat{C}_\text{sen} \circ (\hat{M}, \text{pic}) \circ (\hat{m}_1, \text{act}_1 \triangleright g_1) \circ \cdots \circ (\hat{m}_k, \text{act}_k \triangleright g_k) \circ (\hat{m}_{k+1}, \text{act}_{k+1} \triangleright g_{k+1}) \\
\phantom{\text{T-SES2}} \circ (\pi \leftarrow \hat{m}_\text{os}), (\beta \leftarrow \varepsilon), (\alpha \leftarrow \text{ack}()) \circ \mathcal{B}
\end{array}
\]

where

\[
\hat{m}_{k+1} = (\hat{M}[^{\beta/s_0}], \hat{M}), \phantom{\text{T-SES2}} \text{act}_{k+1} = \text{pic} \phantom{\text{T-SES2}} \text{and} \phantom{\text{T-SES2}} g_{k+1} = \text{init}(\text{pic})
\]

Let us consider the changes made in the conclusion of rule \(T-\text{SES}_2\) :

- A \(k + 1\)th instance \((\hat{m}_{k+1}, \text{act}_{k+1} \triangleright g_{k+1})\) is created with \(\hat{m}_{k+1} = (\hat{M}_{j}[\{s_0\}], \hat{M}_j), \text{act}_{k+1} = \text{pic}_j, g_{k+1} = \text{init}(\text{pic}_j)\)
- Queue \(\beta \leftarrow \varepsilon\) is created and is targeting \((\hat{m}_{k+1}, \text{act}_{k+1} \triangleright g_{k+1})\)
- Message \(\text{new}(\beta)\) was consumed and removed from the queue \((\pi \leftarrow \hat{m}_\text{os})\)

Now we shall evaluate the validity of the enumerated properties of well-typed running configurations against the new configuration \(\hat{C}'\):

1. True for \(\hat{C}'\) because \(\hat{C}_\text{sen}\) is unchanged.
2. \(\hat{m}_{k+1}(p_0) = \hat{M}_j(p_0)\) by construction, so true for the \(k + 1\)th instance.
3. \(\hat{C}_\text{sen}\) is a well-typed configuration of services, therefore \(\hat{M}_j\) is well-partnered. The property holds true for the \(k + 1\)th instance because \(\hat{m}_{k+1} = \hat{M}_j\).
4. True for the $k+1$th instance because

- The id $\beta$ was necessarily created upon execution of rule T-SES1 and was bound to an id $\alpha$, and there exists an instance $(\hat{m}_i, \text{act}_i \triangleright g_i)$ with $s \in \text{dom}(\hat{m}_i)$ such that $\hat{m}_i(s) = \alpha$.
- There were no other changes to session variables in the $k+1$th instance.

5. True for the $k+1$th instance because, in this case the instance is a service instance, and it is in its initial state which is the same as its parent that we call $(\hat{M}_j, \text{pic}_j)$, therefore we have by construction $(\hat{m}_{k+1}, \text{act}_{k+1} \triangleright g_{k+1}) \in \text{states}(\hat{M}_j, \text{pic}_j)$

6. True because no messages were added to or removed from any pairs of sessions queues.

7. True for the $k+1$th instance which inherits its map from its parent, say $(\hat{M}_j, \text{pic}_j)$. Because the parent belongs to a well-typed service configuration, and hence the provided types are subtypes of required types, the same property holds for the new instance.

8. True by construction, because the newly formed pair of session queues contains no messages (neither queue is active), and because no other pairs of session queues were changed by rule $T_{-SES2}$.

