A core calculus for correlation in orchestration languages

Mirko Viroli

DEIS, Alma Mater Studiorum – Università di Bologna, Via Venezia 52, 47023 Cesena (FC), Italy

Abstract

We introduce a formal framework for studying the mechanism of correlation in orchestration languages for Web Services. A core calculus based on typical process algebraic constructs is developed, enhanced with two mechanisms: (i) a management of scopes keeping track of variables, properties, and their assignment to values, and (ii) a construct to spawn service instances handling (cor-)related operations and guaranteeing consistent routing of messages. By abstracting away from low-level details of orchestration languages, this model can be used as a foundation for the correlation mechanism, paving the way towards the analysis of properties and the design of extensions and improvements. As an example application, we show how the calculus introduced can be extended with few imperative and control-flow constructs reaching the expressiveness of a significant fragment of BPEL orchestration language.

© 2006 Elsevier Inc. All rights reserved.

Keywords: Web Services; Orchestration languages; Process algebras

1. Introduction

A main ingredient of today Web Services technology is service orchestration. Orchestration languages specify relationships and constraints over the interactions between existing and loosely coupled Web Services: the resulting coordinating behaviour is called a business process. Orchestration engines are then developed that take a specification in one such language and automatically implement the 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 [1,2], simplifying their analysis, design, and deployment. Some orchestration languages emerged as proposals for a standardisation in the Web Services technology, such as BPEL4WS (Business Process Execution Language for Web Services, BPEL for short), 1 and BPML (Business Process Management Language). 2 In spite of their different proposers, they all basically share the same design principles and language features, which can then be safely ascribed to the class of “orchestration languages” in general.

Orchestration languages are largely inspired by process algebras such as CCS [5] and \( \pi \)-calculus [6]: they provide primitive operations to send and receive messages, composed by operators for choice, parallel composition, and

---

1 By a consortium including Microsoft, IBM, Siebel System, BEA and SAP [3].
2 By Sun Microsystems, Intalio, Sterling Commerce, and CSC [4].
sequential composition. Still, additional and (sometimes) novel features are included to fit the specific aims of business processes. Since a business process concurrently handles multiple conversations with different clients, i.e., different working sessions, a mechanism called correlation is introduced to declaratively (co)relate operations of the same conversation [7]. Basically, this mechanism is used to identify service instances: parts of the business process each in charge of a different working session. Service instances are stateful processes carrying a (possibly multiply-nested) memory of variables, featuring imperative operations and constructs to update it, and possibly structured in a net of sub-activities [8]. Service instances are meant to execute critical operations and possibly last for a long time, hence a support for lung-running transactions is included as well, featuring the compensation mechanism to recover from faulty states [9].

Among all these features, we find correlation at the core of the approach of orchestration languages, for it encapsulates the logic by which a business processes splits its main task to different service instances. Accordingly, the goal of this paper is to study the basic properties and features of correlation, taking as a reference the mechanism of correlation sets of orchestration languages such as BPEL, but generalising and abstracting over it. This is achieved by introducing a core calculus for correlation in business processes, based on the language we presented in [7]. This calculus is meant to provide a foundation to the correlation mechanism: it is a formal tool to be exploited in different contexts and scenarios, such as for studying formal analysis results that help verification of properties, devising extensions and adaptations exploitable in future proposals, studying integration with other constructs, specifying existing orchestration languages, driving the implementation of orchestration engines, and so on.

Our stance here is similar to other researches introducing core calculi for mainstream programming languages. Featherweight Java (FJ) is a remarkable example [10]: it is a calculus used to model the basic object-oriented features of Java while neglecting lower level details such as side-effects, static properties, and primitive types. FJ has been used in a plethora of other researches discussing current and new features of Java and of object-oriented languages in general: the basic well-formedness properties of FJ proved in [10] – including type soundness [11] – generally guarantee these new studies to be grounded on a solid and rigorous framework. The many examples include the study of Java inner classes [12] and separate compilation [13], and the study of the new subtyping scheme for generics in Java 5.0 (also known as wildcards) [14].

This study is meant to contribute towards a similar tool for orchestration languages, and is developed incrementally. Section 2 starts by surveying the main features of orchestration languages, focussing on BPEL and describing details of its correlation mechanism. Section 3 introduces a first core calculus for service instances, which is basically a process algebra where processes run inside scopes: each scope defines a store of variables and properties – late-bound constants handling correlation [3] –, holding the data to be sent and received. Basic properties about this calculus are studied, including a well-formedness judgment guaranteeing avoidance of structural deadlocks. Section 4 extends this calculus to deal with the key mechanism of correlation. In particular, an instance-spawn construct is added which is in charge of (i) creating new instances as interactions occur – each instance characterised by an assignment of properties in a correlation set –, (ii) guaranteeing uniqueness of such assignments by preventing inconsistent interactions, (iii) routing messages to the instance in charge of handling them. We show how the basic properties of the model remain unchanged, and provide examples of how the main correlation schemata allowed by orchestration languages can be modelled. As an example application, Section 5 extends the calculus with few imperative and control-flow constructs, allowing to express significant examples of orchestration specifications. Section 6 discusses related work and Section 7 concludes providing final remarks.

2. Orchestration languages and correlation

This section describes the main aspects of orchestration languages, focussing on the correlation mechanism.

As a representative orchestration language we take BPEL, which at the time of writing appears to be the main candidate for a standardisation.3

2.1. BPEL

BPEL is an XML-based specification language for describing business processes orchestrating the interaction of different, existing or possibly dynamically emerging Web Services. As such, it builds on top of the WSDL language for

---

3 In particular, version 2.0 is about to be released as an official specification.
describing the interface of Web Services [15]. A Web Service interface is specified in terms of port types, operations, 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, port types are lists of operations and operations are of two kinds: one-way, when they are asynchronously invoked without waiting for any reply, or request-response, in the case where a reply is actually expected. Finally, messages are basically XML data records.

On top of a WSDL document describing the above “boundary” aspects, a BPEL specification provides information on the internal business process of the Web Service. This is composed of four declaration parts: (i) the partner link types, (ii) the variables, (iii) the correlation sets, and (iv) the activity.

Partner links identify the relationship between the business process and the other Web Services it interacts to: a partner link is a unique name used to identify either the port types for a process/web-service interaction or the port types for a web-service/process interaction. It is worth noting that a business process never directly refers to a client as a specifically installed Web Service, but to the port types of a partner link, bound to a Web Service at deployment-time or even dynamically at run-time. This abstraction is particularly relevant, since it enables those scenarios where pools of Web Services are dynamically bound and unbound to a business process in an orthogonal way, depending, e.g., on load-balancing issues.

A key aspect of a business process is that its global task is divided into different sessions (called service instances), each responsible for carrying on a separate service or work for each client. Therefore, service instances must be stateful processes, holding the necessary information about the conversation. To support this scenario, the second part of a BPEL specification defines variables that can carry XML data values and messages, and which are used to define the state of each service instance. Most notably, variables can also contain partner links, that is, abstract references to other services: similarly to the π-calculus where channels are used to exchange names of channels [6], this mechanism is useful to express dynamic interconnecting structures.

Correlation sets are then introduced to identify those interactions that are pertinent to a given service instance: this is necessary in order to correctly route incoming messages to the proper instance – see more details on that in Section 2.3.

Finally, the last part specifies the behaviour of the business process, namely an activity. Activities are generally built by composing basic ones through structured ones. Basic activities feature the acts of sending and receiving requests and replies (invoke, receive); others include variable assignment (assign), synchronisation of internal concurrent activities through private links (source and target), waiting for a timeout (wait), and raising faults (throw). Structured activities realise sequential composition (sequence), guarded choice (pick), parallel composition (flow), iteration cycles (while), and multiple cases (switch).

2.2. An example specification

As a reference case study, in this paper we consider the shipping service described in the official specification of BPEL [3] (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 messages management.

This example describes a Web Service that realises a service handling the shipment of orders requested by customers. 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 starts specifying partner link types. Only one partner link is specified here that represents 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 operation named shippingNotice. Then, shipping request messages are defined as being 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. Similarly, shipping notice messages are made of the orderID integer and the count 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. Finally, the activity realising the

---

4 In particular, WSDL standard actually provides more kinds of operation, namely, solicit-response and notification, but these as not supported, e.g., by BPEL and are not of interest in the context of this paper.
The algorithm realised is as follows. As the request is received, a new service instance is spawned that will handle the subsequent operations (initiate="yes"). If the complete part flag of the message is true a reply is immediately invoked with the same count. Otherwise, a while iteration is executed. Each time, the count part of the shipNotice message is assigned to 1. 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 service instance terminates.

2.3. On the correlation mechanism

As in this shipping service, a business Web Service realised through an orchestration specification is typically exploited as a mediator between clients and some orchestrated services, and is charged with the burden of concurrently handling the interactions of a potentially very high number of clients, effectively dealing with long-running sessions and keeping track of the state of each over time. The way this is achieved is by splitting the whole business process into different isolated working sessions called process instances, each having the same behaviour of the business process specification but running on a different scope – with a different store of variables and pertaining different subsequent interactions.

The correlation mechanism is introduced precisely to specify in a declarative way how different interactions can be identified as belonging to the same process instance, and in particular, how incoming messages are to be routed to the proper process instance. To this end, a particular kind of variable called property is introduced in BPEL. It is basically a late-bound constant – once initialised its content will never change – and is defined so as to alias the single field of one or more message types. On the one hand, when a message is sent, a field linked to a property is automatically

---

5 In the shipping example in BPEL specification, this is actually 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. The resulting process is therefore abstract, i.e., non-directly executable. Since this aspect is not of interest in this paper, we reverted to a deterministic process without loss of generality.
bound to hold the value associated to that property. On the other hand, a message is received by a process instance only if the field linked to a property contains the value associated to the property.

