Available online at www.sciencedirect.com

Electronic Notes in Theoretical Computer Science 260 (2010) 3–24

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Formal Specification of Correlation in WS Orchestrations Using BP-calculus

Faisal Abouzaid and John Mullins [1]

CRAC Lab., Computer & Software Eng. Dept., École Polytechnique de Montréal. P.O. Box 6079, Station Centre-ville, Montreal (Quebec), Canada, H3C 3P8. [2]

Abstract

Correlation is an important mechanism used in orchestration languages for Web Services. It expresses means by which many instances of the same service can be carried on at the same time. In this paper we extend the BP-calculus, a language based on the π-calculus and dedicated to the specification of web service orchestrations, with a message algebra, a mechanism of function evaluation and a mechanism of correlation. The mechanism of function evaluation allows message handling while the mechanism of correlation guarantees uniqueness of service instances by preventing reception of messages inducing the same assignments of a correlation set. We also show how it can be used to express the semantics of the BPEL constructs coping with correlation. As an illustration of the usefulness of this process algebraic framework, we terminate with the presentation of a motivating example, the Trade Market example.

Keywords: Web Services; Orchestration languages; Process algebras

1 Introduction

Service Oriented Computing (SOC for short) is emerging as the main paradigm to distribute computation over the Web. It is based on the composition of several services each one providing desired functionalities to its clients.

WS-BPEL is the well admitted standard [3] allowing the definition of WS-Orchestrations, i.e. the description of interactions and messages flow between services in the context of a business process.

Since development of Web applications concerns complex interactions among distributed components, it is an error-prone activity that requires safe development techniques. This is the reason why orchestration languages are largely inspired by process algebras such as CCS [8] and π-calculus [9]: they provide an algebraic

[1] Research partially supported by the author's NSERC grant (Canada)

[2] Email: `mohammed-faical.abouzaid,john.mullins@polymtl.ca`

[3] We base our work on WS-BPEL 2.0 that has been accepted by OASIS as a standard since April 2007.

doi:10.1016/j.entcs.2009.12.029

approach to the study of service composition allowing algebraical languages for the specification of services and the formulation of statements about them, together with calculi for the verification of these statements [11]. One of the most relevant process algebra is the π-calculus [9] because its ability to model mobility.

We have presented in a previous work [2], the BP-calculus, a specification language based on the π-calculus that allows for the verification of WS-BPEL specifications using a refinement/verification process. BP-calculus provides a means to formally verify specifications of business processes and to automatically generate verified WS-BPEL code from these specifications. The refinement is based on a two-way mapping between BP-calculus and WS-BPEL and includes complex structures such as fault and event handlers.

Motivations of the present paper are twofold. First, while the approach of [2] might be considered as a rather low-level implementation mechanism, notably to express the handlers constructs, in this work we extend this approach by introducing a message algebra with functions allowing abstraction from these details. Finally, we also extend the mapping in such way it includes correlation sets and we introduce a formal framework for studying the mechanism of correlation in orchestration languages.

WS-BPEL uses a hierarchy of nested scopes allowing for structuring of business process. Each scope contains variables, partner links, message exchanges, correlation sets, and handlers for handling faults or events. Scoping limits the visibility of these definitions to the enclosed activities and provides the context in which they are executed [10]. Correlation sets are then used to model complex interaction patterns in which process instances are involved. They allow routing the specific invocation parameters (e.g., requests referring to the same customer ID are routed to the same session) to the concerned instance of the invoked service.

Related works

Numerous works have been devoted to the formal specification of business process, WS-BPEL in particular, using process algebra (PA). Several formalisms based on PA have been proposed: SOCK [3], COWS [5] or SSCC [4]. Close to our concern, Lucchi and Mazzara [6] provided the first π-calculus based semantics to BPEL. But at the best of our knowledge Viroli [12] has been the first to provide a framework to address a formalization of the correlation mechanism for orchestration languages. The main difference with the Viroli's approach, is the way we model correlation relying on an expressive message algebra and a powerful recursive definition construct of an extended π-calculus, designed to express orchestrations. By contrast, the incremental approach of [12] results in a core of a language featuring only those few aspects required to analyze basic properties of correlation and abstracts away from other aspects of orchestration languages, such as handlers. We could summarize by saying that our approach integrates the abstract correlation mechanism of [12] to the more concrete Lucchi and Mazzara's framework [6] in order to provide a formal and sufficiently expressive framework that allows verification of real-life web service specifications.

Structure of the paper

The remainder of this paper is organized as follows. The next section, Section 2, introduces WS-BPEL with correlation sets and an illustrating example (Section 2.4). Syntax of the extended BP-calculus is presented in Section 3.2 and its operational semantic in Section 3.3. Encoding of some relevant BPEL constructs into BP-calculus is presented in Section 3.4. A BP-model for the example is given in Section 4. Finally, we conclude the paper in Section 5 by some considerations on future evolutions of this work.

2 Orchestration with Business Process Languages

2.1 WS-BPEL

WS-BPEL [10] 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. This is specified in terms of port types, actions, and messages.

A WS-BPEL specification is made of four declaration parts: the partner links, the variables, the correlation sets, and the activity realizing 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.

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. This feature allows for addressing mobility while coupled with WS-Addressing specification [7].

Correlation sets are used to route a message to a specific instance of an invoked service.

