Towards a Formal Foundation to Orchestration Languages

Mirko Viroli\textsuperscript{a,1}

\textsuperscript{a} DEIS, Università degli Studi di Bologna
via Venezia 52, 47023 Cesena (FC), Italy

Abstract

We introduce a formal framework for studying the semantics of orchestration languages for Web Services. Taking BPEL4WS language as reference case study, we define syntax and semantics of a core language to derive the interactive behaviour of a business process out from a BPEL4WS specification. This is realised by developing a process algebra which, other than usual operators for choice, sequential and parallel composition, features constructs of imperative programming languages, such as iterative cycles and variable assignment. These are meant to focus on the very notion of correlation, which is exploited by BPEL4WS to define a business process as the concurrent behaviour of several process instances.

Keywords: Orchestration languages, web services, process algebra.

1 Introduction

One of the hot topics in the Web Services technology is the development of orchestration languages. Orchestration languages specify relationships and constraints over the occurrences of interactions between existing Web Services: the resulting behaviour is called a business process. Orchestration engines can then be developed that take a specification and automatically implement the new Web Service realising the business process. Relying on orchestration languages is argued to support the development of complex services in a more coherent and robust way \cite{16,20}, simplifying their analysis, design, and deployment. Some orchestration languages emerged as proposals for a standardis-
ation in the Web Services technology, such as BPEL4WS (Business Process Execution Language for Web Services) by a consortium including Microsoft, IBM, Siebel System, BEA and SAP [21], and BPML (Business Process Management Language) by Sun Microsystems, Intalio, Sterling Commerce, and CSC [2].

The notion of Web Services orchestration is generally coupled with that of Web Services choreography: as orchestration is more concerned with the coordination of different Web Services, choreography is about the interaction protocol exposed by a single Web Service or business process. The two notions, however, are not completely distinct. For instance, BPEL4WS can be used both to describe orchestration and choreography issues; Starting from a core part defining abstract processes, two small extensions are introduced to deal with executable processes modelling orchestration issues, and with business protocols modelling choreography issues. On the other hand, BPML deals with orchestration issues only, building on top of the WSCI standard for choreography [3]. As we recognise the tight relationship and overlap between the two concepts, in this paper we choose to take BPEL4WS (sometimes BPEL for short) as a case study for studying the very notion of Web Services orchestration — even though at this time it is not clear which of the two will actually become standard. However, we believe that a significant part of our study and methodology could be applied to BPML as well, the details of which are left for future researches.

On top of a number of existing and loosely coupled Web Services — possibly implemented by different vendors and residing in different organizations — an orchestration language is then used to specify a complete, robust, and coherent business process out of them. In particular, such a language is meant to define the behaviour of a Web Service coordinating existing activities, by enforcing the exploitation of given interaction protocols, supporting different work sessions — also called process instances —, supporting long-running transactions, providing facilities to manage exceptions, and so on. From a Software Engineering perspective, the general task of an orchestration Web Service is not new, but is traditionally referred to as a coordination activity [9,13,19] — namely, ruling and governing interactions of different software components. Many existing coordination models have been studied and engineered since the introduction of the archetype coordination language LINDA [12] — JavaSpaces [11], TuCSoN [22], KLAIM [10] and Manifold [1] to mention a few —, some feature and idea of which have in fact be borrowed by orchestration languages.

In general, the coordination task is a particularly crucial one in a business scenario, since it is meant to include the logics that makes a set of different
software components become a whole system — an engineered artifact reliably and effectively serving its intended purposes. Therefore, it comes not as a surprise that coordination models and languages have been the subject of a thorough formal study, with the goal of precisely describing their semantics, proving their properties, and driving the development of correct and effective implementations. Significant examples are the process algebraic approach to Linda of [6], and the formal framework for coordination services of [23]. Most existing works in this direction rely on process algebra techniques, since they provide a convenient framework for formalising coordination models, for their intrinsic ability to handle the notion of interaction and of concurrent behaviour.

A similar approach is then promising for orchestration languages as well [16,15], with the primary goal of improving their existing specifications, which — as any other prose document — might sometimes be ambiguous, too compact on relevant aspects and/or too verbose on less interesting ones. Nevertheless, in the course of providing a formal model to orchestration languages such as BPEL, one observes that process algebras have been indeed considered as a reference, but new and interesting issues have been added which are worth being investigated. Most notably, other than typical process algebraic operators for sending and receiving messages, and composing activities in a concurrent, exclusive or sequential way, imperative constructs were introduced such as iteration cycles, variables and assignment, as well as ad-hoc mechanisms to deal with separation of a business process into different and isolated process instances. So, the goal of this paper is to study the relationships between all these constructs, providing a formal account to the meaning of BPEL specifications and to the behaviour of the corresponding business processes.

This is achieved by introducing a core language for BPEL4WS focussing on a strict subset of its features, and formally describing its (abstract) syntax and its operational semantics. The goal of this study is not to study the formal properties of some feature introduced by BPEL, as e.g. in [7,5]. Rather, following the intuition behind other core languages for the mainstream technology — Featherweight Java [14] being a remarkable example — our goal is to isolate a subset of the language, providing a precise, formal description of its semantics. In particular, in this paper we identify correlation as the novel mechanism that is worth to be investigated, as it is at the core of the relationship between a business process and process instances characterising the whole orchestration idea. This core language can thus act as a description tool completing the official specification, avoiding ambiguity and underspecification, and to enable formal study of properties — e.g. typing issues or behavioural aspects. Moreover it can serve as a basis for extensions studying
either other features of BPEL4WS (such as compensation and fault handlers), or proposals for adding new mechanisms.

The remainder of this paper is organised as follows. Section 2 briefly describes the BPEL4WS orchestration language, introducing a classical example of business process used to handle shipping of items, which is used throughout the paper as a case study to show the details of the model. Section 3 explains the goals and scope of our formal model, introducing the abstract syntax we use in place of true BPEL specifications. Section 4 provides an operational semantics to the core language, taking into account the necessary execution aspects of business processes, including process instances and variable assignment. Finally, Section 5 provides concluding remarks and discusses perspectives of future works.