This mechanism guarantees all the messages sent and received by a process instance to be compliant with the initialisation of properties. Then, a process instance is associated with a *correlation set*, namely a set of properties: the assignment of the correlation set uniquely characterises the process instance inside the business process, and is sufficient to correctly route incoming messages.

In the shipping service example, for instance, only one correlation set *shipOrder* is defined which contains the property *orderID* aliasing the first part of both *shipRequest* and *shipNotice* messages: This is obtained by the XML code:

```xml
<bpws:propertyAlias propertyName="tns:shipOrderID"
   messageType="sns:shippingRequestMsg"
   part="shipOrder"
   query="/ShipOrderRequestHeader/shipOrderID"/>

<bpws:propertyAlias propertyName="tns:shipOrderID"
   messageType="sns:shippingNoticeMsg"
   part="shipNotice"
   query="/ShipNoticeHeader/shipOrderID"/>

<correlationSets>
   <correlationSet name="shipOrder"
                  properties="props:shipOrderID"/>
</correlationSets>
```

That property will contain the order identifier of the current shipping request, and will uniquely identify the service instance in charge of handling it: shipping requests and notices of the process instances are structurally bound to carry that order identifier.

In the typical pattern, the first basic activity executed by a business service is the reception of a message causing the spawning of a new process instance and the initialisation of its correlation set. The actual XML code of one such activity in the shipping service is

```xml
<receive partnerLink="customer"
         portType="sns:shippingServicePT"
         operation="shippingRequest"
         variable="shipRequest">
   <correlations>
     <correlation set="shipOrder" initiate="yes"/>
   </correlations>
</receive>
```

A number of variations of this basic schema is actually possible, including the scenario where the business process initiates a working session by sending a message (internally generating an initialisation for correlation sets), where it initially receives different kinds of messages (multiple start activities), where a process instance becomes itself a source of new process instances (initialising a new correlation set).

All these cases complicate the understanding of this mechanism, thus requiring a formal addressing as developed in this paper.

3. A basic algebra for service instances

We start our study by introducing a language for service instances, abstracting away from the correlation mechanism used to spawn them. This corresponds to the idea of monolithic business processes, handling only one working session at a time.
In this language, and more generally in the formal framework introduced in this paper, a number of interesting aspects of orchestration languages are intentionally neglected, such as timeouts, fault handlers, compensation handlers, reconfigurability of partner links, XML data representation, and so on. Moreover, following the abstraction used by BPEL when describing partner links, the details of interactions with other Web Services are not described: an interaction is characterised by (i) its direction (sent or received by the business process), (ii) the involved peer (the partner link, modelled as a channel), and (iii) the content (the message carried, as a sequence of values). Including further details would not be relevant to the end of studying the aspects we are interested in here; rather, they can be introduced later on top of our framework in a mostly orthogonal way, so as to provide a complete specification of an orchestration language.

3.1. Stores of variables and properties

We let metavariable $v$ range over the finite set of variable identifiers, $p$ over the finite set of property identifiers, and $u$ over the denumerable set of values (integers, booleans, strings, floating-point numerals, and the like). The following syntax is introduced to describe stores and their content:

- $w ::= v \mid p$ Store locations
- $e ::= w \mid u$ Basic expressions
- $k ::= u \mid \text{null}$ Store locations content
- $\sigma, \rho ::= w \leftrightarrow k \mid \sigma, \sigma$ Stores
- $\pi ::= p \leftrightarrow \text{null} \mid \pi, \pi$ Correlation sets

A store is a composition – through the comma operator – of associations $w \leftrightarrow k$ between a store location $w$, either a variable $v$ or a property $p$, and a store location content $k$, which can be a (regular) value $u$ or a special value null meaning the location is not yet initialised. Metavariable $\pi$ represents a particular kind of store, called a correlation set, namely an uninitialised composition of property locations.

Note that the comma composition operator for stores is neither assumed to be commutative nor associative – as seen later, this is useful to compose different stores and then be able to keep track of each of them separately. Nevertheless, it is useful to rely on the notation $\sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_n$, which is a shorthand for $(\sigma_1, (\sigma_2, \ldots, (\sigma_{n-1}, \sigma_n)))$. The intuition behind the composition $\sigma, \rho$ – which will be later made clear by our operational semantics – is that store $\rho$ is in a scope nested inside the scope of store $\sigma$, hence $\rho$ might override some definitions in $\sigma$.

Two operators are defined for stores: store lookup (expression evaluation) and store update. Store lookup is defined by the partial function $\sigma(e)$, used to evaluate expression $e$ in the scope $\sigma$, yielding a value $u$. We write $\sigma(e) = \perp$ when $\sigma$ is undefined in $e$ (and $\sigma(e) \neq \perp$ in the opposite case) – however, this is just a notation, for $\perp$ is not considered a valid result. The semantics of such an evaluation is defined as

- $\sigma(u) \triangleq u$
- $(w \mapsto u)(w) \triangleq u$
- $(\sigma, \rho)(w) \triangleq \rho(w)$ if $\rho(w) \neq \perp$
- $(\sigma, \rho)(w) \triangleq \sigma(w)$ if $\rho(w) = \perp$

Namely, while a value $u$ evaluates to itself, a store location identifier $w$ is evaluated to the corresponding content as occurring in its right-most occurrence in the store. For instance, let $v_1, v_2, v_3$ be distinct variables, we have

- $(v \mapsto 1, v \mapsto 2)(v) = 2$
- $(v_1 \mapsto 1, v_2 \mapsto 2)(v_1) = 1$
- $(v_1 \mapsto 1, v_2 \mapsto 2, v_1 \mapsto 4)(v_2) = 2$
- $((v_1 \mapsto 1, v_2 \mapsto 2, v_1 \mapsto 4), (v_2 \mapsto 3))(v_2) = 3$
- $((v_1 \mapsto 1, v_2 \mapsto 2, v_1 \mapsto 4), (v_2 \mapsto 3))(v_1) = 4$

Note that the result of a store lookup $\sigma(w)$ is not defined in two cases: when location $w$ is not defined in store $\sigma$, or when the right-most occurrence of $w$ is not initialised – i.e., $w$ is associated to the content null. For instance, we have

- $(v_1 \mapsto 1, v_2 \mapsto 2)(v_3) = \perp$
- $(v_1 \mapsto 1, v_2 \mapsto \text{null})(v_2) = \perp$
Store update is a partial binary operator \( \oplus \) over stores: \( \sigma \oplus \rho \) is the store \( \sigma \) updated with all the new locations of store \( \rho \). We write again \( \sigma \oplus \rho = \bot \) when the operator cannot be applied to the couple \( \sigma, \rho \). The operator is defined as

\[
\begin{align*}
\sigma \oplus (\rho_1, \rho_2) & \triangleq (\sigma \oplus \rho_1) \oplus \rho_2 \\
(\sigma, \rho) \oplus w \mapsto k & \triangleq (\sigma \oplus (\rho \oplus w \mapsto k)) \text{ if } \rho \oplus w \mapsto k \neq \bot \\
(\sigma, \rho) \oplus w \mapsto k & \triangleq (\sigma \oplus w \mapsto k), \rho \text{ if } \rho \oplus w \mapsto k = \bot \\
v \mapsto k \oplus v \mapsto u & \triangleq v \mapsto u \\
p \mapsto \text{null} \oplus p \mapsto u & \triangleq p \mapsto u \\
p \mapsto u \oplus p \mapsto u & \triangleq p \mapsto u
\end{align*}
\]

The first rule makes the right-side store be considered from left to right (\( \rho_1 \) is composed to \( \sigma \) prior to \( \rho_2 \)), the second and third rules make the left-side store be affected from right to left (\( w \mapsto k \) is tentatively combined to \( \rho \) prior to \( \sigma \)). The other three rules handle the core differences between properties and variables: a variable can always be updated (from \( v \mapsto k \) to \( v \mapsto u \)), a property can be initialised (from \( p \mapsto \text{null} \) to \( p \mapsto u \)) and after that only dummy updates (from \( p \mapsto u \) to \( p \mapsto u \)) are allowed – remember that a property is handled as a late-bound constant, hence once initialised it can no longer be updated with a new value. For instance, we have

\[(v_1 \mapsto 1, v_2 \mapsto 2) \oplus (v_1 \mapsto 3) = (v_1 \mapsto 3, v_2 \mapsto 2)\]
\[(v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3) \oplus (v_2 \mapsto 0, v_1 \mapsto 0) = (v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 3)\]
\[(v_1 \mapsto 1, v_2 \mapsto 2, v_1 \mapsto 3) \oplus (v_1 \mapsto 0) = (v_1 \mapsto 1, v_2 \mapsto 2, v_1 \mapsto 0)\]
\[\left((v_1 \mapsto 1, v_2 \mapsto 2), (v_1 \mapsto 3)\right) \oplus (v_1 \mapsto 0, v_2 \mapsto 0) = (\left((v_1 \mapsto 1, v_2 \mapsto 0), (v_1 \mapsto 0)\right)\]
\[(v_1 \mapsto 1, v_2 \mapsto 2) \oplus (v_1 \mapsto 3) \oplus (v_1 \mapsto 5) = (v_1 \mapsto 5, v_2 \mapsto 2)\]

The result of the latter two examples is crucial here: (i) the attempt to update store \( p \mapsto 1, v \mapsto 2 \) by changing the value of property \( p \) is refused (store update \( \oplus \) is undefined), and similarly (ii) for the attempt to update \( v_1 \mapsto 1 \) with a new value for the undefined variable \( v_2 \).