An activity describes the precise behavior of the business process. Basic activities include activities such as sending (*invoke*), receiving (*receive*) requests and replies (*reply*), which can specify one or more existing correlation sets they must adhere to, or new correlation sets to be initialized. Among other basic activities, there are variable assignment (*assign*), synchronization of internal concurrent activities through private source and target links (*links*), waiting for a timeout (*wait*), and raising faults (*throw*). Structured activities realize sequential composition (*sequence*), guarded choice (*pick*), parallel composition (*flow*), iteration cycles (*while*, *foreach* and *repeat*), and conditional (*if then else*).

2.2 Correlation mechanism

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.

In WS-BPEL a receive activity can be defined so that receipt of a message creates a new process instance. Correlation rules can be defined in order to correlate messages with the appropriate instance of a process. Correlation sets are then introduced to identify those interactions that are pertinent to a given process instance; each correlation set is a set of properties, which are aliases for parts of messages. Correlation sets are instantiated while initializing the containing scope.

When a message is sent, a field linked to a property is automatically bound to hold the value associated to that property. Hence, a message is received by a process instance only if the field linked to a property contains the value associated to this property. This mechanism guarantees all the messages sent and received by a process instance to be compliant with the initialization of properties.

2.3 Defining correlation with WS-BPEL

In WS-BPEL the correlations use key fields of data which may uniquely identify the conversation/instance. The correlation set specifications are used in `<invoke>`, `<receive>`, and `<reply>` activities; in the `<onMessage>` branches of `<pick>` activities; and in the `<onEvent>` variant of `<eventHandlers>`.

"*The only way to instantiate a business process in WS-BPEL is to annotate a `<receive>` activity (or a `<pick>` activity) with the `createInstance` attribute set to `yes`. [10]*"

The process of designing correlations in WS-BPEL is illustrated by the following scenario.

1. We first define a *property* (name and data type) in the WSDL file, which will be used by the correlation set. The property name is defined separately because the property can be used by several messages:

```
<bpws:property name="CustomerID" type="xsd:string" />
<bpws:property name="OrderID" type="xsd:string" />
<bpws:property name="BrokerID" type="xsd:string" />
```

2. Then we define a *propertyAlias* for each part of the correlation data, that indicates which part of the message represents the property. The property name may be the same for many aliases:

```
<bpws:propertyAlias messageType="TradeMarket:SellRequest"
          part="ClientAccountNumber" propertyName="CustomerID"/>
<bpws:propertyAlias  messageType="SellRequest"
          part="OrderNumber"  propertyName="OrderID"/>
<bpws:propertyAlias messageType="BrokerResponse"
          part="BrokerAccountNumber" propertyName="BrokerID"/>
```

3. Afterward, we define the `<correlation set>` in the related WS-BPEL process

before any activity. A `<correlationSet>` can be declared within a process or a scope element.

```
<correlationSets>
    <correlationSet name="CustomerCS"
          properties="CustomerID OrderID"/>
    <correlationSet name="BrokerSellOrder"
          properties="BrokerID"/>
</correlationSets>
```

4. Finally, we *reference* the correlation set from within the WS-BPEL definition. One must set the `createInstance` attribute to `"yes"` in the `<receive>` activity. The WS-BPEL engine will create a correlation set instance for each conversation.

```
<receive name="CustomerRequest"
       partnerLink="Customer" portType="CustomerPortType"
       operation="SellRequest" variable="CustomerRequest"
       createInstance="yes">
   <correlations>
        <correlation initiate="yes" set="CustomerSellOrder"/>
   </correlations>
</receive>
.......
<receive name="BrokerNotice"
       partnerLink="Broker"  portType="BrokerPortType"
       operation="OrderNotice"  variable="BrokerRequest">
   <correlations>
       <correlation set="AgentSellOrder"/>
   </correlations>
</receive>
......
```

On receipt of a message the WS-BPEL engine examines the correlation data. If a matching correlation data set instance can be found it will execute the received message.

Note that the correlation data must be sent to any service that may interact with a WS-BPEL process requiring the correlation data. It must also be sent to services that will indirectly call back the WS-BPEL process having need of correlation data. The WSDL of the other services need not be changed to facilitate the correlation set but the WSDL of the WS-BPEL service exposing the service to be consumed must have a provision to receive this correlation data.

The latest remark is important for it compels to broadcast the data to all instances (processes) and to use a conditional statement to test whether the instance is concerned or not with the data.

2.4 A motivating example

To illustrate the correlation mechanism, we consider the following example: a bank agent receive orders requested by customers in order to sell an amount of shares from a company. With all financial services exposed as Web Services, the agent can place his orders by contacting a broker service stating that he wishes to sell a certain amount of shares. The broker service can split its action between several brokers (instances).

The WS-BPEL specification of the agent service starts by specifying the partner link types. Two partners links are specified: one partner link representing the customer service and another one representing the broker service.

The customer invokes the agent service by a one-way request named `sellRequest`, the agent service provides notices by executing one-way invocations to the customer, by an operation named `sellNotice`. Then, selling request messages are defined. They are made of three parts: a `customerOrderID` integer, an `orderOrderID` integer, and a `Quantity` integer denoting the number of shares to be sold. Selling notice messages are made of the `customerOrderID` integer, an `orderOrderID` integer, and the `customerAmountSold` integer, representing the number of shares currently sold by all brokers.

In his turn the agent invokes the broker service by a one-way request named `orderRequest`, the broker service provides notices by executing one-way invocations (named `orderNotice`) to the agent. Ordering request messages are defined as being made of two parts: a `brokerID` integer and a `brokerQuantity` integer denoting the number of shares to be sold by the designed broker. We suppose that the broker sells all the shares at once. Then, selling notice messages are made of the `brokerID` integer and the `amountSold` integer, indicating the amount of shares sold by the designed broker.