2 Orchestration with Business Process Languages

2.1 BPEL4WS

BPEL4WS is an XML-based specification language for describing business processes orchestrating the interaction of different, existing and possibly dynamically emerging Web Services. As such, it builds on top of the WSDL language for describing the interface of Web Services [8]. This is specified in terms of port types, actions, and messages – which, e.g., in an object oriented settings would roughly correspond to the interface types, the method names, and the method types, respectively. In particular, as far as BPEL is concerned, actions can have two kinds of interactive behaviour: one-way, when they are asynchronously invoked without waiting for any reply, or request-reply, in the case where a reply is actually expected.

A BPEL4WS specification is made of four declaration parts: the partner links, the variables, the correlation sets, and the activity realising the business process.

Partner links identify the relationship of the business process with the other Web Services it interacts to, by specifying the port types for both process/web-service and web-service/process interactions. It is worth noting that an orchestration language never directly refers to a specifically installed Web Service, but it rather refers to generic port types, which are to be bound at deployment-time or even dynamically at run-time. This abstraction is particularly relevant, since it enables also those scenarios where pools of Web Services are dynamically bound and unbound to a business process depending on load-balancing issues.

The key idea of a business process is that its global task is divided into different sessions, called process instances, each responsible for carrying on a
separate service or work for each user — whether this is a human or another service. To support this scenario, variables can be defined that can carry XML data values and messages, and which are used to define the state of each process instance. Most notably, variables can also contain partner links, that is, abstract references to other services: similarly to the $\pi$–calculus where channels are used to exchange names of channels [18], this mechanism is useful to express dynamic interconnecting structures.

Correlation sets are then introduced to identify those interactions that are pertinent to a given process instance, which is necessary in order to correctly dispatch messages between the various concurrent sessions. Each correlation set is a set of properties, which are aliases for parts of messages and are treated similarly to write-once variables: at a given time, for each process instance a number of correlation sets are active, and the values associated to all their properties uniquely identify the process instance. For instance, all the messages related to a given customer’s request must carry the same unique identifier, e.g. the order ID.

Finally, an activity is specified that describes the precise behaviour of the business process. Activities are generally built by composing basic ones through structured ones. Basic activities include the acts of sending and receiving requests and replies (\texttt{invoke}, \texttt{receive}, and \texttt{reply}), which can specify one or more existing correlation sets they must adhere to, or new correlation sets to be initialised. Among other basic activities, there are variable assignment (\texttt{assign}), synchronisation of internal concurrent activities through private links (\texttt{source} and \texttt{target}), waiting for a timeout (\texttt{wait}), and raising faults (\texttt{throw}). Structured activities realise sequential composition (\texttt{sequence}), guarded choice (\texttt{pick}), parallel composition (\texttt{flow}), iteration cycles (\texttt{while}), and multiple cases (\texttt{switch}).

2.2 A Case Study

As a reference case study, in this paper we consider the shipping service described in the official specification of BPEL4WS [21] (Section 16.1). In spite of its simplicity, this example covers most of the language features we are interested in, including correlation sets, variables, and flow control structures.

This example describes a Web Service realising a service handling the shipment of orders requested by customers, which are themselves modelled as Web Services. Two types of shipments are handled: a customer may require the orders to be atomically shipped, in which case a single ship notice callback is sent to the customer; or it may specify an uncompleted order, in which case the items are shipped in different stages, sending a different ship notice each time.
Following the schema presented in previous section, the BPEL specification defines partner links, variables, correlation sets and the business activity. Only one partner link is specified here representing the customer service: the customer invokes the service by a one-way request named `shippingRequest`, the service provides notices by executing one-way invocations to the customer, by action named `shippingNotice`. Shipping request messages are made of three parts: an `orderID` integer, a `complete` boolean specifying whether the request is to be treated atomically or not, and an `itemsTotal` integer denoting the number of items to be shipped. Shipping notice messages are made of the `orderID` integer and the `itemsCount` integer, representing the number of items currently shipped. Three variables are used in this business process: `shipRequest` for storing the received message, `shipNotice` for storing the message to be sent, and `itemsShipped` for counting the amount of items already shipped. Only one correlation set is defined which contains the property `orderID`: process instances are then uniquely characterised by the same order identifier. Finally, the activity realising the business process is of the kind shown in Figure 1. There, underlined parts do not represent actual XML code, but are rather placeholders informally describing a more complex XML code, whose details are not reported for the sake of brevity.

The algorithm realised is as follows. As the request is received, if its `complete` part flag is true a reply is immediately invoked with the same `itemsCount`. Otherwise, a `while` iteration is executed. Each time, `itemsCount` part of the `shipNotice` message is assigned to the special identifier `opaque`, which means that the result of the assignment is non-deterministic — modelling e.g. the interaction with some back-end service which is not interesting to model. Correspondingly, a message is sent to the customer notifying the number of items shipped. When this number reaches the total amount requested by the customer, the process instance terminates. In particular, all the `invoke` and `receive` activities specify are linked to the correlation set, so that interactions carrying the same `orderID` are bound to the same process instance, representing a session of work with a customer.

3 A Formal Specification of BPEL4WS

In this paper we introduce a formal model for the behaviour of orchestration services adhering to BPEL4WS specifications, providing a formal account to the existing informal specification [21]. BPEL4WS indeed features a relatively large and heterogeneous set of constructs, mechanisms and details: providing a full formalisation is out of the scope of this paper. Rather, following the usual research approach in the context of programming languages [14,6], we find
useful to focus on a smaller yet significant subset of the language, providing a core language of the features of interest.