### 3.2. Interactive update of stores

We let metavariable \( \bar{c} \) range over channels (modelling partner links), and for any meta-variable \( a \) denote by \( \bar{a} \) the meta-variable ranging over comma-separated sequences \( a_1, a_2, \ldots, a_n \) of \( a \) elements – where there is no risk of ambiguity, such sequences are also seen as sets of \( a \) elements. Correspondingly, the notation for stores and store lookup is augmented, writing \( \bar{w} \mapsto \bar{u} \) for the store \( w_1 \mapsto u_1, \ldots, w_n \mapsto u_n \) (\( \bar{w} \) and \( \bar{u} \) are supposed to have same arity by construction), \( \sigma(\bar{e}) \) for \( \sigma(e_1), \ldots, \sigma(e_n) \), and \( \bar{w} \mapsto \text{null} \) for \( w_1 \mapsto \text{null}, \ldots, w_n \mapsto \text{null} \).

We start by introducing a fragment of our final core language, neglecting the spawning of different service instances while focusing on how typical process algebraic constructs work with stores of variables and properties.

The syntax of this fragment is defined as follows:

\[
S, R ::= \begin{align*}
0 & \text{ Void process} \\
| (\sigma)S & \text{ Scoped process} \\
| x.S & \text{ Action prefix} \\
| S + R & \text{ Choice} \\
| S || R & \text{ Parallel interleaved composition}
\end{align*}
\]

\[
x ::= \begin{align*}
| e!\bar{w} & \text{ Message sending} \\
| e?\bar{w} & \text{ Message receiving} \\
| w \triangleleft e & \text{ Location assignment}
\end{align*}
\]
We rely here on the term “process” basically as a synonym for “service”. A process \((\sigma)S\) consists of process \(S\) running inside the scope defined by store \(\sigma\), so that \(S\) is allowed to read/update its content. A process \(x.S\) executes prefix \(x\) and then behaves like \(S\). Executing \(c!w\) means to send a message towards the channel \(c\): this message is formed by the sequence of values currently contained in the store locations \(w\). Dually, \(c?w\) means receiving from the channel \(c\) a message, so that its values will be used to update the content of store locations \(w\). Finally, \(w \Leftarrow e\) updates the content of \(w\) in the current store by the result of evaluating expression \(e\). Terminated process 0, operators for choice \(+\) and parallel composition \(\parallel\) are as usual, and for them the following typical congruence rules are assumed:

\[
\begin{align*}
0 + S & \equiv S + 0 + R & \equiv R + S = (S + R) + R' & \equiv S + (R + R') \\
0 \parallel S & \equiv S \parallel R \equiv R \parallel S & (S \parallel R) \parallel R' & \equiv (S \parallel R') \parallel R
\end{align*}
\]

\[
\text{Moreover, the following congruence rule is added:}
\]

\[
(\sigma)0 \equiv 0
\]

stating that scopes of terminated processes are of no relevance. To simplify notation, the final “.0” notation is often omitted at the end of processes.

We give semantics to this algebra by a labelled transition system \(\langle S, \rightarrow, \text{Act} \rangle\), where \(S\) is the set of processes (ranged over by \(S\) and \(R\)), \(\text{Act}\) is the set of labels defined by syntax

\[
\alpha ::= \tau \quad \text{Internal (silent) action}
\]

\[
\mid c!u \quad \text{Message sending}
\]

\[
\mid c?u \quad \text{Message receiving}
\]

and the transition relation \(\rightarrow\) is of the kind \(\rightarrow \subseteq S \times \text{Act} \times S\). We write \(S \overset{\alpha}{\rightarrow} R\) as usual for \(\langle S, \alpha, R \rangle \in \rightarrow\), and \(S \not\rightarrow\) when for no \(\alpha\) and no \(R\) we have \(\langle S, \alpha, R \rangle \in \rightarrow\). In the style of structural operational semantics (SOS [16]), relation \(\rightarrow\) is defined as the smallest relation satisfying the following operational rules:

\[
\frac{S \equiv S' \quad S' \overset{\alpha}{\rightarrow} R' \quad R' \equiv R}{S \overset{\alpha}{\rightarrow} R} \quad \text{[CGR]}
\]

\[
\frac{\text{(\(\sigma\))S} \overset{\alpha}{\rightarrow} (\text{(\(\sigma'\))S}')}{(\text{(\(\sigma\))(S + R)} \overset{\alpha}{\rightarrow} (\text{(\(\sigma'\))(S')}} \quad \text{[SUM]}
\]

\[
\frac{\text{(\(\sigma\))S} \overset{\alpha}{\rightarrow} (\text{(\(\sigma'\))S}')}{(\text{(\(\sigma\))(S || R) \overset{\alpha}{\rightarrow} (\text{(\(\sigma'\))(S || R)}} \quad \text{[PAR]}
\]

\[
\frac{\text{(\(\sigma\))(\(\rho\))S} \overset{\alpha}{\rightarrow} (\text{(\(\sigma'\))(\(\rho'\))S')}}{\text{(\(\sigma\))w \leftarrow e,S} \overset{\tau}{\rightarrow} (\text{(\(\sigma \oplus w \mapsto \sigma(e))S)}} \quad \text{[NEST]}
\]

\[
\frac{\text{(\(\sigma\))c!\(\overline{w}\).S} \overset{c!\(\overline{w}\)}{\rightarrow} (\text{(\(\sigma\))S)}}{\text{(\(\sigma\))c?\(\overline{w}\).S} \overset{c?\(\overline{w}\)}{\rightarrow} (\text{(\(\sigma \oplus \overline{w} \mapsto \overline{w})S)}} \quad \text{[NEST]}
\]

Rule [CGR] formally expresses the fact that, when operational semantics is concerned, two equivalent processes are to be considered syntactically the same. Rule [SUM] and [PAR] define parallel and choice composition of processes as usual, but stating that each update of variables and properties is applied to the outer store, thus possibly leading to some side-effect. Rule [NEST] is used to handle nested scopes: process \(S\) sees \(\rho\) as an inner scope with respect to \(\sigma\) by means of the comma operator, hence any definition in \(\rho\) shadows the corresponding one in \(\sigma\). Rule [ASG] defines the semantics of assignment \(w \leftarrow e\): expression \(e\) is evaluated in the current store \(\sigma\), and the new location \(w \mapsto \sigma(e)\) is used to update \(\sigma\). Rule [SND] and [RCV] handle message sending and receiving: in the first case locations \(w\) are evaluated in \(\sigma\) and the result is sent in the message, in the second case the received values \(\overline{w}\) are used to update the store.
Note that because of the definition of the operator $\oplus$ and of rule [RCV], a message $c?w$ can be received in store $\sigma$ only if it induces correct reassignments of store locations, namely if $\sigma \oplus w \mapsto w$ yields a valid store. Therefore, this mechanism makes a process receiving only those messages matching the existing values of properties – as shown in the next section, this will guarantees to preserve the correlation property. Similarly, for rule [ASG], an assignment leading to a wrong store update enables no transitions.

The calculus obtained by these definitions is basically an hybrid between a standard CCS-like process algebra – with action prefix, choice and parallel composition – and a stateful imperative core language such as lambda-calculus with references (Reference ML) [11]. In particular, processes are seen as (possibly) multi-threaded activities sharing a common store of information, with operations to access and update it. A process can send a message containing values taken from the current state of the store, can receive values to be written in the store, or can directly update the store by an assignment instruction $w \leftarrow e$. A store is made by a composition of variables and properties: while the former can be re-assigned many times, the latter are initialised only once, and attempts to change their assignment are prohibited. Finally, because business processes typically feature nested scopes, stores can occur at different levels and be arbitrarily nested: due to the mechanism of stores composition, an inner scope shadows the outer scope – redefined variables hide the outer ones, as with variable definition in imperative languages such as Java.\(^6\)

### 3.3. Examples

To emphasise the main features of this language, some examples of specification and the corresponding behaviour are here provided.

- **Side effect on stores**

  

  \[
  S_1 = (v \mapsto 1)(c_a!v \parallel v \leftarrow 2, cb!v)
  \]

  This process runs on the store initially assigning the value 1 to the variable $v$, and is made of two subprocesses: one sending the content of $v$ along channel $c_a$, the other assigning 2 to $v$ and then sending $v$ along $c_b$. Three interaction histories can result from this specification:

  \[
  S_1 \xrightarrow{c_a^{\text{ll}}} (v \mapsto 1)(v \leftarrow 2, cb!v) \xrightarrow{r} (v \mapsto 2) cb!v \xrightarrow{cb^{\text{ll}}} (v \mapsto 2)0 \equiv 0
  \]

  \[
  S_1 \xrightarrow{r} (v \mapsto 2)(c_a!v \parallel cb!v) \xrightarrow{cb^{\text{ll}}} (v \mapsto 2) cb!v \xrightarrow{cb^{\text{ll}}} (v \mapsto 2)0 \equiv 0
  \]

  \[
  S_1 \xrightarrow{r} (v \mapsto 2)(c_a!v \parallel cb!v) \xrightarrow{cb^{\text{ll}}} (v \mapsto 2) cb!v \xrightarrow{cb^{\text{ll}}} (v \mapsto 2)0 \equiv 0
  \]

  In the third case, note that firstly variable $v$ is assigned to 2 creating a side-effect on the left-side subprocess, which then sends 2 along $c_a$.