The process execution is as follows : as the request is received by the agent, it is split into N parts and the agent invokes the broker services N times. The reception by the broker service of a request, causes the spawning of a new service instance, e.g the attribute `initiateInstance` in the `<receive>` element is set to "yes". A while iteration is executed. Each time a broker sends a selling notice, the count part of the `agentSharesSold` message is incremented by the amount sold. Correspondingly, a message is sent to the customer notifying the number of shares sold. When this number reaches the total amount requested by the customer, the service instance terminates.

This process is illustrated by Figure 1.

A simplified WS-BPEL code of this process is shown in Appendix A.

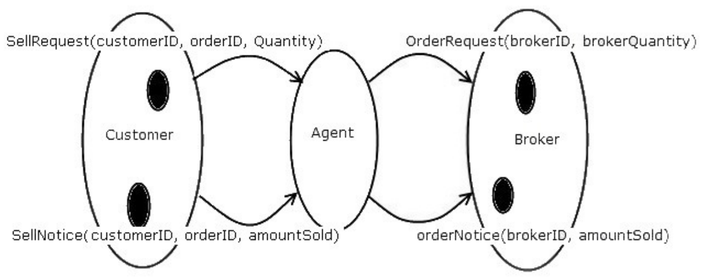
SellRequest(customerID, orderID, Quantity)
OrderRequest(brokerID, brokerQuantity)
Customer
Agent
Broker
SellNotice(customerID, orderID, amountSold)
orderNotice(brokerID, amountSold)

Fig. 1. Trade Market Example.

3 Extending the BP-calculus

3.1 Outline

In order to deal with correlation mechanisms and to simplify the formalization of complex constructs, we provide an extension of the BP-calculus we introduced in [2] with a message algebra and a mechanism of function evaluation capturing message handling. We use an appraoch similar to Abadi and Fournet's applied π-calculus [1] and we rely on proof techniques from concurrency theory.

The calculus we define, is meant to provide a foundation to the verification/refinement process including fault and event handling and 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. Whilst it can be used with other languages than WS-BPEL, it is focused on this specific standard.

The message algebra specifies messages as tuples that are use to model correlation sets. Messages are handled by means of functions (creation, construction, selection or update). Also, the operational semantics has to provide a mechanism for function evaluation, as required for the process to evolve.

3.2 Syntax of extended BP-calculus

Terms.

The set of terms $\mathcal{T}$ consists of variables $\mathcal{V}$, names $\mathcal{N}$ and values $(\mathcal{U})$ (integers, booleans, strings, ...). For each term t, $fv(t)$ is the set of variables in t. A message is a closed term (i.e. not containing variables). The set of messages is denoted $\mathcal{M}$.

Terms allow for the definition of correlations considered as message components carrying correlation data.

Functions.

Primitives that manipulate messages, such as generation or extraction of the correlation set, are modeled by functions in $\mathcal{F} \subseteq [\mathcal{M}^k \rightarrow \mathcal{M}^n]$.

Example of useful functions in this context are :

- The constructor *build*$(M_1, ..., M_n)$ that builds a message from $M_1, ..., M_n$,

- The selector *data(M)* that returns the part of the message containing the significant data,
- The boolean selector *create(M)* that returns a boolean indicating whether to create an instance or not,
- The selector *source(M)* (resp. *target(M)*) that returns the part of the message that identifies the emitting (resp. receiving) service.
- The selector *correlationPart(M)* that returns the parts of the message used for correlation.

Some other functions will be introduced when necessary.

Syntax.

We let $\tilde{x} = (x_1, ..., x_n)$, (resp. $\tilde{a} = (a_1, ..., a_m)$, $\tilde{u} = (u_1, ..., u_m)$) range over the infinite set of n-tuples of variable (resp. name, value) identifiers. We denote $\tilde{x} \leftarrow \tilde{u}$ the assignment of values $\tilde{u}$ to variables $\tilde{x}$.

The syntax of the extended BP-calculus is given in table 1.

A brief informal account of the intended interpretation of the processes follows:

- $\overline{a}^t\langle M\rangle$ ($t \in \{invoke, reply, throw\}$) is the usual output which can be an invocation, or a reply to a solicitation, or the throw of a fault, and which can be translated by a `<reply>`, an `<invoke>` or a `<throw>`. Semantically the annotation does not interfere. $\overline{a}^t\langle\rangle$ is a signal. Annotations of input or output operations are used to ease the translation into WS-BPEL.
- A restriction $(\nu\ x)P$ behaves as P but x is local to P.
- IG is an input guarded process and $IG + IG'$ behaves like a guarded choice and is intended to be translated by a `<pick>`. We do not consider non-determinism in service behavior. An input may be annotated by s indicating whether we catch a fault or in event within a handler.
- $P \rhd_{c(M)} Q$ expresses a sequential composition from process P passing M to Q (Q can perform actions when P has terminated). This construct is introduced here to more easily mimic the `<sequence>` construct of WS-BPEL. Note that it is not a necessary one, for it can be expressed by action prefix and parallel operator. We use the notation $P \rhd Q$ (or $P.Q$) when nothing is transmitted.
- *if then else* expresses a classical choice based on messages identity and replaces the initial names equality. It is intended to be naturally translated by an `if then else` construct in WS-BPEL 2.0.
- $\mathcal{C}$ is a correlation set, i.e a set of specific valued variables within a scope acting as properties and transported by dedicated parts of a message. Given a correlation set $\mathcal{C}$ we will say that an assignment $\tilde{x} \leftarrow \tilde{u}$ does not belong to $\mathcal{C}$ (denoted $\tilde{x} \leftarrow \tilde{u} \notin \mathcal{C}$) iff $\mathcal{C} = null$ or $\mathcal{C} = \mathcal{C}'[\tilde{y} \leftarrow \tilde{v}]$, $(x, u) \neq (y, v)$ and $\tilde{x} \leftarrow \tilde{u} \notin \mathcal{C}'$. This binding holds the data instantiating service instances and has to be unique in order to prevent inconsistent interaction between instances. Intuitively, $[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y})$ represents an orchestration service running a process defined as