3.1 Modelling Choices

Our goal here is to study the very notion of process instance, that is, the idea that a single business process is actually constituted by different sub-processes, each representing a different interaction session. This aspect, represented in BPEL by the mechanisms of correlation sets and of variable assignments, is at the root of the whole orchestration approach, and it thus deserves a careful investigation. Moreover, such a study gives us the chance to deepen the

```xml
<sequence>
  <receive shipRequest> ... </receive>
  <switch>
    <case shipRequest.shipComplete>
      <sequence>
        <assign> shipNotice.itemsCount:=shipRequest.itemsCount </assign>
        <invoke shipNotice> ... </invoke>
      </sequence>
    </case>
    <otherwise>
      <sequence>
        <assign> itemsShipped:=0 </assign>
        <while condition = itemsShipped < shipRequest.itemsTotal>
          <sequence>
            <assign> shipNotice.itemsCount:=opaque </assign>
            <invoke shipNotice> ... </invoke>
            <assign> itemsShipped+=shipNotice.itemsCount </assign>
          </sequence>
        </while>
      </sequence>
    </otherwise>
  </switch>
</sequence>
```

Fig. 1. BPEL specification schema for the shipping service
relationships between programming constructs typical of the process algebra approach (such as parallel composition and choice), others of imperative languages (variables and their assignments), along with the novel declarative approach to process instances introduced through correlation sets.

Thus, our core language intentionally neglects a number of interesting aspects such as timeouts, fault handlers, compensation handlers, reconfigurability of partner links, XML data representation, and so on. Including these mechanisms would not be relevant to the end of studying the aspects we are here interested in; rather, they can be introduced later on top of our language in a mostly orthogonal way, so as to provide a complete formal account of BPEL4WS.

Syntactically, the language we introduce is an abstract version of a subset of BPEL4WS: even though most constructs have a one-to-one translation into the core language, sometimes the mapping needs some accessory work which is only informally described for simplicity of treatment. The operational semantics follows the standard approach of labelled transition systems over process algebras, describing how a behaviour specification moves to another as internal computations or interactions with the environment occur. Following the abstraction used by BPEL4WS when describing partner links, we do not describe the details of interactions with other Web Services and related properties such as synchrony, directness, transport protocol, and the like. Rather, any interaction is characterised by (i) its meaning (invocation/reply), (ii) its direction (sent or received by the business process), (iii) the involved peer (the partner link), and (iv) the content (the message carried).

The language we introduce is not equipped by the necessary additional formal structure to check for the semantic correctness of a specification, e.g. verifying that variables are used before initialised, correlation sets are not initialised twice, and so on. As this would lead to a quite remarkable modelling tool, whose importance for the Web Services technology is e.g. discussed in [16], it is not analysed here for it would likely be the subject of our future research. As a result, we also avoid to take into account the parts of a specification dealing with declaration of properties, variables, correlation sets, port types, etcetera, focusing instead on their exploitation and their influence on the business process run-time behaviour.

3.2 Abstract Syntax

Let $u$ be a meta-variable ranging over integer and boolean values, $v$ over variables (of all values), and $p$ over properties (forming correlation sets). Meta-variable $l$ ranges over end-points, which can be conceptually seen as triples of a partner link, an action name, and a direction: they uniquely identify the
| Syntax | Description |
|--------|-------------|
| \( w := v \mid p \) | Variables and properties |
| \( e := \text{opaque} \mid u \mid v \mid f(e) \) | Expressions |
| \( s := 0 \mid \) | Empty specification |
| \( \text{send}(l, \overline{v}, \overline{w}) \mid \) | Sending a message |
| \( \text{recv}(l, \overline{v}, \overline{w}) \mid \) | Receiving a message |
| \( \text{while}(e)\{a\} \mid \) | Iteration structure |
| \( \text{switch}(e)\{a : a\} \mid \) | Switching structure |
| \( \text{assign}(v, e) \mid \) | Variable assignment |
| \( \text{source}(\lambda) \mid \text{target}(\lambda) \mid \) | Internal links operations |
| \( a; a \mid (a \parallel a) \mid a + a \) | Sequence, parallel composition, and choice |
| \( a := s \) | Activity |

Fig. 2. Abstract syntax

channel through which an interaction occurs. Given any meta-variable \( x \), we let \( \overline{x} \) range over sequences of \( x \) elements (such as \( \langle x_1, x_2, \ldots, x_k \rangle \)); correspondingly, \( x_i \) is the \( i^{th} \) element of one such sequence, \( x \in \overline{x} \) is used to state that \( x = x_i \) for some \( i \), and symbol \( \bullet \) is used for the void sequence. The syntax of the language we introduce is shown in Figure 2. Notice that while structured activities resembling very well known process algebraic operators are represented by the standard theoretic notation, we used for the other activities an abstract syntax resembling BPEL language.

Meta-variable \( w \) ranges over both variables and properties: their assignment to values defines the current state of a given process instance. Expressions can here be the \text{opaque} identifier — which at assignment-time may non-deterministically assume any value —, boolean or integer values, variables, or a composition of expressions through some mathematical function or operator \( f \). For instance, we allow the expression \(+ (v, 1)\) — also written using the infix notation \( v + 1 \) for clarity — to stand for the sum of the value associate to \( v \) and 1, and similarly write e.g. \( v > 10, v \geq v', v + 5 = 8 \) and so on.

Figure 3 describes how the concrete XML syntax of activities relates to the abstract one. In the left-hand side, we used \( \text{x} \) to denote the XML representation describing the corresponding right-side meta-variable \( x \) — e.g. \( a \cdot 1 \)
| Tag          | Description                                                                 | Abstract Syntax |
|--------------|-----------------------------------------------------------------------------|-----------------|
| `<sequence>` | <code>a_1 ... a_n</code>                                                   | <code>a_1; ... ;a_n</code> |
| `<flow>`     | <code>a_1 ... a_n</code>                                                   | <code>a_1 \parallel ... \parallel a_n</code> |
| `<pick>`     | <code>onMessage a_M1</code> <code>a_1</code> <code>/onMessage</code>       | <code>a_1^M : a_1 + ... + a_k^M : a_k</code> |
| `<while>`    | <code>condition= e</code> <code>a</code> <code>/while</code>              | <code>while(e)\{a\}</code> |
| `<case>`     | <code>condition= e_1</code> <code>a_1</code> <code>/case</code>           | <code>switch(e_1)\{a_1 :</code> |
| `<source>`   | <code>linkName= lambda</code>                                             | <code>source(\lambda)</code> |
| `<target>`   | <code>linkName= lambda</code>                                             | <code>target(\lambda)</code> |
| `<assign>`   | <code>from expr.=e</code> <code>to variable=v</code> <code>/assign</code> | <code>assign(v,e)</code> |
| `<invoke>`   | <code>l v</code> <code>/corr.> w</code> <code>/invoke</code>             | <code>send(l,v,w)</code> |
| `<receive>`  | <code>l v</code> <code>/corr.> w</code> <code>/receive</code>            | <code>recv(l,v,w)</code> |
| `<reply>`    | <code>l v</code> <code>/corr.> w</code> <code>/reply</code>              | <code>send(l,v,w)</code> |
| `<invoke>`   | <code>l \text{in} v \text{out}</code> <code>/corr.> w</code> <code>/invoke</code> | <code>send(l,v_{in},w_{out});</code> |
|              |                                                                             | <code>recv(l,v_{out},w_{out})</code> |