- **Nested scopes and shadowing**

  \[
  S_2 = (v_1 \mapsto \text{null}, v_2 \mapsto 1)(c_a!v_2 \parallel ((v_2 \mapsto \text{null}) cb?v_2))
  \]

  In this case the store comprises the uninitialised variable $v_1$, and $v_2$ initialised to 1: the right-side subprocess, here, runs inside a sub-scope where variable $v_2$ is redefined. Interestingly, as the message is received from $cb$, variable $v_2$ in the outer scope is left assigned to 1, as the following evolution shows:

  \[
  S_2 \xrightarrow{cb^{\text{ll}}} (v_1 \mapsto \text{null}, v_2 \mapsto 1)(c_a!v_2 \parallel (v_2 \mapsto 8)0) \equiv
  \]

  \[
  (v_1 \mapsto \text{null}, v_2 \mapsto 1) c_a!v_2 \xrightarrow{c_a^{\text{ll}}} (v_1 \mapsto \text{null}, v_2 \mapsto 1)0 \equiv 0
  \]

  In particular the first transition occurs due to the following derivation (rules [PAR] and [NEST]):

  \[
  (v_1 \mapsto \text{null}, v_2 \mapsto 1), v_2 \mapsto \text{null}) cb?v_2 \xrightarrow{cb^{\text{ll}}} ((v_1 \mapsto \text{null}, v_2 \mapsto 1), v_2 \mapsto 8)0
  \]

  \[
  (v_1 \mapsto \text{null}, v_2 \mapsto 1)(v_2 \mapsto \text{null}) cb?v_2 \xrightarrow{cb^{\text{ll}}} (v_1 \mapsto \text{null}, v_2 \mapsto 1)(v_2 \mapsto 8)0
  \]

  \[
  S_2 \xrightarrow{cb^{\text{ll}}} (v_1 \mapsto \text{null}, v_2 \mapsto 1)(c_a!v_2 \parallel ((v_2 \mapsto 8)0))
  \]

---

\(^6\) In BPEL property definition cannot occur inside scopes, even though the definition of new correlation sets can. However, in our formal framework we find it useful to generalise and treat variables and properties in a uniform way.
Properties constraining messages

\[ S_3 = (p \mapsto \text{null}, v \mapsto \text{null})c_e^?p.c_b^!p.(p < 1 \parallel c_e^?((p, v) \parallel c_e^?((p, v))) \]

This example shows how properties are used to constrain the content of messages sent and received by a process. The store of \( S_3 \) has the uninitialised variable \( v \) and property \( p \). Initially, a value for \( p \) is received from \( c_b^! \); from then on the content of the property cannot be changed. The process sends \( p \) towards \( c_b^! \) and then waits for the reception of two double-value messages from \( c_e^? \), each of them having as first component precisely \( p \) (which can be seen as the ID for the interaction session). Consider this evolution:

\[
\begin{align*}
S_3 \xrightarrow{c_e^?1053} & \quad (p \mapsto 1053, v \mapsto \text{null})c_b^!p.(p < 1 \parallel c_e^?((p, v) \parallel c_e^?((p, v))) \\
& \xrightarrow{c_b^!1053} (p \mapsto 1053, v \mapsto \text{null})c_b^!p.(p < 1 \parallel c_e^?((p, v) \parallel c_e^?((p, v)))c_e^?1053,1) \\
& \xrightarrow{c_e^?1053,3} (p \mapsto 1053, v \mapsto 1)(p < 1 \parallel c_e^?((p, v)))c_e^?1053,3 \\
& \xrightarrow{c_e^?1053} (p \mapsto 1053, v \mapsto 3)(p < 1)
\end{align*}
\]

Notice that the request for assigning \( p \) to \( 1 \) can never be executed for the store update \( (p \mapsto 1053, v \mapsto 3) \oplus (p \mapsto 1) \) yields no result. The system remains in a deadlock state, which can be interpreted as an exception has occurred.

3.4. Basic properties of the model

3.4.1. Well-formedness

We are interested in providing (type-)soundness-like properties, studying a procedure for checking the static well-formedness of a specification, so as to avoid run-time errors as much as possible – faults when executing the specification. Since this is an untyped language, a full type-system is not provided: only a set of checks can be defined to avoid using stores in an inconsistent way, namely, (i) reading a location which has not been defined and (ii) writing a location which has not been defined.

We introduce the judgment \( W, W' \vdash S \text{ ok} \), where \( W \) and \( W' \) are sets of locations ranged over by \( w \). The intuition behind it is that a process \((\sigma)S\) where store \( \sigma \) defines at least locations \( W \) and initialises at least locations \( W' \) will not try to update the store in a wrong way – i.e., one that would get the process stuck. We write \( \vdash S \text{ ok} \) as a shorthand for \( \emptyset, \emptyset \vdash S \text{ ok} \), and correspondingly say that \( S \) is well-formed, and write \( W, W' \vdash S_1, \ldots, S_n \text{ ok} \) for \( W, W' \vdash S_1 \text{ ok}, \ldots, W, W' \vdash S_n \text{ ok} \).

This judgment is formally defined by the rules:

\[
\begin{align*}
& W, W' \vdash 0 \text{ ok} \\
& W, W' \vdash (\sigma, \sigma')S \text{ ok} \quad \text{if} \quad W, W' \vdash (\sigma)(\sigma')S \text{ ok} \\
& W, W' \vdash (w \mapsto u)S \text{ ok} \quad \text{if} \quad W \cup \{w\}, W' \cup \{w\} \vdash S \text{ ok} \\
& W, W' \vdash (w \mapsto \text{null})S \text{ ok} \quad \text{if} \quad W \cup \{w\}, W' \vdash S \text{ ok} \\
& W \cup \{w\}, W' \vdash w < u. S \text{ ok} \quad \text{if} \quad W \cup \{w\}, W' \cup \{w\} \vdash S \text{ ok} \\
& W \cup \{w, w'\}, W' \cup \{w'\} \vdash w < w'. S \text{ ok} \quad \text{if} \quad W \cup \{w, w'\}, W' \cup \{w, w'\} \vdash S \text{ ok} \\
& W \cup \overline{w}, W' \cup \overline{w} \vdash c!\overline{w}. S \text{ ok} \quad \text{if} \quad W \cup \overline{w}, W' \cup \overline{w} \vdash S \text{ ok} \\
& W \cup \overline{w}, W' \vdash c ? \overline{w}. S \text{ ok} \quad \text{if} \quad W \cup \overline{w}, W' \cup \overline{w} \vdash S \text{ ok} \\
& W, W' \vdash S + R \text{ ok} \quad \text{if} \quad W, W' \vdash S, R \text{ ok} \\
& W, W' \vdash S \parallel R \text{ ok} \quad \text{if} \quad W, W' \vdash S, R \text{ ok}
\end{align*}
\]

Each rule handles a different construct of the language. For instance, fifth rule handles the case \( w < u. S \): for one such process to be well-formed, \( w \) should be already defined (it should not necessarily be initialised), and \( S \) should be well-formed under the further assumption that \( w \) is now also initialised. The other rules behave similarly.

3.4.2. Contexts

To reason about evolution of subprocesses, tackling our particular structure of scopes, a notion of context is first introduced that is inspired by the one introduced in functional languages [11].\footnote{And used in process algebra as well, as in [17].} A context is defined here by the grammar:
A context $E$ is basically a process of the kind of $S$, but with a number of holes $\langle \rangle$ and exactly one final (right-most) hole $\langle \rangle$. For instance, $E_0 = \langle \rangle(\langle \rangle \parallel c?w)$ is an example of context. Observe that holes $\langle \rangle$ substitute scopes, while the final hole $\langle \rangle$ substitutes a whole subprocess. We write $E(\sigma_1, \ldots, \sigma_n)\parallel S_0$ for the process $S \in E$ obtained from $E$ by substituting from left to right each hole $\langle \rangle$ with the scope $(\sigma_i)$, orderly, and the final hole $\langle \rangle$ with $S$. We accordingly write $S \equiv E(\sigma_1, \ldots, \sigma_n)\parallel S_0$. For instance, given the above $E_0$, then notation $E_0(\sigma_1, \sigma_2)\parallel c?w$ is used to mean the process $(\sigma_1)((\sigma_2)c?w)\parallel c?w$. Note that because of commutativity of $\parallel$ and $+$, we were allowed to drop syntax $S + E$ in $E$: this is because $E + S$ already covers the case of a subprocess inside a sum – and similarly for parallel composition. We often denote $E(\sigma_1, \ldots, \sigma_n)\parallel S_0$ simply by $E(\sigma)\parallel S_0$, assuming $\sigma = \sigma_1, \ldots, \sigma_n$. The key interpretation of a process $E(\sigma)\parallel S_0$ is as a process containing the subprocess $S_0$ that runs under the store $\sigma$. In our example above, process $(\sigma_1)((\sigma_2)c?w)\parallel c?w$ has subprocess $c?w$ in it, which runs under the store $\sigma_1, \sigma_2$ – therefore it can be written $E_0(\sigma_1, \sigma_2)\parallel c?w$.

Given context $E$, $\hat{E}$ is used to denote the context obtained from $E$ by dropping choices excluded by the hole $\langle \rangle$, defined as

$$\hat{E} \triangleq \hat{E}$$

The basic property of contexts is that they allow us to reason about the effects of an action prefix by getting rid of the complexity of the store and process it runs into.

**Proposition 1.** Transitions locality. Any transition actually updates a process only locally to a subprocess and to its store; formally:

If $S \xrightarrow{\alpha} S'$ then there exist $E, \sigma, x, R, \sigma'$ so that:

(i) $S \equiv E(\sigma)[[x.R]]$, (ii) $(\sigma)x.R \xrightarrow{\alpha} (\sigma')R$, (iii) $S' \equiv \hat{E}(\sigma')[[R]]$

**Proof.** By induction on the derivation of $S \xrightarrow{\alpha} S'$.

- Rule [SUM]: $S \equiv (\sigma)(S_0 + S_1) \xrightarrow{\alpha} (\sigma')(S'_0) \equiv S'$, where $(\sigma)S_0 \xrightarrow{\alpha} (\sigma')S'_0$. Let $E_0, x, R$ (and the above $\sigma$ and $\sigma'$) be the elements satisfying the inductive hypothesis, that is,

  $$S_0 \equiv E_0(\sigma)[[x.R]] \quad (\sigma)x.R \xrightarrow{\alpha} (\sigma')R \quad S'_0 \equiv \hat{E}_0(\sigma)[[R]]$$

  with $E_0$ of the kind $\langle \rangle E_0$. Then, the context $E \equiv \langle \rangle (E_0 + S_1)$, along with $\sigma, \sigma', x, R$, are the elements satisfying the thesis; in particular, note that $\hat{E}(\sigma')[[R]] \equiv E_0(\sigma')[[R]] \equiv S'$.