Terms			
t	$::=$	x	(metavariables)
	$\mid$	a	(names)
	$\mid$	u	(value)
	$\mid$	$(t_1, \ldots t_k)$	(tuple)
$\mathcal{C}$	$::=$	$null \mid \mathcal{C}[\tilde{x} \leftarrow \tilde{u}]$	(correlation set)
Processes :			
P, Q	$::=$	IG	(input guard)
	$\mid$	$\overline{c}^t\langle M\rangle.P$	(annotated output)
	$\mid$	$P \mid Q$	(parallel composition)
	$\mid$	$P \rhd_{c(M)} Q$	(sequential composition)
	$\mid$	$(\nu\ n)P$	(restriction)
	$\mid$	$if\ M = N\ then\ P\ else\ Q$	(conditional)
	$\mid$	$[\tilde{x} \leftarrow f(M_1, ..., M_n)]P$	(function evaluation)
	$\mid$	$A(x_1, \ldots, x_n)$	(service definition)
	$\mid$	$[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$	(instance spawn)
Guarded choice :			
IG	$::=$	0	(empty process)
	$\mid$	$c^s(u).P$	(annotated input)
	$\mid$	$IG + IG'$	(guarded choice)
Scopes :			
S	$::=$	$\{\tilde{x},\ P,\ H\}$	(scope)
H	$::=$	$\prod_i W_i(P_{i_1}, \cdots, P_{in_i})$	(handlers)
E	$::=$	$S \mid P \mid S\mid P$	(global system)

Table 1
Extended BP-calculus Syntax

$c_A(\tilde{x}).A(\tilde{y})$. A reception of a message M over the dedicated channel c_A causes a new service instance (defined as $A(\tilde{y})$) to be spawned. The process P represents the parallel composition of service instances already spawned, $\mathcal{C}$ the correlation set characterizing instances and $\tilde{y}$ the correlation part of M.

- $[x \leftarrow f(M_1, ..., M_n)]P$ assigns the value $f(M_1, ..., M_n)$ to variable x before executing process P. For instance, $[x \leftarrow build(M_1, ..., M_n)]\overline{c}\langle x\rangle$ means that the n-tuple

M is built from components $M_1, ..., M_n$ before being sent over the channel c.

Input $c(u).P$ binds the names u and c. The scope of this binder is the process P and its instances. The free and bound names of processes are noted $fn(P)$ and $bn(P)$ respectively.

3.2.1 Scopes and handlers

We present in this section a mechanism that abstracts the WS-BPEL scopes. Scopes act as containers for WS-BPEL processes and handlers. A scope contains a primary structured activity which defines its normal behavior; it might contain variable definitions and handlers (fault, compensation, event and termination handlers). In case of normal execution, a scope is activated at the same time as its activities are and terminates when all its activities have been accomplished.

In order to formally verify the handlers' behavior, we need to provide their formal semantics in term of BP-calculus. We formalize these handlers by means of contexts.

Let $S ::= \{\tilde{x},\ P,\ H\}$ be a scope, with handlers $H ::= \prod_i W_i(P_{i_1}, \cdots, P_{in_i})$. Then,

- $\tilde{x}$ are the local variables of the scope, and P its primary activity,
- H is the scope's execution environment that is modeled as the parallel composition of handlers W_i. Each handler is a wrapper for a tuple of processes $\widehat{P} = (P_1, \ldots, P_n)$ that correspond to the activities the handler has to run when invoked. Not all handlers are mandatory.
- $W_i(P_{i1}, \cdots, P_{in_i})$ is the process obtained from the multi-hole context $W_i[\cdot]_1 \cdots [\cdot]_{n_i}$ by replacing each occurrence of $[\cdot]_j$ with P_{ij}. It is intended to abstract the WS-BPEL handlers.
- Scope initialization occurs when a process or a scope is entered. It consists of instantiating and initializing the scope's variables and partner links; instantiating the correlation sets; and installing fault, termination and event handlers.
- The case where the variable x is restricted to a simple process P that is not within a scope, is the usual restriction of the π-calculus $(\nu x)P$. In this case, $c\langle \nu n\rangle$ where c, n are nouns will denote a bound output action.

The handlers' syntax is out of the scope of this paper and we refer the reader to [2] for a detailed description of this syntax.