Fig. 3. Encoding of concrete syntax to abstract syntax

stands for the right-side symbol \( a_1 \), which defines an activity.

Structured activities `<sequence>` and `<flow>` are mapped to successive application of standard binary operators “;” and “\parallel” for sequential and parallel composition, respectively. Similarly, `<pick>` is mapped onto a choice operator +, guarded by the occurrence of a message exchange for each different choice. Activities `<while>`, `<link>`, `<source>`, and `<assign>` have a one-to-one mapping, while
The activities `invoke` and `reply` are used to send invocations or replies, and are mapped to the same construct `send` — their difference is meant to be recorded into the two end-points `l`, which have opposite direction. Conversely, activity `receive` is mapped onto construct `recv`. Finally, since an `invoke` specifying an input and an output message refers to a request-reply primitive, it is mapped to a sequence of a `send` and a `recv`.

The only part which needs a more accurate description should be the management of correlations, which leads to the extra-argument `w` in the activities related to message exchanges `send` and `recv`. As a first example consider a simple `invoke` with no correlations, sending a given structured message. Without loss of generality, in our abstract syntax we always see messages as made of a sequence of values, we therefore denote by `v` a variable holding messages, and by `v_i` the corresponding variable over its `i`th subpart (or field). For instance, the one-way invoke activity without correlations

```xml
<invoke ... inputVariable=varName> </invoke>
```

is expressed as `send(l, \text{v}_{\text{varName}}, \text{v}_{\text{varName}})`. The first argument is the message receiver end-point, the second is the variable that will contain the received message, while the third contains information on correlations: when no correlations exist this is exactly the same as the second argument.

Now suppose the message has type `msg`, it is made of three parts `part1`, `part2`, and `part3`, it is denoted by variable `v = \langle v_1, v_2, v_3 \rangle`, and suppose that the following correlation set is specified inside the invoke activity:

```xml
<correlations> <correlation set=cName ...> <correlations>
```

By the definitions

```xml
<correlationSet name=cName properties="p1 p2">
  <bpws:propAlias propName=p1 messageType=msg part=part1 ...>
  <bpws:propAlias propName=p2 messageType=msg part=part2 ...>
```

we define `cName` as a correlation set made of the properties `p1` and `p2`, which are linked to the first and second part of message `msg`. This correlation set will characterise those process instances exchanging messages with given values in the first and second part of the `msg` messages. The corresponding abstract syntax is now `send(l, \langle v_1, v_2, v_3 \rangle, \langle p_1, p_2, v_3 \rangle)`. In particular, the third argument `\langle p_1, p_2, v_3 \rangle` says that while the third part of the message is not related to any correlation set, for it is exactly variable `v_3`, the first two parts are properties `p_1`
and \( p_2 \), and are then correlated to the process instance. So, if in the current process instance \( p_1 \) is bound to value \( u_1 \) and \( p_2 \) to \( u_2 \), then the first and second part of the message are automatically set to \( u_1 \) and \( u_2 \). Dually, in

\[
\text{recv}(l, \langle v_1', v_2', v_3' \rangle, \langle p, v_3' \rangle);
\]

\[
\text{switch}(v_2')\{ \text{assign}(v_n^a, v_3'); \text{send}(l, \langle v_3' \rangle) ;
\]

\[
\text{assign}(v^i, 0); \text{while}(v^i < v_3')\{ \text{assign}(v_2^n, \text{opaque});
\]

\[
\text{send}(l, \langle v^n \rangle) ;
\]

\[
\text{assign}(v^i, v^i + v_2^n) \} \}
\]

Fig. 4. Abstract syntax for the shipping service

and \( p_2 \), and are then correlated to the process instance. So, if in the current process instance \( p_1 \) is bound to value \( u_1 \) and \( p_2 \) to \( u_2 \), then the first and second part of the message are automatically set to \( u_1 \) and \( u_2 \). Dually, in

\[
\text{recv}(l, \langle v_1', v_2', v_3' \rangle, \langle p_1, p_2, v_3' \rangle),
\]

only a message carrying \( u_1 \) and \( u_2 \) in the first and second part can be received by the process instance. This is why we generally denote a message sending as \( \text{send}(l, v, v') \) and similarly for message reception.

3.3 Abstract Syntax of the Shipping Service

To provide some more details on our abstract syntax, we describe the specification of the shipping service shown in Section 2.2.

Two end-points are used: \( l_n \) represents action \textit{shippingNotice} invoked on the customer, and \( l_r \) is used for invocation of action \textit{shippingRequest} by the customer. Symbol \( \overline{v} \) is used for variable \textit{shipRequest}, containing received request messages, which is made by the three parts \( v_1', v_2', v_3' \) representing \textit{orderID}, \textit{complete}, and \textit{itemsTotal}. Symbol \( \overline{v}^a \) is used for variable \textit{shipNotice}, containing the messages to send, which is made by the two parts \( v_1^n \) and \( v_2^n \), representing \textit{orderID}, and \textit{itemsCount}. Then, symbol \( v^i \) is used for variable \textit{itemsShipped}. Property \( p \) of the correlation set is used for the \textit{orderID}. The abstract syntax of the shipping service is then quite directly obtained from the BPEL specification, as shown in Figure 4. From the first glance, the reader might enjoy its compactness over the actual specification, as well as its intuitive structure.