**T-INV:** We now look at the case in which $T-\text{INV} : \hat{C} \rightarrow \hat{C}'$.

\[
\begin{array}{c}
\hat{C}_{\text{serv}} \circ \cdots (\hat{m}, \text{act} \triangleleft g) \cdots \circ (\beta \leftarrow \hat{\text{Mes}}) \cdots \circ \text{B} \rightarrow \\
\hat{C}_{\text{serv}} \circ \cdots (\hat{m}', \text{act} \triangleright g') \cdots \circ (\beta \leftarrow \hat{\text{Mes}} \cdot \text{op}(\hat{m}(x_1) : \hat{m}(x_1), \cdots, \hat{m}(x_n) : \hat{m}(x_n)) \cdots \circ \text{B}
\end{array}
\]

where

\[
\hat{m}' = (\hat{m}, \hat{m}[\text{next}(\hat{m}(s), \text{op})/s])
\]

Let us consider the changes made in the conclusion of rule T-INV:

- The instance $(\hat{m}, \text{act} \triangleright g)$ is modified to $(\hat{m}', \text{act} \triangleright g')$ where $\hat{m}' = (\hat{m}, \hat{m}[\text{next}(\hat{m}(s), \text{op})/s])$

- Message $\text{op}(\hat{m}(x_1) : \hat{m}(x_1), \cdots, \hat{m}(x_n) : \hat{m}(x_n))$ was added to the queue $(\beta \leftarrow \hat{\text{Mes}})$
Now we shall once again evaluate the validity of the enumerated properties of well-typed running configurations against the new configuration $\hat{C}'$:

1. True for $\hat{C}'$ because $\hat{C}_\text{serv}$ is unchanged.
2. True because $p_0$ cannot be modified, so the parent remains unchanged.
3. True because no service location variables can be modified in transition $\text{T-INV}$.
4. True because no new bindings were created, and all existing bindings remain valid.
5. For the instance $(\hat{m}, \text{act} \triangleright g)$, with parent say $i$, we have:
   - $(\hat{m}, \text{pic}_i \triangleright g) \in \text{states}(M_i, g_i)$ and
   - $g \xrightarrow{\text{slop}(x_1 \ldots x_n)} g'$ (the service is in a state in which it can make this transition) and
   - $\hat{m}(s) \xrightarrow{\text{op}}(X_1 \ldots X_n) \xrightarrow{\text{next}}(\hat{m}(s), \text{op})$ (the service is well-typed and as such there is a matching type transition) with $\hat{m} \vdash x_1 : X_1, \ldots, x_n : X_n$
   \[ \Rightarrow (\hat{m}[\text{next}(\hat{m}(s), \text{op})], \text{pic}_i \triangleright g) \in \text{states}(M_i, g_i). \]
6. Suppose that $(\hat{m}, \text{act} \triangleright g) = \text{server}(\hat{m}, g)$ and so $s = s_0$, and $\hat{m}_j(q) = \beta$, and we know that:
   - $\exists T$ such that $\text{consume}(\hat{m}_j(q), \text{Mes}) = T$
   - We know that the queue targeting $(\hat{m}, \text{act} \triangleright g)$ is empty because the pair of session queues is exclusively active (property (8)) and because instance $\hat{m}$ is in a sending state.
   - The subtyping relation (6) gives us: (A) $\hat{m}(s_0) \preceq T$
   If we add message $\text{op}(\hat{m}(x_1) : \hat{m}(x_1), \ldots, \hat{m}(x_n) : \hat{m}(x_n))$ to the queue ($\beta \xrightarrow{\text{Mes}}$) then, because $\hat{m}$ is well-typed, we have:
   - $\hat{m}(s_0) \xrightarrow{\text{op}}(\hat{m}(x_1), \ldots, \hat{m}(x_n)) \xrightarrow{\text{next}}(\hat{m}(s_0), \text{op})$
   - $\hat{m}(s_0) \xrightarrow{\text{op}}(\hat{m}(x_1), \ldots, \hat{m}(x_n)) \xrightarrow{\text{next}}(\hat{m}(s_0), \text{op})$

Based on the subtyping relation (A) (above), $T \xrightarrow{\text{op}(X_1 \ldots X_n)} \text{next}(T, \text{op})$ with $\hat{m}(x_1) \preceq X_1, \ldots, \hat{m}(x_n) \preceq X_n'$.

and (A) also gives us $\text{next}(\hat{m}(s_0)) \preceq \text{next}(T, \text{op})$.

Hence property (6) is true in $\hat{C}'$.

The reasoning for the case in which $j = \text{server}((\hat{m}, \text{act} \triangleright g), s, \hat{C})$ is symmetrical.
7. True by construction, because no service addresses are changed by rule $T^{-\text{INV}}$.