3.3 Operational Semantics

The structural congruence is the smallest equivalence relation closed under the rules in Table 2. The first six rules are standard rules of the π-calculus. All the other rules but the last are about the sequence and scopes and we refer the reader to [2] for detailed comments. The last rule is closely related to the semantics of correlation set update (rule `C-SPF` in Table 3) which guarantees uniqueness of each running instance by a recursive search of the current correlation set to make sure that

$$
\begin{aligned}
P \mid 0 &\equiv P \\
P \mid Q &\equiv Q \mid P \\
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\
(\nu\ \tilde{u})0 &\equiv 0 \\
(\nu\ u)(\nu\ v)P &\equiv (\nu\ v)(\nu\ u)P \\
(\nu\ \tilde{u})P \mid Q &\equiv (\nu\ \tilde{u})(P \mid Q)\ (\forall_i u_i \ \notin fn(Q)) \\
\{\tilde{x}, P, H\} \mid \{\tilde{x}, Q, H'\} &\equiv \{\tilde{x}, Q, H'\} \mid \{\tilde{x}, P, H\} \\
\{\tilde{x}, P, H\} \mid 0 &\equiv \{\tilde{x}, P, H\} \\
\{\tilde{x}, P, H\} \mid (\{\tilde{x}, Q, H'\} \mid \{\tilde{x}, R, H''\}) &\equiv (\{\tilde{x}, P, H\} \mid (\{\tilde{x}, Q, H'\}) \mid \{\tilde{x}, R, H''\} \\
P \ \rhd_{c(M)}\ 0 &\equiv P \\
0 \ \rhd_{c(M)}\ P &\equiv P \\
P \ \rhd_{c(M)}\ (Q \ \rhd_{c(M')}\ R) &\equiv (P \ \rhd_{c(M)}\ Q) \ \rhd_{c(M')}\ R \\
(IG_1 \ + \ IG_2) \rhd_{c(M)}\ P &\equiv IG_1 \ \rhd_{c(M)}\ P \ + \ IG_2 \ \rhd_{c(M)}\ P \\
[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y}) &\equiv [null, \mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y})
\end{aligned}
$$

Table 2
Structural Congruence.

the new instance parameters are consistent by comparing them with the running instances' ones before updating. Also, the last rule ensures that the correlation sets $\mathcal{C}$ and $null, \mathcal{C}$ will be considered as equal along this recursive process.

The operational semantics of the BP-calculus is a labeled transition system generated by inference rules given in Table 3.

The first eleven rules are the standard late semantics' ones in π-calculus without replication. Actually, the construct $[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$ may be viewed as an indexing replication. Semantics of the sequential operator $(\rhd_{c(M)})$ is given by rules `SEQ1`, `SEQ2` and `SEQ3`. Rules `SCO`, `HAN` and `S-PAR` define the behavior of scopes and handlers. These constructs are defined as multihole contexts. Thus, they can be derived from previous rules for handlers are processes. Semantics for message handling is defined by rules `IFT-M` and `IFF-M`. We refer the reader to [2] for detailed comments on these rules. Rule `EVAL` handles function evaluation. The last three rules cope with the correlation's semantics and are presented in the next section.

Correlation's semantics

Rule `C-SP1` allows a spawned service P to carry on in isolation.

Rule `C-SPT` handles the initial spawning of an instance and the initialization of a correlation set after receiving a message containing a request to create an instance. The process $[null : 0]c_A(\tilde{x}).A(\tilde{y})$ indicates that no instance is running and

$$\texttt{RES}\quad \frac{P\xrightarrow{\alpha}P' \;\; n\notin fn(\alpha)\cup bn(\alpha)}{(\nu n)P\xrightarrow{\alpha}(\nu n)P'} \qquad \texttt{OPEN}\quad \frac{P\xrightarrow{\overline{c}(n)}P' \;\; c\neq n}{(\nu n)P\xrightarrow{\overline{c}(\nu n)}(\nu n)P'}$$

$$\texttt{CLOSE}\quad \frac{P\xrightarrow{c(n)}P' \;\; Q\xrightarrow{\overline{c}\langle \nu n\rangle}Q' \;\; n\notin fn(P)}{(\nu n)P|Q\xrightarrow{\tau}(\nu n)P'|Q'} \qquad \texttt{TAU}\quad \frac{}{\tau.P\xrightarrow{\tau}P}$$

$$\texttt{OUT}\quad \frac{}{\overline{c}^t\langle M\rangle.P\xrightarrow{\overline{c}\langle M\rangle}P} \qquad \texttt{IN}\quad \frac{}{c(x).P\xrightarrow{c(M)}P\{M/x\}}$$

$$\texttt{PAR}\quad \frac{P\xrightarrow{\alpha}P' \;\; bn(\alpha)\cap fn(Q)=\emptyset}{P|Q\xrightarrow{\alpha}P'|Q} \qquad \texttt{SYNC}\quad \frac{P\xrightarrow{\alpha}P' \;\; Q\xrightarrow{\overline{\alpha}}Q'}{P|Q\xrightarrow{\tau}P'|Q'}$$

$$\texttt{STRUCT}\quad \frac{P\equiv P' \quad P'\xrightarrow{\alpha}Q' \quad Q\equiv Q'}{P\xrightarrow{\alpha}Q} \qquad \texttt{CHOICE}\quad \frac{P_i\xrightarrow{\alpha}P_i' \;\; i\in\{1,2\}}{P_1+P_2\xrightarrow{\alpha}P_i'}$$

$$\texttt{DEF}\quad \frac{P\{\tilde{y}/\tilde{x}\}\xrightarrow{\alpha}P' \;\; A(\tilde{x})=P}{A(\tilde{x})\xrightarrow{\alpha}P'} \qquad \texttt{SEQ1}\quad \frac{P\xrightarrow{\alpha}P'}{P\rhd_{c(M)}Q\xrightarrow{\alpha}P'\rhd_{c(M)}Q}$$

$$\texttt{SEQ2}\quad \frac{Q\xrightarrow{\alpha}Q' \;\; P\equiv 0}{P\rhd_{c(M)}Q\xrightarrow{\alpha}P\rhd_{c(M)}Q'} \qquad \texttt{SEQ3}\quad \frac{P\xrightarrow{\overline{c}(M)}P' \;\; Q\xrightarrow{c(M)}Q' \;\; P'\equiv 0}{P\rhd_{c(M)}Q\xrightarrow{\tau}P'\rhd_{c(M)}Q'}$$