4 Semantics

4.1 Extended Syntax

The syntax described in the previous section is the surface syntax that can be used to describe in an abstract way BPEL specifications. On the other
hand, to provide an operational semantics describing the actual behaviour of orchestrating Web Services, we also need this syntax to be able to represent the state of a business process at a given time. In particular, the notions of current variable assignment and of process instance have to enter the picture, since they play a fundamental role in the model we intend to define. So, as common practice in functional and concurrent programming languages [14,24], we extend the syntax to deal with run-time aspects. The production for symbol $a$ is changed as follows:

$$ a ::= 0 \mid (\overline{w} \mapsto \overline{u})s \mid !a \mid a \parallel P a $$

In a term $(\overline{w} \mapsto \overline{u})s$, notation $(\overline{w} \mapsto \overline{u})$ is called a store and is used to specify that in the activity specification $s$, variable or property $w_i$ is associated to value $u_i$, for any $i$. Notation $(\overline{w}, w \mapsto u)$ is used in the following with the same meaning of $(\overline{w}, w \mapsto u)$. Symbol $!a$ means that the activity $a$ can be spawned (replicated) and then executed infinite times, while operator $\parallel_P$ is for composition of process instances. In particular, while stores are used to model the current state of a process instance (both in terms of variables and correlation sets), replication is used at the top level to deal with process instances. Notice that we have two operators for parallel composition: $\parallel_P$ for composing different process instances, and $\parallel$ to compose different activities inside the same process instance: they are to kept distinct since stores do not propagate through $\parallel$ but not through $\parallel_P$. The computational model is then as follows: if $s$ is the activity syntactically obtained by a BPEL process specification, $!(\bullet \mapsto \bullet)s$ is the initial state of the business process, which spawns a new process instance with initially void store each time the first message of each work session is received.

4.2 Congruence Rules and Auxiliary Operators

As a first step towards defining operational semantics, we refine the abstract syntax by introducing the congruence rules reported in Figure 5, specifying those terms which are to be considered syntactically equivalent. These rules, giving semantic details about replication [C-SPN], parallel composition [C-PAR], flow [C-FLW], choice [C-CHO], and sequential composition [C-SEQ] are quite standardly derived from the process algebraic style (e.g. of CCS [17] and BPA [4]).

Before introducing operational semantics, we clarify the management of stores and variables. The main difference between variables and properties is tackled by introducing a partial operator $\oplus$ for modelling store update, enforcing the idea that properties are constants: into a process instance the
value of correlation sets identifies the process identity, which never changes. This operator is defined as:

\[(\overline{w} \mapsto \overline{u}) \oplus (v \mapsto u) \triangleq (\overline{w} \mapsto \overline{u})(v \mapsto u)\]

\[(\overline{w} \mapsto \overline{u})(p \mapsto u) \oplus (p \mapsto u') \triangleq (\overline{w} \mapsto \overline{u})(p \mapsto u)\]

\[(\overline{w} \mapsto \overline{u}) \oplus (p \mapsto u) \triangleq (\overline{w} \mapsto \overline{u})(p \mapsto u)\quad \text{if} \quad p \notin \overline{w}\]

The first definition states that variable assignment always update the store, the second and third that updating a property assignment is allowed only if an equivalent assignment already occurs in the store. We use this definition to enforce the idea that messages sent and received by a given process instance must conform to the correlation set, since e.g. notation \((p \mapsto u) \oplus (p \mapsto u')\) does not make sense when \(u \neq u'\).

For simplicity, the notation for stores is abused writing \(\overline{w} \mapsto \overline{u} \oplus (w'_1 \mapsto u'_1) \oplus (w'_2 \mapsto u'_2) \oplus \ldots \oplus (w'_k \mapsto u'_k)\), where \(k\) is the size of \(\overline{w}'\) and \(\overline{u}'\).

Evaluation of expression \(e\) under the store \(\overline{w} \mapsto \overline{u}\) is defined by notation \{\(\overline{w}/\overline{u}\)\}\(e\), with semantics:

\[
\{\overline{w}/\overline{u}\} u \triangleq u \quad \{\overline{w}/\overline{u}\} \overline{w}_i \triangleq u_i \quad \{\overline{w}/\overline{u}\} f(\overline{v}) \triangleq f(\{\overline{w}/\overline{u}\} \overline{v})
\]

\[
\{\overline{w}/\overline{u}\} \text{opaque} \triangleq u \quad v \notin \overline{w} \Rightarrow \{\overline{w}/\overline{u}\} v \triangleq u
\]

Orderly, values are evaluated to themselves, variables and properties to their associated value, functions by propagating the evaluation to their arguments, the opaque identifier or a variable not already initialised to any value \(u\) (non-deterministically). Notice that the notation for evaluation is again abused by using \{\(\overline{w}/\overline{u}\)\}\(\overline{v}\) to evaluate all expressions \(\overline{v}\), and is used similarly for evaluating either integer and boolean expressions.