8. We know that the pair of session queues $(\beta \leftarrow \tilde{\operatorname{Mes}}, \hat{m}(s) \leftarrow \tilde{\operatorname{Mes}}')$ is exclusively active. There are three points to consider:

- If both lists $\tilde{\operatorname{Mes}}$ and $\tilde{\operatorname{Mes}}'$ are empty (neither queue is active), then adding message $\operatorname{op}(\hat{m}(x_1) : \hat{m}(x_1), \ldots, \hat{m}(x_n) : \hat{m}(x_n))$ to the empty queue $\beta \leftarrow \varepsilon$ results in the pair of session queues $(\beta \leftarrow \operatorname{op}(\hat{m}(x_1) : \hat{m}(x_1), \ldots, \hat{m}(x_n) : \hat{m}(x_n)), \hat{m}(s) \leftarrow \varepsilon)$ which is also exclusively active.

- If $\beta \leftarrow \tilde{\operatorname{Mes}}$ is active and $\hat{m}(s) \leftarrow \tilde{\operatorname{Mes}}'$ is not ($\tilde{\operatorname{Mes}}' = \varepsilon$), then adding message $\operatorname{op}(\hat{m}(x_1) : \hat{m}(x_1), \ldots, \hat{m}(x_n) : \hat{m}(x_n))$ to $\beta \leftarrow \tilde{\operatorname{Mes}} \ldots \operatorname{op}(\hat{m}(x_1) : \hat{m}(x_1), \ldots, \hat{m}(x_n) : \hat{m}(x_n))$, $\hat{m}(s) \leftarrow \varepsilon$) which is exclusively active.

- Queue $\hat{m}(s) \leftarrow \tilde{\operatorname{Mes}}'$ simply cannot be active. Indeed, property (6) indicates that instance $(\hat{m}, \operatorname{act} \triangleright g)$ should be able to consume any messages in $\tilde{\operatorname{Mes}}$, and because $(\hat{m}, \operatorname{act} \triangleright g)$ is in a sending state, $\tilde{\operatorname{Mes}}$ must be empty (not active) or else (6) would not hold true.

Therefore the pair of session queues that changes as a result of rule $T^{-\text{INV}}$ remains exclusively active.

**T-REC:** We now look at the case in which $\text{T-REC} : \hat{C} \rightarrow \hat{C}'$.

\[
\begin{array}{c}
\text{T-REC} \\
\hline
\hat{C}_{\text{serv}} \circ \cdots \cdot (\hat{m}, \operatorname{act} \triangleright g) \cdots \circ (\beta \leftarrow \operatorname{op}(w_1 : X_1, \ldots, w_n : X_n) \tilde{\operatorname{Mes}}) \cdots \circ B & \rightarrow \\
\hat{C}_{\text{serv}} \circ \cdots \cdot (\hat{m}', \operatorname{act} \triangleright g') \cdots \circ (\beta \leftarrow \tilde{\operatorname{Mes}}) \cdots \circ B \\
\end{array}
\]

where $\hat{m}' = (\hat{m}[w_1/x_1, \ldots, w_n/x_n], \hat{m}[^{\operatorname{next}(\hat{m}(q))} / q])$

Let us consider the changes made in the conclusion of rule T-REC:

- The instance $(\hat{m}, \operatorname{act} \triangleright g)$ is modified to $(\hat{m}', \operatorname{act} \triangleright g')$ where $\hat{m}' = (\hat{m}[w_1/x_1, \ldots, w_n/x_n], \hat{m}[^{\operatorname{next}(\hat{m}(q))} / q])$

- Message $\operatorname{op}(w_1 : X_1, \ldots, w_n : X_n)$ was consumed and removed from queue $(\beta \leftarrow \tilde{\operatorname{Mes}})$
Now we shall once again evaluate the validity of the enumerated properties of well-typed running configurations against the new configuration \( \hat{C}' \):