$$\texttt{SCO}\quad \frac{P\xrightarrow{\alpha}P'}{\{x,P,H\}\xrightarrow{\alpha}\{x,P',H\}} \qquad \texttt{HAN}\quad \frac{H\xrightarrow{\alpha}H'}{\{x,P,H\}\xrightarrow{\alpha}\{x,P,H'\}}$$

$$\texttt{SPAR}\quad \frac{P\xrightarrow{\alpha}P' \quad Q\xrightarrow{\overline{\alpha}}Q'}{\{x,P,H_1\}|\{x,Q,H_2\}\xrightarrow{\tau}\{x,P',H_1\}|\{x,Q',H_2\}}$$

$$\texttt{IFT-M}\quad \frac{P\xrightarrow{\alpha}P' \;\; M=N}{if(M=N)\ then\ P\ else\ Q\xrightarrow{\alpha}P'} \qquad \texttt{IFF-M}\quad \frac{Q\xrightarrow{\alpha}Q' \;\; M\neq N}{if\ (M=N)\ then\ P\ else\ Q\xrightarrow{\alpha}Q'}$$

$$\texttt{EVAL}\quad \frac{\tilde{M}=f(M_1,\ldots,M_n)\;\; P\{\tilde{M}/\tilde{x}\}\xrightarrow{\alpha}P'}{[x\leftarrow f(M_1,\ldots,M_n)]P\xrightarrow{\alpha}P'} \qquad \texttt{C-SP1}\quad \frac{P\xrightarrow{\alpha}P'}{[\mathcal{C}{:}P]c_A(\tilde{x}).A(\tilde{y})\xrightarrow{\alpha}[\mathcal{C}{:}P']c_A(\tilde{x}).A(\tilde{y})}$$

$$\texttt{c-SPT}\quad \frac{createInstance(M)=true \quad [\tilde{z}\leftarrow\tilde{u}]=correlationPart(M)}{[null{:}0]c_A(\tilde{x}).A(\tilde{y})\xrightarrow{c_A(M)}[[\tilde{z}\leftarrow\tilde{u}]{:}A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})}$$

$$\texttt{c-SPF}\quad \frac{createInstance(M)=true \quad [\tilde{z}\leftarrow\tilde{u}]=correlationPart(M)\;\; [\tilde{z}\leftarrow\tilde{u}]\notin\mathcal{C}}{[\mathcal{C}{:}P]c_A(\tilde{x}).A(\tilde{y})\xrightarrow{c_A(M)}[\mathcal{C},[\tilde{z}\leftarrow\tilde{u}]{:}P|A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})}$$

Table 3
Operational semantics of extended BP-calculus.

the correlation set is empty. After creation of the instance $A(\tilde{u})$, the correlation part of the message $[\tilde{z} \leftarrow \tilde{u}]$ is extracted from the correlation part of the request M received over the channel c_A and the correlation set is updated.

Rule C-SPF manages the subsequent instance creations. In this case, before creating a new instance, an inspection of the current correlation set is performed (by mean of the recursive test $[\tilde{z} \leftarrow \tilde{u}] \notin \mathcal{C}$) in order to guarantee that the correlation of the new instance $A(\tilde{u})$ is fresh i.e. is different from all service instances running in parallel and resulting in the process P.

As an example, let $\mathcal{C} = [p \leftarrow null]$ be the correlation set with property p set initially to $null$, and R the process $c_a(p).\overline{c_b}\langle v, p\rangle$. Basically, R receives a message

from c_a carrying the new value of a property p, which is then sent along with v to channel c_b. The initial process is then : $[null : 0]R$
Using rule `C-SPT`, it evolves as :

$$[null : 0]R \overset{c_a(1)}{\rightarrow} [\ [p \leftarrow 1] : \overline{c_b}\,\langle v, p\rangle]R$$

and then by rule `C-SPF`:

$$\overset{c_a(2)}{\rightarrow} [\ [p \leftarrow 1] : \overline{c_b}\,\langle v, p\rangle \ \mid [p \leftarrow 2] : \overline{c_b}\,\langle v, p\rangle]R$$

3.4 Semantics of BPEL constructs coping with correlation

In the following, we use the extended BP-calculus presented in Section 3.2 to redefine the semantics of some basic WS-BPEL constructs to the BP-calculus taking into account correlation mechanisms. The mapping is represented by a function $[\![.]\!]$: BPEL $\longrightarrow$ BP-calculus that maps BPEL constructs to BP-calculus.

- **correlation:** A `<correlation>` element can be used on every messaging activity (`<receive>`, `<reply>`, `<onMessage>`, `<onEvent>`, and `<invoke>`). The common syntax is :

```
<correlations>
    <correlation set="CS" initiate="yes|join|no"? />
</correlations>
```