![Fig. 5. Structural congruence rules](image-url)
\[
\begin{align*}
& a \parallel_P a' \xrightarrow{\alpha} a'' \parallel_P a' \quad \text{if } a \xrightarrow{\alpha} a'' \quad \text{[PAR]} \\
& (\overline{w} \mapsto \overline{v})(s_0 + s_1) \xrightarrow{\alpha} (\overline{w'} \mapsto \overline{v'})s'_0 \\
& \quad \text{if } (\overline{w} \mapsto \overline{v})s_0 \xrightarrow{\alpha} (\overline{w'} \mapsto \overline{v'})s'_0 \quad \text{[PCK]} \\
& (\overline{w} \mapsto \overline{v})(s_0 \parallel s_1) \xrightarrow{\alpha} (\overline{w'} \mapsto \overline{v'})(s'_0 \parallel s_1) \\
& \quad \text{if } (\overline{w} \mapsto \overline{v})s_0 \xrightarrow{\alpha} (\overline{w'} \mapsto \overline{v'})s'_0 \quad \text{[FLW]} \\
& (\overline{w} \mapsto \overline{v})(s_0; s_1) \xrightarrow{\alpha} (\overline{w'} \mapsto \overline{v'})(s'_0; s_1) \\
& \quad \text{if } (\overline{w} \mapsto \overline{v})s_0 \xrightarrow{\alpha} (\overline{w'} \mapsto \overline{v'})s'_0 \quad \text{[SEQ]} \\
& (\overline{w} \mapsto \overline{v})(\text{source}(\lambda); s \parallel \text{target}(\lambda); s') \xrightarrow{\tau} (\overline{w} \mapsto \overline{v})(s \parallel s') \\
& \quad \text{[LNK]} \\
& (\overline{w} \mapsto \overline{v})\text{while}(e)\{s\} \xrightarrow{\tau} (\overline{w} \mapsto \overline{v})s; \text{while}(e)\{s\} \\
& \quad \text{if } \overline{w}/\overline{v} e = \text{true} \quad \text{[WH]} \\
& (\overline{w} \mapsto \overline{v})\text{while}(e)\{a\} \xrightarrow{\tau} (\overline{w} \mapsto \overline{v})0 \\
& \quad \text{if } \overline{w}/\overline{v} e = \text{false} \quad \text{[NWH]} \\
& (\overline{w} \mapsto \overline{v})\text{switch}(e)\{s : s'\} \xrightarrow{\tau} (\overline{w} \mapsto \overline{v})s \\
& \quad \text{if } \overline{w}/\overline{v} e = \text{true} \quad \text{[SW]} \\
& (\overline{w} \mapsto \overline{v})\text{switch}(e)\{s : s'\} \xrightarrow{\tau} (\overline{w} \mapsto \overline{v})s' \\
& \quad \text{if } \overline{w}/\overline{v} e = \text{false} \quad \text{[NSW]} \\
& (\overline{w} \mapsto \overline{v})\text{assign}(v, e) \xrightarrow{\tau} (\overline{w} \mapsto \overline{v}) \oplus (v \mapsto \{\overline{w}/\overline{v}\} e)0 \\
& \quad \text{[ASG]} \\
& (\overline{w} \mapsto \overline{v})\text{send}(l, \overline{v}, \overline{w}_p) \xrightarrow{l!(\overline{w}/\overline{v})} (\overline{w} \mapsto \overline{v}) \oplus (\overline{w}_p \mapsto \{\overline{w}/\overline{v}\} \overline{v})0 \\
& \quad \text{[SND]} \\
& (\overline{w} \mapsto \overline{v})\text{recv}(l, \overline{v}, \overline{w}_p) \xrightarrow{l!(\overline{w}/\overline{v})} (\overline{w} \mapsto \overline{v}) \oplus (\overline{v} \mapsto \overline{v'})(\overline{w}_p \mapsto \overline{v'})0 \\
& \quad \text{[REC]}
\end{align*}
\]

Fig. 6. Operational semantics

### 4.3 Operational Semantics

As common for concurrent languages \cite{17,6,23}, operational semantics is defined here by a labelled transition system \(\langle A, \rightarrow, I \rangle\), where \(A\) is the set of activities (states of the system to model), ranged over by meta-variable \(a\), and \(I\) is the set of labels representing interactions, ranged over by meta-variable \(\alpha\) with syntax:

\[
\alpha ::= \tau \mid l!\overline{v} \mid l?\overline{v}
\]

This transition system is used to determine how the state of the business process evolves as interactions with the partners occur, writing \(a \xrightarrow{\alpha} a'\) for activity (state) \(a\) moving to \(a'\) by interaction \(\alpha\). \(\tau\) is the silent action, modelling an internal computation inside the business process; \(l!\overline{v}\) represents the business process sending a message to the end-point \(l\) specifying message \(\overline{v}\), and similarly \(l?\overline{w}\) represents receiving a message from end-point \(l\). The rules defining transition relation \(\rightarrow\) are shown in Figure 6.

Rule [PAR] defines the semantics of operator \(\parallel_P\) in terms of interleaved concurrency, and allows us to focus, from here on, on the behaviour of a single
process instance of the kind \((w \mapsto u)s\) at a time. Rules \([PCK, FLW, SEQ]\) describe the semantics of choice, flow, and sequential composition: the standard algebraic behaviour of such operators is adopted, along with the idea that each transition may affect the store of the enclosing process instance. Two different concurrent activities in the same process instance can synchronise by a source and target link \(\lambda\), as shown in rule \([LNK]\).

Given the above rules, we can now deal with the execution of the other basic activities one at a time, without the need to consider their mutual influence. The structured activity \(\textbf{while}\) is executed by evaluating the boolean condition \(e\) in the current store: if this is true ([WH]) the action \(s\) is executed first and then the \(\textbf{while}\) activity is executed again, otherwise ([NWH]) no other local activity has to be executed — but e.g. the sequential continuation can carry on by rule \([SEQ]\). The behaviour of the \(\textbf{switch}\) structure is similar, the first case is evaluated, and if its condition is satisfied ([SW]) the corresponding activity \(s\) is executed, otherwise the subsequent case is considered ([NSW]). Assigning an expression \(e\) to a variable \(v\) simply means to update the store by the new assignment of \(v\) to the value \(u\) which \(e\) is evaluated to ([ASG]).

Action \(\textbf{send}\) causes a message to be sent outside, whose content is the evaluation \(\{w/u\}v\) of variable \(v\) in the current store. As a result, the update \((w_p \mapsto \{w/u\}v)\) is applied, which both checks for the conformance of the message with respect to the properties of existing correlation sets and adds new properties in the store, modelling the initialisation of new correlation sets ([SND]). Viceversa, by action \(\textbf{recv}\) a new message is received, so that its content \(u'\) is assigned to variable \(v\) by update \((v \mapsto u')\). Moreover, the new update \((w_p \mapsto u')\) is also applied, which controls whether the received values conforms to the existing properties where necessary, and adds the properties of new correlation sets. In particular, notice that an incoming message is necessarily associated to the proper process instance, according to the fact that it is allowed only within the process instance providing compatible correlation sets.