- Rule [PAR]: Similar, with $E \equiv \langle \rangle (E_0' \parallel S_1)$.

- Rule [NEST]: Similar, with $E \equiv \langle \rangle E_0$.

- Rule [ASG]: $(\sigma)w \triangleleft e.R \xrightarrow{\alpha} (\sigma \oplus w \mapsto \sigma(e))R$. Let $x$ be $w \triangleleft e$, $\sigma'$ be $\sigma \oplus w \mapsto \sigma(e)$, and $E$ be $\langle \rangle[[\cdot]]$. Then $x, \sigma'$, and $E$, along with $\sigma$ and $R$ above, are the elements satisfying the thesis.

- Rule [SND] and [RCV]: Similar. □

### 3.4.3. Soundness and deadlocks

**Proposition 2.** Transitions preserve well-formedness. Any well-formed process evolves to well-formed processes; formally:

$$\vdash S \text{ ok}, \quad S \xrightarrow{\alpha} S' \quad \Rightarrow \quad \vdash S' \text{ ok}$$

**Proof.** For Proposition 1, $S \equiv E(\sigma)[[x.R]]$ and $S' \equiv \hat{E}(\sigma')[[R]]$, where $(\sigma)x.R \xrightarrow{\alpha} (\sigma')R$. By induction on the structure of $E$, it is easy to show that $\vdash (\sigma)x.R$ ok.

We prove here that $\vdash (\sigma')R$ ok as well. For construction, there exist $W$ and $W'$ (uniquely identified by $\sigma$) such that $W, W' \vdash x.R$ ok: we should prove that $W_0$ and $W'_0$ identified by $\sigma'$ are such that $W_0, W'_0 \vdash R$ ok as well.
• $x$ is $w \prec u$. We have that $w \in W$ and $W', W' \cup \{w\} \vdash R \oka$, and $\sigma' = \sigma \oplus w \mapsto u$. Correspondingly, $W_0 = W$ and $W'_0 = W' \cup \{w\}$ prove the thesis.
• $x$ is $w \prec w'$. Similar, with $w, w' \in W, w' \in W'$, and $W_0 = W, W'_0 = W' \cup \{w\}$.
• $x$ is $e!\overline{w}$. Similar, with $\overline{w} \subseteq W$ and $\overline{w} \subseteq W'$, and $W_0 = W, W'_0 = W'$.
• $x$ is $e?\overline{w}$. Similar, with $\overline{w} \subseteq W$, and $W_0 = W, W'_0 = W' \cup \overline{w}$.

Now, it is easy to show that in general, $\vdash E(\sigma)[S] \oka$, $\vdash (\sigma)S \oka$, and $\vdash (\sigma')S' \oka$ imply $\vdash E(\sigma)[S'] \oka$. Hence from the hypothesis and from $\vdash (\sigma')R \oka$ the thesis follows. □

**Proposition 3.** Progress. Wrong assignment of properties is the only cause for well-formed processes to deadlock; formally:

In any well-formed deadlock process $R$, i.e., where

\[ \vdash R \oka \quad \text{and} \quad R \not\not \quad \text{and} \quad R \neq 0 \]

for any $\sigma, E, S$ such that $E(\sigma)[S] \equiv R$ we have that:

(i) $S$ is of the kind $p \prec e.S'$, (ii) $\sigma(p) \neq \sigma(e)$

**Proof.** For hypothesis, and for Proposition 1, we have that (i) $S = x.R$, (ii) $\vdash (\sigma)S \oka$, and (iii) $(\sigma)S \not\not$. Let $W$ and $W'$ be those sets such that $W, W' \vdash S \oka$, then:

• $x$ is $v \prec w$. Then $w \in W'$, $\sigma(w) \neq \bot$, and $\sigma \oplus v \mapsto \sigma(w) \neq \bot$, hence $S \overset{\tau}{\rightarrow} E(\sigma)[[R]]$ would be a valid transition violating the thesis.
• $x$ is $v \prec u$. This case would similarly allow for a valid transition.
• $x$ is $e!\overline{w}$. This case would similarly allow for a valid transition.
• $x$ is $e?\overline{w}$. This case would similarly allow for a valid transition.

Hence, the only case that do not allow for transitions is $S \equiv p \prec e.R$, where $\sigma(p) \neq \sigma(e)$. □

### 4. Handling correlation

We extend the calculus for service instances introduced in previous section with the true notion of message correlation and business process. By the simple extension we provide, a process can act as a business process in charge of spawning a number of similar service instances, each characterised by a different assignment of the correlation set.

#### 4.1. Core language for correlation

We consider an extended fragment of our language, whose syntax is as follows:

\[
S, R ::= 0 \\
| (\sigma)S | x.S | S + R | S \parallel R \quad \text{Service constructs} \\
| [\pi : S]R \quad \text{Instance spawn}
\]

An instance spawn $[\pi : S]R$ represents a business service (here called an orchestration service) running process $R$, whose first action causes a new service instance to be spawned, having $S$ as the parallel composition of the service instances already spawned, and $\pi$ as the correlation set characterising such service instances. Initially, $S$ is typically 0, for it is likely that no spawned process instances exist. On the other hand, as a result of our operational semantics, as far as interactions occur $S$ grows becoming the composition of several service instances, which then eventually complete becoming 0. In particular, the first action of $R$ is responsible for an instance creation, likewise a receive
action in BPEL whose flag initiate is set to true. As an action with label \( \alpha \) occurs, \( R \) actually spawns a service instance defined as the continuation of \( R \) to action \( \alpha \).

The formal semantics of the instance spawn construct is defined by the following operational rules:

\[
\alpha \rightarrow \frac{S_0 \rightarrow \alpha S'_0}{(\sigma)[\pi : S \parallel S_0]R \rightarrow (\sigma)[\pi : S \parallel S'_0]R} \quad [\text{INST}]
\]

\[
\alpha \rightarrow \frac{(\sigma, \pi) \rightarrow (\sigma', \rho)R'}{(\sigma)[\pi : 0]R \rightarrow (\sigma)\{\pi : (\sigma' \oplus \rho)\}R} \quad [\text{SPW0}]
\]

\[
\frac{(\sigma, \pi) \rightarrow (\sigma', \rho)R' \quad \sigma_0 \oplus \rho = \bot \quad (\sigma)[\pi : S_1]R \rightarrow \alpha S_2}{(\sigma)[\pi : S_1 \parallel (\sigma_0)S_0]R \rightarrow (\sigma)\{\pi : (\sigma_0)S_0 \parallel (\sigma' \oplus \rho)\}R} \quad [\text{SPWR}]
\]

Rule [INST] is used to allow any spawned service instance \( S_0 \) to carry on in isolation.

Rule [SPW0] handles the first spawning of a service instance by process \( (\sigma)[\pi : 0]R \). Suppose action \( \alpha \) can occur on process \( R \) under the store \( \sigma, \pi \) – that is, the correlation set is seen as an inner initialised store of (correlated) properties. Also, suppose the effect of \( \alpha \) would be to move from process \( R \) to continuation \( R' \), \( \sigma \) to \( \sigma' \) (due to, e.g., some new assignment of variables), and \( \pi \) to \( \rho \). In particular, in that case \( \rho \) would contain the assignments of the correlation set \( \pi \) that will characterise the new service instance \( R' \). As a result of these preconditions, the business process allows for the creation of a new process instance, made by \( R' \) running on the store \( \sigma' \oplus \rho \) – representing \( \sigma' \) after propagating all the assignments of the correlated properties.

Rule [SPWR] is responsible for handling all subsequent spawns, taking care that the new service instance has an assignment of correlated properties, which is different from all other existing service instances. As for rule [SPW0], we consider as precondition that \( \alpha \) can occur on process \( R \) under the store \( \sigma, \pi \), and call \( \rho \) the resulting assignment of the correlation set. Action \( \alpha \) is actually allowed if \( \rho \) is different from those of the other service instances. To perform this check, rule [SPWR] picks an existing service instance \( (\sigma_0)S_0 \) and verifies whether \( \sigma_0 \oplus \rho = \bot \). If this is the case – \( \rho \) and \( \sigma_0 \) are incompatible, thus different – the rule propagates the check to all other existing service instances \( S_1 \), recursively. Note that rule [SPW0] guarantees the fixpoint be reached. If the propagation successfully terminates the preconditions are satisfied, hence the new process instance \( (\sigma' \oplus \rho)R' \) is created similarly to rule [SPW0].

As an example, let \( \sigma \) be the store \( v \mapsto 1 \), \( \pi \) be the correlation set \( p \mapsto \text{null} \), and \( R \) the process \( c_a ? p. c_b !(v, p) \). Basically, \( R \) receives a message from \( c_a \) carrying the new value of a property, which is then sent along with \( v \) to channel \( c_b \). The initial process \( (\sigma)[\pi : (\sigma, p \mapsto 1)c_b !(v, p)]R \) represents \( R \) spawning new processes with different values for the property \( \rho \), with the initial process instance \( (\sigma, p \mapsto 1)c_b !(v, p) \) where \( p = 1 \). When a new message \( c_a ? 3 \) arrives, rules [SPWR] and [SPW0] allow for the creation of a new process instance handling it, by the derivation (from bottom to up, the sequence of rules [SPWR,SPW0,INST] is applied):

\[
\frac{(\sigma, \pi) \rightarrow \alpha (\sigma, p \mapsto 3)c_b !(v, p)}{(\sigma)[\pi : 0]R \rightarrow (\sigma)[\pi : (\sigma, p \mapsto 3)c_b !(v, p)]R \quad (\sigma)[\pi : (\sigma, p \mapsto 1)c_b !(v, p)]R \rightarrow (\sigma)[\pi : (\sigma, p \mapsto 1)c_b !(v, p) \parallel (\sigma, p \mapsto 3)c_b !(v, p)]R}
\]

4.2. Basic correlation pattern

The basic example of correlation pattern that can be realised with this new construct is as follows.