Initiation Constraint:
"*The* ***initiate*** *attribute on a* ***<correlation>*** *specification is used to indicate whether the correlation set is being initiated or not. When the* ***initiate*** *attribute is set to "yes", the related activity MUST attempt to initiate the correlation set. When the* ***initiate*** *attribute is set to "join", the related activity MUST attempt to initiate the correlation set, if the correlation set is not yet initiated*" [10].
Assuming that the result of function *initiatePart(M)* is the `initiate` attribute of the `<correlation>` element, we can formalize the correlation construct as follows:

$$\begin{aligned}[\![correlate(M)]\!] := if(initiatePart(M) = \text{``}yes\text{''})\ then \\ \overline{initiate}\langle correlationPart(M)\rangle.0 \ \mid\ [x \leftarrow build(fault \leftarrow \text{``}correlationViolation\text{''})].\overline{throw}\,\langle x\rangle\,.0 \\ else\ if(initiatePart(M)\ =\ \text{``}join\text{''})\ then\ \overline{initiate}\langle correlationPart(M)\rangle.0 \\ else \quad [x \leftarrow build(fault \leftarrow \text{``}correlationViolation\text{''})].\overline{throw}\,\langle x\rangle\,.0\end{aligned}$$

$\overline{initiate}\langle correlationPart(M)\rangle$ causes the initiation of the correlation set, i.e the assignment of an initial value to the correlation set $\mathcal{C} = correlationPart(M)$.

In the case where $initiatePart(M)$ = "*yes*" and the initialization fails (i.e $\mathcal{C}$ is already initiated), a "correlationViolation" fault is thrown. On the other hand, if $initiatePart(M)$ = "*join*" , no fault is thrown and a correlation set is initiated if necessary. If $initiatePart(M)$ = "*no*" and the correlation set is not initiated, a "correlationViolation" fault is thrown.

- **Synchronous invoke:** Invoking a service can be both synchronous or asynchronous accordingly with the fact that a reply is expected or not. The invoke activity is defined by the following `<invoke>` construct:

```
<invoke partnerLink=l operation="x"
        inputVariable="i" outputVariable="o" >
    <correlations>
        <correlation set="CS" initiate="yes|join|no"
            pattern="request|response|request-response"? />+
    </correlations>
</invoke>
```

The pattern attribute on the `<correlation>` specification is used to indicate whether the correlation applies to the outbound message, the inbound message, or both.

The `<correlationSets>` applicable to each message must be separately considered, because they can be different. The `<invoke>` construct is formally specified by:

$$\begin{aligned}\llbracket\ invoke(c,M)\ \rrbracket := \llbracket correlate(M)\rrbracket\\ \rhd\ [t \leftarrow build(cs \leftarrow \{CS_i, CS_o\}, initiatePart_i \leftarrow \text{"}yes/no\text{"},\\ initiatePart_o \leftarrow \text{"}yes/no\text{"}, link \leftarrow l, input \leftarrow i, output \leftarrow o)].0\ \rhd\ \overline{c}^{inv}\langle M\rangle.0\end{aligned}$$

Here, there are 2 correlated sets: CS_i for input and CS_o for output. The update function sets the correlation part of the message M corresponding to CS_i and to CS_o to values that identifies the targeted instances, the initiate parts to true or false, the target value to l and the input and output variables to i and o. The message is sent to its target and the correlations are updated.

- **Asynchronous invoke:** This behavior is similar to the synchronous invoke, except that there is no input variable and thus no correlation set for this, for we expect no reply. Its specification is :

$$\begin{aligned}\llbracket\ invoke(c,M)\ \rrbracket := \llbracket correlate(M)\rrbracket\\ \rhd\ [t \leftarrow build(cs \leftarrow CS_o, initiatePart_o \leftarrow \text{"}yes/no\text{"}, link \leftarrow l, output \leftarrow o)].0\\ \rhd\ \overline{c}^{inv}\langle M\rangle.0\end{aligned}$$

- **Receive:** The following `<receive>` construct that waits for a request:

```
<receive partnerLink="l" operation="x" variable="i"
        createInstance="yes|no">
    <correlations>
        <correlation set="CS" initiate="yes|join|no"? />
    </correlations>
</receive>
```

The `<receive >` construct is formally specified by :

$$
\begin{aligned}
[\![\ receive(c_A, A(\tilde{x}), \mathcal{C}, P) \ ]\!] ::= \ & c_A(M) \ \rhd \ if \ (target(M) = self()) \ then \\
& correlate(M) \\
& \rhd \ if \ (createInstance(M) = yes) \ then \\
& [correlationPart(M) : P|A(correlate(M))]c_A(\tilde{y}).A(\tilde{z}) \\
& else \ [\mathcal{C} : P]c_A(\tilde{y}).A(\tilde{z})
\end{aligned}
$$

If the receiving instance (identified by the function *self()*) is the target of the message and createInstance is set to yes, then a new instance of the process P is spawned, leading to a new instance of P running concurrently with existing instances Q_i.

- **Reply:** A `<reply>` must be preceded by a `<receive>` for which it provides a response:

```
<reply partnerLink="l"  operation="x" variable="o" >
    <correlations>
        <correlation set="NCName" initiate="yes|join|no"? />
    </correlations>
</reply>
```

This behavior is specified similarly to the asynchronous invoke activity:

$$
\begin{aligned}
[\![\ reply(c, o) \ ]\!] := \ & [\![correlate(t)]\!] \\
& \rhd \ [t \leftarrow build(cs \leftarrow CS_o, initiatePart_o = \text{"}yes/no\text{"}, link \leftarrow l, output \leftarrow o)].0 \\
& \rhd \ \overline{c}^{rep} \, \langle initiatePart_o(t) \rangle \, .0
\end{aligned}
$$

- **Pick:** This is the nondeterministic execution of one of several activities depending on an external event. Exactly one branch of the construct will be selected according to the occurrence of the related event; other events are no longer accepted by that pick. The syntax is :

```
<pick createInstance="yes|no"? >
    <onMessage partnerLink="l1" operation="x1"
        variable="M1">
    <correlations>
         <correlation set="CS1" initiate="yes|no">
    </correlations>
         activity
    </onMessage>
    <onMessage partnerLink="l2" operation="x2"
            variable="M2">
    </onMessage>
</pick>
```