4.4 The Behaviour of the Shipping Service

To better understand the main underpinnings of our operational semantics, we show the evolution of the shipping service which results from the interaction with some customers. For brevity, we denote by \(s_T\) the specification shown in
Figure 4, by $s_{sw}$ and $s_{wh}$ the switch and while activities inside it, that is:

$$s_T ::= \text{recv}(l_r, \overline{v}, (p, v^i_r, v^3_r)); s_{sw}$$

$$s_{sw} ::= \text{switch}(v^i_2)\{\text{assign}(v^2_n, v^i_3); \text{send}(l_n, \overline{v^n}, (p, v^2_n)); \text{assign}(v^i, 0); s_{wh}\}$$

$$s_{wh} ::= \text{while}(v^i < v^i_2)\{\text{assign}(v^2_n, \text{opaque}); \text{send}(l_n, \overline{v^n}, (p, v^2_n)); \text{assign}(v^i, v^i + v^2_n)\}$$

As a case we consider the following evolution:

- The initial state of the business process is $a_0 = !(\bullet \rightarrow \bullet) s_T$, that is, any copy of the specification $s_T$ is ready to be spawned whose initial store is void.

- At a given time, a customer sends a shipping request of the kind $(101, \text{true}, 5)$, asking for the atomic shipment of 5 items, specifying the ID 101:

$$a_0 \xrightarrow{l_r ? (101, \text{true}, 5)} a_0 \parallel p a_1,$$

$$a_1 = (\overline{v} \mapsto (101, \text{true}, 5))(p \mapsto 101) s_{sw}$$

This is obtained first by spawning a new process instance by congruence rule [C-SPN], which receives the request by operational rule [REC] (executed in the context of rule [SEQ]). The effect of this execution is that the store will contain the variable $v^r$ for the received message, while the correlation set property $p$ defining the identity of the process instance is bound to 101.

- Concurrently, a new customer sends the request $(180, \text{false}, 3)$, asking for the possibly non-atomic shipment of 3 items, using ID 180; a new process instance is therefore created similarly to the previous case:

$$a_0 \parallel p a_1 \xrightarrow{l_r ? (180, \text{false}, 3)} a_0 \parallel p a_1 \parallel p a_2,$$

$$a_2 = (\overline{v} \mapsto (180, \text{false}, 3))(p \mapsto 180) s_{sw}$$

Notice that two process instances are now currently active, each with own state of the store, characterised by different orderID.

- The executions of $a_1$ and $a_2$ are isolated. Without losing generality, as we are in the context of interleaved concurrency, we can suppose that the execution of the first process instance $a_1$ carries on until completion, that is (i) evaluating the switch condition, (ii) assigning the variable and then
(iii) performing the invocation:

\[
a_1 \xrightarrow{\tau} \\\n(\overline{v} \mapsto \langle 101, true, 5 \rangle)(p \mapsto 101) assign(v^n_2, v^n_2); send(l_n, \overline{v^n}, \langle p, v^n_2 \rangle) \xrightarrow{\tau} \\\n(\overline{v} \mapsto \langle 101, true, 5 \rangle)(p \mapsto 101)(v^n_2 \mapsto 5); send(l_n, \overline{v^n}, \langle p, v^n_2 \rangle) \xrightarrow{l_n!(101,5)} 0
\]

In the last interaction, notice that the first message part 101 is obtained by evaluating variable \( v^n_1 \) which is not initialised: however, from all the possibilities 101 is chosen because is the only matching the current value of correlation set property \( p \) (rule [SND]).

• Similarly, we report the evolution of the second process instance, which deals with a non-atomic shipment of 3 items. We first suppose that the business process ships 1 item:

\[
a_0 \parallel_P a_2 \xrightarrow{\tau} \\\na_0 \parallel_P (\overline{v} \mapsto \langle 180, false, 3 \rangle)(p \mapsto 180) assign(v^i, 0); s_{wh} \xrightarrow{\tau} \\\na_0 \parallel_P (\overline{v} \mapsto \langle 180, false, 3 \rangle)(p \mapsto 180)(v^i \mapsto 0)s_{wh} \xrightarrow{\tau} \\\na_0 \parallel_P (\overline{v} \mapsto \langle 180, false, 3 \rangle)(p \mapsto 180)(v^i \mapsto 0)assign(v^n_2, opaque); send(l_n, \overline{v^n}, \langle p, v^n_2 \rangle); assign(v^i, v^i + v^n_2); s_{wh} \xrightarrow{\tau} \\\na_0 \parallel_P (\overline{v} \mapsto \langle 180, false, 3 \rangle)(p \mapsto 180)(v^i \mapsto 0)(v^n_2 \mapsto 1)send(l_n, \overline{v^n}, \langle p, v^n_2 \rangle); assign(v^i, v^i + v^n_2); s_{wh} \xrightarrow{l_n!(180,1)} \\\na_0 \parallel_P (\overline{v} \mapsto \langle 180, false, 3 \rangle)(p \mapsto 180)(v^i \mapsto 0)(v^n_2 \mapsto 1)assign(v^i, v^i + v^n_2); s_{wh} \xrightarrow{\tau} \\\na_0 \parallel_P (\overline{v} \mapsto \langle 180, false, 3 \rangle)(p \mapsto 180)(v^i \mapsto 1)(v^n_2 \mapsto 1)s_{wh}
\]