The first operation of the process we consider is reception of a message providing a session identifier, like the ID of a purchase in a virtual market. That ID is to be assigned to a specific instance of the orchestration service, in charge

---

8 The meaning of the construct \( [\pi : S]R \) is in fact closely related to the specification \( \lambda x(y). P \) in \( \pi \)-calculus – see Section 6 for more details on that.
of handling all the subsequent interactions relative to that purchase – carrying the same ID. Consider the following specification:

\[
S = (\sigma)[\pi : 0]R
\]

\[
= (p_{ID} \mapsto \text{null}, v \mapsto \text{null})((p_{ID} \mapsto \text{null}) : 0)[c_a ? (p_{ID}, v).c_b ? (p_{ID}, v).S']
\]

It defines the process \((\sigma)R\), whose first receiving action spawns a new service instance characterised by correlation set \(p_{ID}\). For instance, by receiving message \((1001, 5)\) we have the transition:

\[
S \xrightarrow{c_a ? (1001,5)} (\sigma)[\pi : (p_{ID} \mapsto 1001, v \mapsto 5)c_b ? (p_{ID}, v).S']R
\]

The service instance \((p_{ID} \mapsto 1001, v \mapsto 5)c_b ? (p_{ID}, v).S'\) has been spawned, which will handle the session with \(p_{ID} = 1001\). As the new message \((2207, 3)\) is received, we have the transition

\[
(\sigma)[\pi : (p_{ID} \mapsto 1001, v \mapsto 5)c_b ? (p_{ID}, v).S']R \xrightarrow{c_a ? (2207,3)}
\]

\[
(\sigma)[\pi : ((p_{ID} \mapsto 1001, v \mapsto 5)c_b ? (p_{ID}, v).S')] ||
(\sigma)[\pi : ((p_{ID} \mapsto 2207, v \mapsto 3)c_b ? (p_{ID}, v).S')]R
\]

where a new, similar service instance is created. After some more messages the situation of the whole process is as follows:

\[
(\sigma)[\pi : ((p_{ID} \mapsto 1001, v \mapsto 5)c_b ? (p_{ID}, v).S')] ||
(\sigma)[\pi : ((p_{ID} \mapsto 2207, v \mapsto 3)c_b ? (p_{ID}, v).S')]R \xrightarrow{c_a ? (504,3)}
\]

\[
(\sigma)[\pi : ((p_{ID} \mapsto 2207, v \mapsto 3)c_b ? (p_{ID}, v).S')]R \xrightarrow{c_a ? (3123,-1)}
\]

Now, during the creation of these new service instances, some messages can be received through channel \(c_b\), which are to be routed to the correct service instance. This is automatically achieved thanks to our management of stores and properties. Suppose a message \((2207, 2)\) is received, the only allowed transition is the following, enabled by rule [INST]:

\[
(\sigma)[\pi : ((p_{ID} \mapsto 1001, v \mapsto 5)c_b ? (p_{ID}, v).S')] ||
(\sigma)[\pi : ((p_{ID} \mapsto 2207, v \mapsto 3)c_b ? (p_{ID}, v).S')] ||
(\sigma)[\pi : ((p_{ID} \mapsto 504, v \mapsto 3)c_b ? (p_{ID}, v).S')] ||
(\sigma)[\pi : ((p_{ID} \mapsto 3123, v \mapsto -1)c_b ? (p_{ID}, v).S')]R \xrightarrow{c_b ? (2207,2)}
\]

This is because the property update \(p_{ID} \mapsto 2207\) is allowed only for one service instance, the one with store \((p_{ID} \mapsto 2207, v \mapsto 3)\). On the other hand, receiving, e.g., message \((2206, 1)\) through \(c_b\) is structurally prevented, for it is compatible with no current service instance – such a prevention can be interpreted as that message causing an exception, to be properly handled [3,18]. This is the core of the correlation mechanism for orchestration languages analysed in this paper, which is in fact meant to “... [provide a] declarative mechanism to specify correlated groups of operations within a service instance.” [3]. In particular, such a mechanism pertains both instance identification and message routing towards it.
An additional feature of our model – which is not shared with the language we described in [7] – is that the identity of service instances is guaranteed to be unique. Suppose in the above situation a new message \((1001, 0)\) is actually received. Thanks to rule [SPWR] no transition can be applied that allows this message, because a service instance already exists with the assignment \(p_{ID} \mapsto 1001\). As a result, this prevents a new service instance with \(p_{ID} \mapsto 1001\) to be created, so that the value assigned to the correlation set – \(p_{ID}\) in this case – can be really treated as the service instance unique identity. Again, at the implementation level, receiving a message that allows no transition should raise an exception.

4.3. Other correlation patterns

The correlation pattern described above is actually the most common case: a message is received that contains an ID, and characterises the rest of the specific conversation. In this case, the orchestration process is a follower of the conversation, initiated by the Web Service which sent the message. Still, other patterns can actually occur, some of which are described in turn.

Sets of correlation

The main reason why correlation is expressed in terms of “sets”, is that more parts of a message might be used to correlate operations. For instance, these could be the first and third part of the first message received, as in the following example:

\[
S = (\sigma)[\pi : 0]R \\
= (p_1, p_2, v \mapsto \text{null})[(p_1, p_2 \mapsto \text{null}) : 0]c_a?(p_1, v, p_2) \cdot c_b!(p_2, p_1, v)
\]

The two messages containing \((100, 1, 101)\) and \((100, 2, 102)\) are not correlated, for their reception leads to the evolution:

\[
\frac{c_a?(100,1,101)}{(\sigma)[\pi : 0]R} \frac{c_a?(100,2,102)}{(\sigma)[\pi : (p_1 \mapsto 100, p_2 \mapsto 101, v \mapsto 1)c_b!(p_2, p_1, v)]R (\sigma)[\pi : (p_1 \mapsto 100, p_2 \mapsto 101, v \mapsto 1)c_b!(p_2, p_1, v) \parallel (p_1 \mapsto 100, p_2 \mapsto 102, v \mapsto 2)c_b!(p_2, p_1, v)]R}
\]

In this example, the correlation set \(p_1, p_2\) characterises the identity of each service instance. In an actual orchestration language such as BPEL, this is realised through a correlation set with two properties \(p_1\) and \(p_2\), the former aliasing the first part of the message from \(c_a\) and the second part of that from \(c_b\), the latter aliasing the third part of the message from \(c_a\) and the first part of that from \(c_b\).

Multiple start activities

Another interesting case is when the orchestration process is actually made of two parallel subprocesses, which should be correlated through a unique ID. In particular, each of the two subprocesses initially receives a message carrying the ID: as the first is received the service instance is to be spawned waiting for the other message. This example is called in BPEL specification “multiple start activities”, and is used to model, e.g., an auction orchestration, where data are collected separately from the buyer and the seller. This correlation pattern is naturally modelled in our language through the following specification:

\[
S = (\sigma)[\pi : 0]R \\
= ((p_{ID}, v_1, v_2) \mapsto \text{null})[(p_{ID} \mapsto \text{null}) : 0](c_a?(p_{ID}, v_1) \parallel c_b?(p_{ID}, v_2))
\]

In this process, couples of messages coming from \(c_a\) and \(c_b\), respectively, are correlated and routed to the same service instance. In the following evolution, for instance, messages \((1001, 1)\) and \((2207, 3)\) are received from \(c_a\), then \((753, 2)\) and \((1001, 5)\) from \(c_b\): the former and latter messages are routed to the same instance as expected.
Consider the following evolution:

\[
\begin{align*}
(\sigma)[\pi : 0]R &\xrightarrow{c_a ?(1001,1)} \\
(\sigma)[\pi : ((p_{ID} \mapsto 1001, v_1 \mapsto 1, v_2 \mapsto \text{null})c_b ?(p_{ID}, v_2))]R &\xrightarrow{c_b ?(2207,3)} \\
(\sigma)[\pi : ((p_{ID} \mapsto 2207, v_1 \mapsto 3, v_2 \mapsto \text{null})c_b ?(p_{ID}, v_2))]R &\xrightarrow{c_b ?(753,2)} \\
(\sigma)[\pi : ((p_{ID} \mapsto 1001, v_1 \mapsto 1, v_2 \mapsto \text{null})c_b ?(p_{ID}, v_2))]R &\xrightarrow{c_b ?(1001,5)} \\
\end{align*}
\]

**Nested correlation**

A final interesting case of correlation pattern allowed by our language is that of nested correlations, which occurs when two instance spawn constructs are nested. This can be used when a service instance is created to handle messages carrying some ID, and then this service becomes itself a source of new service instances handling messages carrying ID as well as a new identifier ID. An example application is when an orchestration process creates instances to handle the purchase of a good, using ID to characterise that purchase, then the instance itself tries to set up a proper shipment of that order, by creating new instances each in charge of handling communication with a different shipment service, using ID. This pattern is modelled by the specification:

\[
\begin{align*}
S &= (\sigma)[\pi_1 : 0]R \\
&= (\sigma)[p_1 \mapsto \text{null} : 0]c_a ?p_1.[\pi_2 : 0]R' \\
&= ((p_1, p_2) \mapsto \text{null})[p_1 \mapsto \text{null} : 0]c_a ?p_1.[p_2 \mapsto \text{null} : 0]c_b ?(p_1, p_2).S'
\end{align*}
\]

Consider the following evolution:

\[
\begin{align*}
(\sigma)[\pi_1 : 0]R &\xrightarrow{c_a ?1001} \\
(\sigma)[\pi_1 : ((p_1 \mapsto 1001, p_2 \mapsto \text{null})[\pi_2 : 0]R') \xrightarrow{c_b ?2207} \\
(\sigma)[\pi_1 : ((p_1 \mapsto 2207, p_2 \mapsto \text{null})[\pi_2 : 0]R') \xrightarrow{c_b ?(2207,131)} \\
(\sigma)[\pi_1 : ((p_1 \mapsto 2207, p_2 \mapsto \text{null})[\pi_2 : 0]R') \xrightarrow{c_b ?(2207,150)} \\
\end{align*}
\]

Messages \(c_a ?1001\) and \(c_b ?2207\) create two first-level instances, then, messages \(c_b ?(2207, 131)\) and \(c_b ?(2207, 150)\) are routed to the second instance, and lead to the creation of two second-level instances in it, characterised by stores \((p_1 \mapsto 2207, p_2 \mapsto 131)\) and \((p_1 \mapsto 2207, p_2 \mapsto 150)\).