1. True for \( \hat{C}' \) because \( \hat{C}_{\text{serv}} \) is unchanged.
2. Holds true because \( p_0 \) cannot be modified, so the parent remain unchanged.
3. True because although service location variables can be modified in transition \( T\text{-REC} \), all service location values present in the configuration \( \hat{C} \) correspond to existing services in \( \hat{C}_{\text{serv}} \), and no new service location values were introduced, therefore the property holds true for \( \hat{C}' \).
4. True because no new bindings were created, and all existing bindings remain valid.
5. For instance \((\hat{m}, \text{act}\triangleright g)\), with parent say \( j \), we have:
   - \((\hat{m}, \text{pic}_j \triangleright g) \in \text{states}(\hat{M}_j, g_j) \) and
   - \( g \xrightarrow{s\text{op}(x_1, \ldots, x_n)} g' \) (the service is in a state in which it can make this transition) and
   - \( \hat{m}(q) \xrightarrow{\text{op}(X_1, \ldots, X_n)} \text{next}(\hat{m}(q), \text{op}) \) (the service is well-typed and as such there is a matching type transition) with \( \hat{m} \vdash x_1:X_1, \ldots x_n:X_n \)
   \( \Rightarrow (\hat{m}[\text{next}(\hat{m}(q), \text{op})/q], \text{pic}_j \triangleright g) \in \text{states}(\hat{M}_j, g_j) \)
6. Suppose that \((\hat{m}, \text{act}\triangleright g) = \text{server}(j, q, \hat{C})\) and so \( s = s_0 \), and \( \hat{m}_j(q) = \beta \), and we know that:
   - \( \exists T \) such that \( \text{consume}(\hat{m}(s_0), \text{op}(w_1:X_1, \ldots, w_n:X_n) \cdot \hat{\text{Mes}}) = T \)
   - We know that the queue targeting instance \( j \) is empty because the pair of sessions queues is exclusively active (eq 8) and \( \hat{m} \) is in a receiving state.
   - The subtyping relation (6) gives us: \( (B) \hat{T} \preceq \hat{m}_j(q) \)
   If we remove message \( \text{op}(w_1:X_1, \ldots, w_n:X_n) \) from queue \( (\beta \leftarrow \hat{\text{Mes}}) \) then we simply reach the same subtyping conclusion, i.e:
   - \( \exists T' \) such that \( \text{consume}(\hat{m}(s_0), \text{op}(w_1:X_1, \ldots, w_n:X_n)) = T' \)
   - and \( \text{consume}(T', \hat{\text{Mes}}) = T \)
   with \( \hat{T} \preceq \hat{m}_j(q) \).
   The reasoning for the case in which \( j = \text{server}(\hat{m}, \text{act}\triangleright g), s, \hat{C}) \) is symmetrical.
7. We consider the case in which \( i = \text{server}(\hat{m}, \text{act} \triangleright g, q, \hat{C}) \) such that instance \( i \) sends a service location value \( \hat{m}_i(p) \) to instance \( (\hat{m}, \text{act} \triangleright g) \). Let \( \hat{m}_i(p) = P, \hat{m}(p') = P', s = s_0 \) and \( (\hat{m}_i(s_0), \hat{m}(q)) \in B \).

We have \( \hat{m}_i, \text{act} \triangleright g_i \xrightarrow{slop(p)} \hat{m}'_i, \text{act} \triangleright g'_i \), and
\[
(\hat{m}, \text{act} \triangleright g) \xrightarrow{q_{op}(p')} (\hat{m}', \text{act} \triangleright g')
\]
(the instances are both ready to exchange a service provider address).

Let \( l = \text{provider}(i, p, \hat{C}) \).

We want to show that \( \hat{M}_l(p_0) \preceq \hat{m}_i(p') \):

- Because instance \( m_i \) is well-typed, we know from (7) that \( \hat{M}_l(p_0) \preceq \hat{m}_i(p) \).
- Based on the subtyping relation in (6) we know that
\[
\text{consume}(\hat{m}_i(s), \hat{M}_{\hat{m}_i}(s)) \preceq \text{consume}(\hat{m}(q), \hat{M}_{\hat{m}(q)})
\]
- and hence \( \hat{m}_i(p) \preceq \hat{m}(p') \).
- Hence by transitivity \( \hat{M}(p_0) \preceq \hat{m}_j(p') \).

8. Because rule T-REC does not add any messages to queues, the active status of queues is unchanged, hence property (8) remains true.

\[\square\]