Notice that after this evolution the interaction state is moved back to the initial state \( s_{wh} \), even though the store is changed reflecting e.g. the shipment of 1 item. If at the next step the business process ships 2 items the
process instance terminates, through the evolution:

\[ a_0 \parallel P \left( v^r \mapsto \langle 180, \text{false}, 3 \rangle, p \mapsto \rightarrow 180, v^i \mapsto \rightarrow 1, v^n_2 \mapsto \rightarrow 1 \right) s_{wh} \xrightarrow{\tau} \]

\[ a_0 \parallel P \left( v^r \mapsto \langle 180, \text{false}, 3 \rangle, p \mapsto \rightarrow 180, v^i \mapsto \rightarrow 1, v^n_2 \mapsto \rightarrow 1 \right) \]

\[ \text{assign}(v^n_2, \text{opaque}); \text{send}(l_n, v^n_2, \langle p, v^n_2 \rangle); \text{assign}(v^i, v^i + v^n_2); s_{wh} \xrightarrow{\tau} \]

\[ a_0 \parallel P \left( v^r \mapsto \langle 180, \text{false}, 3 \rangle, p \mapsto \rightarrow 180, v^i \mapsto \rightarrow 1, v^n_2 \mapsto \rightarrow 2 \right) \]

\[ \text{send}(l_c, v^n_2, \langle p, v^n_2 \rangle); \text{assign}(v^i, v^i + v^n_2); s_{wh} \xrightarrow{l_{n!}(180,2)} \]

\[ a_0 \parallel P \left( v^r \mapsto \langle 180, \text{false}, 3 \rangle, p \mapsto \rightarrow 180, v^i \mapsto \rightarrow 1, v^n_2 \mapsto \rightarrow 2 \right) \]

\[ \text{assign}(v^i, v^i + v^n_2); s_{wh} \xrightarrow{\tau} \]

\[ a_0 \parallel P \left( v^r \mapsto \langle 180, \text{false}, 3 \rangle, p \mapsto \rightarrow 180, v^i \mapsto \rightarrow 3, v^n_2 \mapsto \rightarrow 2 \right) s_{wh} \xrightarrow{\tau} a_0 \]

5 Conclusion and Future Works

In this paper we applied typical techniques for formalising syntax and semantics of imperative and concurrent programming languages to define a core language for BPEL4WS orchestration language. To the best of our knowledge, in spite BPEL exploits very well known process algebraic constructs, this is the first attempt to provide a comprehensive model taking into account all these heterogeneous features, and precisely modelling a significant portion of BPEL.

Nevertheless, we believe our work could pave the way for further investigations. First of all, an observational semantics could be introduced to enable formal verification of properties. Then, following the methodological approach for orchestration languages proposed in [16], a type system could be introduced for our core language, allowing to statically check for the correct exploitation of variables and properties, and for analysing the composition of a business process with the collaborating Web Services. Also, it would be interesting to further enlarge the modelled portion of BPEL until a substantial completion. Other than issues such as timing and faults, a challenging and interesting feature is compensation handling, which has recently being investigated [7,5], and it is likely to be formalised by a smooth extension to the model presented here. Finally, as an operational semantic approach is proved feasible here, we are also interested in deepening the relationship between BPEL specification and implementations, for instance evaluating the exploitation of some coordination infrastructure such as TuCSoN [22] as a platform for orchestration engines.
References

[1] Arbab, F., I. Herman and P. Spilling, An overview of MANIFOLD and its implementation, Concurrency: Practice and Experience 5 (1993), pp. 23–70.

[2] Arkin, A., Business Process Modelling Language (2002), http://www.bpmi.org/bpmi-spec.esp.

[3] Assaf Arkin et al., Web Service Choreography Interface (wsci) 1.0 (2002), http://www.w3.org/TR/wsci/.

[4] Baeten, J. C. M., J. A. Bergstra and J. W. Klop, Decidability of bisimulation equivalence for processes generating context-free languages, in: PARLE, Parallel Architectures and Languages Europe, Volume I: Parallel Languages, Lecture Notes in Computer Science 259 (1987), pp. 94–111.

[5] Bocchi, L., C. Laneve and G. Zavattaro, A calculus for long-running transactions, in: Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003), Lecture Notes in Computer Science 2884 (2003), pp. 124–138.

[6] Busi, N., R. Gorrieri and G. Zavattaro, A process algebraic view of Linda coordination primitives, Theoretical Computer Science 192 (1998), pp. 167–199.

[7] Butler, M. and C. Ferreira, A process compensation language, in: Integrated Formal Methods IFM2000, Lecture Notes in Computer Science 1945 (2000), pp. 61–76.

[8] Christensen, E., F. Curbera, G. Meredith and S. Weerawarana, Web Services Description Language (wsdl) 1.1 (2001), http://www.w3.org/TR/wsd1.

[9] Ciancarini, P., Coordination models and languages as software integrators, ACM Computing Surveys 28 (1996), pp. 300–302.

[10] De Nicola, R., G. Ferrari and R. Pugliese, KLAIM: a kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering 24 (1998), pp. 315–330.

[11] Freeman, E., S. Hupfer and K. Arnold, “JavaSpaces: Principles, Patterns, and Practice,” Addison-Wesley, 1999.

[12] Gelernter, D., Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1985), pp. 80–112.

[13] Gelernter, D. and N. Carriero, Coordination languages and their significance, Communication of the ACM 35 (1992), pp. 96–107.

[14] Igarashi, A., B. C. Pierce and P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (2001), pp. 396–450.

[15] Mecella, M., F. Parisi-Presicce and B. Pernici, Modeling e-service orchestration through Petri Nets, in: Technologies for E-Services, Third International Workshop, TES 2002, Hong Kong, China, August 23-24, 2002, Proceedings, Lecture Notes in Computer Science 2444 (2002), pp. 38–47.

[16] Meredith, L. G. and S. Bjorg, Contracts and types, Communications of the ACM 46 (2003), pp. 41–47.

[17] Milner, R., “Communication and Concurrency,” Prentice Hall, 1989.

[18] Milner, R., “Communicating and Mobile Systems: The π-calculus,” Cambridge University Press, 1999.

[19] Omicini, A. and F. Zambonelli, Coordination for Internet application development, Journal of Autonomous Agents and Multi-Agent Systems 2 (1999), pp. 251–269.

[20] Peltz, C., Web services orchestration and choreography, IEEE Computer 36 (2003), pp. 46–52.
[21] Tony Andrews et al., *Specification: Business Process Execution Language for Web Services version 1.1* (2003),
http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[22] Tucson home page, http://lia.deis.unibo.it/research/TuCSoN (2002).

[23] Viroli, M. and A. Omicini, *Coordination as a service: Ontological and formal foundation*, in: A. Brogi and J.-M. Jacquet, editors, *1st International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA '02)*, Electronic Notes in Theoretical Computer Science 68(3), Elsevier Science B. V., 2003.

[24] Wright, A. K. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation 115 (1994), pp. 38–94.