**4.4. Basic properties**

We introduce an extended form of contexts \(\mathbb{E}\) that apply in this new fragment of the language and generalises over previous contexts:
\[ E ::= [] | ()E \mid E + S \mid E \parallel S \]
\[ E ::= [] (\sigma)E \mid ()E \mid E + S \mid E \parallel S \mid [\pi : E]S \]

The main difference between environment \( E \) and \( E \) is that in \( E \) a hole of the kind \([[]]\) can propagate inside spawned activities \( (\pi : E)S \), but in this case \((()[])]\) holes occur only inside the inner-most spawned activity in which \([[]]\) occurs. For instance, \((()[])\) and \([\pi : ()[])]S\) are valid elements of \( E \), whereas \((()[][\pi : ()[])]S\) is not – for a hole \((())\) occurs outside the process of the kind \( E \), that is, it cannot provide a spawning construct. We use this mechanism since a spawned instance runs in isolation, i.e., it never affects stores outside the spawn construct, due to rule \([\text{INST}]\).

A specification \( S \) in this new fragment is said to be well-formed, reusing notation \( \vdash S \) (and \( W, W' \vdash S \) ok), if other than the rules defined in Section 3.4.1, the following rule holds:

\[ W, W' \vdash [\pi : S]R \] ok \quad \text{if} \quad W, W' \vdash R \] ok \quad \text{and} \quad \vdash S \] ok

That is, while the process \( R \) should adhere to the usual rules, \( S \) should be well-formed per se.

**Proposition 4.** Extension. Propositions 1 and 2 directly apply to this new fragment of the language by replacing environments \( E \) with \( E \).

Progress property requires instead a different definition of deadlock, applying to instances only, since a spawning construct always allow transitions.

**Definition 5.** \( R \) is said to be a spawn-free instance in \( S \), write \( sf(S, R) \), if \( R \) occurs as a service instance in \( S \), and \( R \) has no pending spawning constructs in it. Formally, there should exist \( E_0, \sigma_0, \pi_0, S_0 \) such that \( S = E_0(\sigma_0)[[\pi_0 : R]S_0] \), and for all \( E_1, \sigma_1, S_1 \) such that \( R = E_1(\sigma_1)[[S_1] \), we have that \( S_1 \) is not of the kind \([\pi' : R']S' \) (for any \( \pi', R', S' \)).

**Proposition 6.** Extended progress. In a well-formed process, wrong assignment of properties is the only cause for spawn-free instances to deadlock; formally:

\[ \vdash R \] ok \quad \text{and} \quad sf(R, R') \quad \text{and} \quad R' \not\rightarrow \quad \text{and} \quad R' \not\equiv 0 \]

for any \( \sigma, E, S \) such that \( E(\sigma)[[S]] \equiv R' \) we have that:

(i) \( S \) is of the kind \( p \triangleq e.S' \), (ii) \( \sigma(p) \neq \sigma(e) \)

**Proof.** This theorem directly follows from the fact that \( R' \) has no spawning constructs, hence it is the same as in the previous fragment of the language. \( \square \)

5. Towards full orchestration languages

Among the various applications of the calculus introduced in previous section, an interesting one is to see it as a “core” upon which a full-featured orchestration language can be defined. To show this application we extend here our language with few constructs making it sufficiently expressive to model a number of real-case orchestration services.

5.1. An extended language

In order to equip our framework with the ability of performing algorithmic computation, expressions are extended as follows:

\[ e ::= w \mid u \mid f(\overline{e}) \]

Meta-variable \( f \) ranges over function symbols, used to take into account simple data-manipulation operators such as integer sum. Accordingly, we add the following rule for store evaluation:

\[(w \mapsto u)(f(\overline{e})) \triangleq f \left( (w \mapsto u)(e) \right)\]
We also assume from here on that \( \text{true} \) is an allowed value. In this extended language, it makes sense, e.g., to evaluate expression \( v + (p \ast 2) \) in store \( (v \mapsto 5, p \mapsto 2) \), by viewing \(+\) and \(\ast\) as binary functions (written \( f_+ \) and \( f_\ast \)) over integer values:

\[
(v \mapsto 5, p \mapsto 2)(f_+(v, f_\ast(p, 2))) = 9
\]

We denote by syntax \( W(e) \) the set of locations appearing in expression \( e \). An example of core language for orchestration services extending the previous framework is then as follows:

\[
S, R ::= 0 | (\sigma)S | x.S | S + R | S \parallel R
\]

Service constructs

\[
\left\lfloor\pi : S\right\rfloor R | \text{Instance spawn}
\]

New constructs

Process \( \text{wh}(e)\{S\}.R \) realises a \textit{while} construct: as long as expression \( e \) is evaluated to \( \text{true} \) (in the current store) \( S \) is executed, otherwise \( R \) is taken as continuation. Process \( \text{sw}(e)\{S : R\} \) models activity \textit{switch}, and acts basically as an \textit{if-then-else} construct: if \( e \) is evaluated to \( \text{true} \), then \( S \) is executed, otherwise \( R \) is executed. Process \( S; R \) is the sequential composition of \( S \) and \( R \), that is, \( R \) is allowed to execute when \( S \) terminates. Finally, \( \star l.S \) and \( o.l.R \) are constructs used to synchronise subprocesses through links, ranged over by meta-variable \( l \): when a process \( S \) has the source link \( \star l \) as prefix, and a concurrent process \( R \) has the target link \( o.l \) as prefix, then \( R \) and \( S \) are allowed to carry on.

The semantics of these new constructs is introduced by the congruence rules

\[
0; S \equiv S \quad (R; R') \equiv (S; R) \quad (R; R') \equiv S; R
\]

and by the operational rules:

\[
\frac{\alpha}{\sigma S \xrightarrow{\alpha} (\sigma')S'} \quad \text{[SEQ]}
\]

\[
\frac{\sigma(e) \neq \text{true}}{(\sigma)\text{wh}(e)\{S\}.R \xrightarrow{\tau}(\sigma)R} \quad \text{[WH1]}
\]

\[
\frac{\sigma(e) = \text{true}}{(\sigma)\text{wh}(e)\{S\}.R \xrightarrow{\tau}(\sigma)S; (\text{wh}(e)\{S\}.R)} \quad \text{[WH2]}
\]

\[
\frac{\sigma(e) \neq \text{true}}{(\sigma)\text{sw}(e)\{S : R\} \xrightarrow{\tau}(\sigma)R} \quad \text{[SW1]}
\]

\[
\frac{\sigma(e) = \text{true}}{(\sigma)\text{sw}(e)\{S : R\} \xrightarrow{\tau}(\sigma)S} \quad \text{[SW2]}
\]

\[
(\sigma)(\star l.S \parallel o.l.R) \xrightarrow{\tau}(\sigma)(S \parallel R) \quad \text{[LNK]}
\]

Rule [SEQ], along with the above congruence rules, defines the semantics of sequential composition for processes, which is used to extend the sequencing expressiveness of action prefix.

\footnote{Note that this construct is not a necessary one, for continuation to action prefix is typically considered a sufficient enough mechanism to express sequences \cite{5}. However, it is introduced here to more easily mimic the \texttt{<sequence>} construct of orchestration languages. On the other hand, prefixing of actions is not dropped to let this new language be a syntactic extension of our core calculus.
Rules [WH1, WH2, SW1, SW2] define straightforward semantics of while and if-then-else constructs, based on our mechanism of expressions evaluations through stores.

Rule [LNK] defines a simple semantics to the synchronisation mechanism of links. The link construct is used in orchestration languages such as BPEL to express complex dependencies between subprocesses – and it typically comes with additional features such as multiple join conditions. Here we are not interested in fully modelling the link mechanism – this aspect is already covered in other papers, such as [8]. Our semantics resembles that of channels in CCS and π-calculus, and is sufficiently expressive to denote quite sophisticated cases, as the flow-graph example in BPEL specification (Section 12.5.3), which is expressed as follows:

$$S ::= (S_1; \bullet_{i_0}) \parallel (S_2; \bullet_{l_0}) \parallel (o_{l_0}; o_{l_0}; S_3; \bullet_{l_0}; \bullet_{l_0}) \parallel (o_{l_0}; S_4) \parallel (o_{l_0}; S_5)$$

Activity $S_3$ is executed only when $S_1$ and $S_2$ terminate, then, when $S_3$ terminates both $S_4$ and $S_5$ can execute.

5.2. Properties

It can be shown that the properties studied in previous sections hold to a certain extent in this language, though some modifications need to be applied. For instance, whereas it is true that the trivial extension of the well-formedness property ensures that stores are always accessed in a consistent way, other forms of deadlock can occur because of the links construct – e.g., a source link might never synchronise with a target link.

A specification $S$ in this language is said to be well-formed, reusing notation $\vdash S \triangleq \omega$ (and $W, W' \vdash S \omega$), if other than the rules defined in previous sections, the following hold:

$$W \cup W(e), W' \cup W(e) \vdash wh(e)\{S\}, R \omega \quad \text{if} \quad W \cup W(e), W' \cup W(e) \vdash S, R \omega$$

$$W \cup W(e), W' \cup W(e) \vdash sw(e)\{S : R\} \omega \quad \text{if} \quad W \cup W(e), W' \cup W(e) \vdash S, R \omega$$

$$W, W' \vdash \bullet_{l.S}, o_{l.S} \omega \quad \text{if} \quad W, W' \vdash S \omega$$

$$W, W' \vdash S ; R \omega$$

if

$$W, W' \cup W+(S) \vdash R \omega$$

The rule for the while construct states that processes $S$ and $R$ must be well-formed under the same hypothesis of the construct $wh(e)\{S\} R$: there, variables used in the evaluated expression $e$ (that is, $W(e)$) are to be both defined and initialised. The switch construct is handled similarly; the links construct case is straightforward. In the case of process $S ; R$, it should be noted first that any definition of new variables in $S$ – due to an inner scope in it – do not affect $R$, whereas initialisation of already defined variables can, because of rule [SEQ] affecting the outer scope. Hence, on the one hand $R$ can be checked considering the same definition of variables that $S ; R$ is checked against (that is $W$). On the other hand, while checking $R$ we can suppose to have all the variable initialisations used for $S ; R$ (that is $W'$) plus the further variables initialised in $S$ (those that are not defined inside $S$). This latter contribution is given by set $W^+(S)$, recursively defined as follows:

$$W^+(0) \triangleq \{\} \quad W^+((\sigma, \sigma')S) \triangleq W^+((\sigma)(\sigma')S)$$

$$W^+(w \rightarrow u.S) \triangleq W^+(S) \setminus \{w\} \quad W^+(w \rightarrow \text{null})S) \triangleq W^+(S) \setminus \{w\}$$

$$W^+(w < e.S) \triangleq W^+(S) \cup \{w\} \quad W^+(c\vec{m}.S) \triangleq W^+(S) \cup \vec{m}$$

$$W^+\{S + R\} \triangleq W^+(S) \cap W^+(R) \quad W^+(S \| R) \triangleq W^+(S) \cup W^+(R)$$

$$W^+(wh(e)\{S\}, R) \triangleq W^+(R) \quad W^+(sw(e)\{S : R\}) \triangleq W^+(S) \cap W^+(R)$$

$$W^+(o_{l.S}) \triangleq W^+(S) \quad W^+(\bullet_{l.S}) \triangleq W^+(S)$$

$$W^+(S ; R) \triangleq W^+(S) \cup W^+(R)$$

Regarding contexts, note that constructs while, switch, and link never allow a prefix process to carry on (differently from both “∥”, “+”, and “;”), hence contexts $E$ and $\Xi$ are simply extended in this new fragment as follows:

$$E ::= \ldots | E ; S$$

$$\Xi ::= \ldots | \Xi ; S$$
The progress result is then formulated as follows.

**Proposition 7.** Weak extended progress. In a well-formed process (in the extended language), wrong assignment of properties is the only cause for prefixed spawn-free instances to deadlock; formally:

In any well-formed process $R$, and in any deadlock spawn-free instance $R'$ in $R$, i.e., when:

$$\vdash R \text{ ok and } sf(R, R') \quad \text{and} \quad R' \not\rightarrow \quad \text{and} \quad R' \neq 0$$

for any $x, \sigma, E, S$ such that $E(\sigma)\{x.S\} \equiv R$ we have that:

1. $x.S$ is of the kind $p \prec e.S$,
2. $\sigma(p) \neq \sigma(e)$

5.3. A shipping service example

To provide an example of application of this language, we describe the specification of the shipping service shown in Section 2.2.

Two channels are used: $cn$ represents action `shippingNotice` invoked on the customer, and $cr$ is used for invocation of action `shippingRequest` by the customer. Symbol $\overrightarrow{r}$ is used for variable `shipRequest`, containing received request messages, which is made by the three parts $v_{r1}$, $v_{r2}$, and $v_{r3}$ representing `orderID`, `complete`, and `itemsTotal`. Symbol $\overrightarrow{n}$ is used for variable `shipNotice`, containing the messages to send, which is made by the two parts $v_{n1}$ and $v_{n2}$, representing `orderID`, and `itemsCount`. Then, symbol $\overleftarrow{i}$ is used for variable `itemsShipped`. Property $p$ of the correlation set is used for the `orderID`. The abstract syntax of the shipping service is then quite directly obtained from the BPEL specification, as shown in Fig. 2.

The most notable difference with respect to the original specification is due to the management of aliases, which is used in BPEL to bound a property to a message part. For instance, when receiving message $(v_{r1}, v_{r2}, v_{r3})$, property $p$ is automatically bound to the first part of the message, whereas in our language this is to be realised by receiving to locations $(p, v_{r2}, v_{r3})$ and then assigning $p$ to $v_{r1}$. The case of message sending is handled dually: the message part is to be assigned with the current value of the property before sending the message.

```
(p \mapsto \text{null})
(v^r_1 \mapsto \text{null}, v^r_2 \mapsto \text{null}, v^r_3 \mapsto \text{null})
(v^i \mapsto \text{null}, v^n_1 \mapsto \text{null}, v^n_2 \mapsto \text{null})
[p \mapsto \text{null : 0}]
cr?(p, v^r_2, v^r_3);
\overleftarrow{i} < p;
su(v^r_2)\{v^n_2 < v^n_3; v^i_1 < p; c_n!(v^n_1, v^n_2)\}
: \quad v^i < 0; wh(v^i < v^n_3)\{
\quad v^n_2 < 1;
\quad c_n!(p, v^n_2);
\quad v^i < v^i + v^n_2
\}
```

Fig. 2. Abstract syntax for the shipping service.
6. Related work

6.1. Correlation in core calculi for interaction

The idea of correlation is strictly related with that of a server handling requests by spawning each time a different subprocess (or execution thread). In foundational calculi, this scenario can be modelled in $\pi$-calculus [6] relying on the infinite replication and restriction constructs. Consider the following specification:

$$BProc(x) := !x(y).(\nu z)yz.\text{Instance}(z)$$
$$Client(x, y) := \exists y.y(z).\text{Session}(z)$$

A business process $BProc$ is an infinite replication of processes receiving at the main port $x$ an address $y$ specifying the client location. As a message is received a new process is conceptually spawned. It creates a fresh name $z$ which is communicated to the client; from then on, agent $\text{Instance}(z)$ will communicate with the client along that private channel $z$. Dually, the client first sends a request to the business process specifying its address $y$, and receives from $y$ a new private channel $z$, which is actually used by $\text{Session}(z)$ to realise the rest of the conversation. This mechanism guarantees each client to have a private conversation with an instance of the server.

Whereas this mechanism is well-known, orchestration languages choose a different one based on correlation through message parts. A possible argument against applying the replication/restriction approach to orchestration languages is that creating new names representing communication channels can be seen as a rather low-level implementation mechanism for business processes, which instead strives to abstract away from details on channels and ports (using the partner links abstraction). Nevertheless, analysing the differences between the two approaches, providing an abstract semantics of correlation based on a translation to $\pi$-calculus, and evaluating new proposals for correlation based on replication/restriction are interesting issues for future researches, which might emphasise interesting connections between the two of them.

The work we present here is based on [7], where a core language for a fragment of BPEL has been introduced which has similar expressiveness to the language introduced in Section 5. As a main difference, our approach here is incremental, allowing to isolate a core of the language featuring only those very few aspects of correlation required to analyse its basic properties, and in the end ascribing correlation and instance spawning to the single construct introduced in Section 4. Moreover, here we improve the work in [7], because we guarantee uniqueness of service instances by preventing reception of messages inducing the same assignments of a correlation set. The semantics in [7] abstracts away from this key feature, since spawning is modelled directly relying on the standard semantics of the replication construct (as in $\pi$-calculus or CCS). Additionally, we moved towards a more typed framework, due to the introduction of (nested) scopes, representing definition of variables and properties. In fact, this enabled the possibility of providing a well-formedness result avoiding trivial run-time errors.

6.2. Models of other aspects of orchestration

Our work here and in [7] addresses a formalisation focussing on the correlation mechanism for orchestration languages for the first time. To the best of our knowledge, the only formal model dealing with correlation is the technical report [19], where a rather large model of BPEL is provided based on Abstract State Machines (ASM). In the style of ASMs, such a model is more the specification of an abstract machine able to execute BPEL specifications – a sort of formal orchestration engine – rather than a true language for studying properties of BPEL. Indeed, this work is valuable for it can make many obscure parts of the official specification clearer, and could be used to systematically compare our work with the existing features of BPEL.

Other researches have been developed studying different peculiar aspects of orchestration languages, and delivering core languages covering fragments with different constructs.

In [8], the BPE-calculus is introduced to study the fragment of BPEL focussing on the links construct, analysing features such as joint conditions and transition conditions for links. A verification methodology is developed for that calculus based on the Concurrency Workbench.\(^\text{10}\)

\(^{10}\) www.cs.sunysb.edu/~cwb.
The issue of compensation of long-running transaction has been studied as well, leading to several research papers. In [9], the StAC language for compensation in business processes is introduced, featuring full support for handling nested compensation activities. A main contribution of this work is to show that a full model of compensation as found, e.g., in BPEL is significantly complex, and might impact with other language features such as event and fault handlers. In [17], the asynchronous π-calculus is extended with few constructs for handling a set of practical examples of compensation. An encoding of this language into π-calculus is then provided which preserves operational semantics. This work has then been improved in [20] to handle composition and nesting of transactions.

Finally, the work in [18] studies aspects related to event (and fault) handlers, by providing a simple core calculus extending the π-calculus with a scope construct inspired by the Ambient calculus [21].

7. Conclusions

In this paper we applied typical techniques for formalising syntax and semantics of imperative and concurrent programming languages to define a core calculus for the correlation mechanism of orchestration languages. This calculus is meant to serve as a starting tool on top of which several studies can be developed, which are all interesting directions for future work. These span from studying verification of properties to devising extensions and adaptations; from analysing integration with other constructs to specifying current orchestration languages and the implementation of orchestration engines.

References