The `<onMessage>` element behaves like a `<receive>` activity and a new instance of a business process is to be created when `createInstance` attribute is set to yes. The formal specification is as follows:

$$\begin{array}{rl}
[\![\ pick(\{c_1, A_1, M_1, P_1\}, \{c_2, A_2, M_2, P_2\})\ ]\!] := & c_1(M_1)\ \rhd\ if\ (target(M_1) = self())\ then \\
[\![correlate(M_1)]\!] & \rhd\ \ if\ (createInstance(M_1) = yes)\ then \\
[\![[correlationPart(M_1) : P_1]c_1(M_1).A_1(\tilde{z})]\!] & \quad else\ [\![P_1]\!] \\
 & +\ x(M_2) \rhd if\ (target(M_2) = self())\ then \\
[\![correlate(M_2)]\!] & \rhd\ if\ (createInstance(M_2) = yes)\ then \\
[\![[correlationPart(M_2) : P_2]c_2(M_2).A_2(\tilde{z})]\!] & \quad else\ [\![P_2]\!]
\end{array}$$

4 Formal specification of the Trade Market example.

We present in this section a BP-calculus specification of a fragment of the Trade Market example. The system is made of three processes, the agent, the customer and the broker, that run in parallel. The customer process and the broker process are instantiated. The former by the external environment of the whole system, while broker instances are created by the agent, when sending sell orders. The agent acts as a coordinator between other processes and represents the BPEL process we are interested in. For sack of simplicity, we abstract from defining handlers and scopes.

In the following, we present the BPEL code for the agent and the broker processes and then we show how we can automatically generate their corresponding BP-calculus code, using the mapping of Section 3.4. For the customer process we only present their BP-calculus formalization, since it is not directly concerned with instance creation.

4.1 The agent

The agent waits for a request from a customer. Note that this process is not instantiated.

The BPEL code for this fragment is:

```
<! Receiving the request -->
<receive partnerLink="customer" operation="sellRequest"
        createInstance="no"  variable = "CustomerQuantity"/>
    <correlations>
        <correlation set="CustomerCS" initiate="yes" />
    </correlations>
</receive>
```

Then, the agent possibly creates a broker instance and transfers the request to this instance using a synchronous invoke.

```
<! Invoking a broker instance  -->
<invoke partnerLink="broker" operation="orderRequest"
    variable="brokerQuantity">
    <correlations>
        <correlation set="BrokerCS" initiate="no" />
    </correlations>
</invoke>
```

Then the agent waits for a response from the broker.

```
<! Receiving a response from the broker  -->
<receive partnerLink="broker" operation="orderNotice"
        variable="amountSold">
    <correlations>
       <correlation set="BrokerCS" initiate="no" />
    </correlations>
</receive>
```

Finally, he sends the received response to the requesting customer.

```
<! Replying to the customer  -->
<assign>
    <copy>
        sellNotice.count:= amountSold
    </copy>
```

```
</assign>
<reply  partnerLink="customer" operation="sellNotice"
        variable="amountSold">
    <correlations>
         <correlation set="CustomerCS" initiate="no" />
    </correlations>
</reply>
<assign>
    <copy>
        sharesSold+=sellNotice.count
    </copy>
</assign>
```

For a more detailed BPEL code for this process that takes into account the multiplicity of broker's instances, see Appendix A.

Now, we give the formalization into BP-calculus of the agent process.
In this case, since customer's instances are not created by the agent, the create message's attribute is set to false (within the process $C()$ of the agent definition). The channel y is used to signal to the external environment that the sale is done.

$$\begin{aligned}
Agent(x_c, y_c, x_b, y_b) = {} & y_c(M) \;\triangleright\; update(M', (brokerID, createInstance \leftarrow \text{"}true\text{"}, dest \leftarrow \text{"}broker\text{"},\\
& \quad data \leftarrow extractData(M'))) \;\triangleright\; \overline{x_b}^{inv}\langle M' \rangle\\
& \mid y_b(M) \;\triangleright\; update(M\text{"}, (orderID, customerID, createInstance \leftarrow \text{"}false\text{"},\\
& \quad dest \leftarrow \text{"}customer\text{"}, data \leftarrow extractData(M))).\overline{x_c}^{rep}\langle M\text{"} \rangle\\
& \quad \triangleright\; Agent(x_c, y_c, x_b, y_b)
\end{aligned}$$

4.2 The customer

After updating a message with an order ID and a quantity of shares to be sold, the customer sends a request containing this information to the agent and waits for a response:

$$\begin{aligned}
Customer(x_c, y_c, z) = {} & update(M, (orderID, customerID, dest \leftarrow \text{"}agent\text{"}, data \leftarrow qty)).\overline{y_c}\langle M \rangle\\
& \mid\; y_c(M) \;\triangleright\; if\ (target(M) = self())\ then\ update(M_{ok}, (ok)) \;\triangleright\; \overline{z}^{rep}\langle M_{ok} \rangle\\
& \quad else\ 0
\end{aligned}$$

4.3 The broker

The broker waits for a request from the agent. He identifies itself as the targeted instance by comparing the target ID with its own one. Finally, he replies by sending an updated message containing its ID and the amount of shares sold.

Since this process is concerned with the correlation mechanism, we provide a fragment of its BPEL code, and we map this code to the corresponding BP-calculus code.

```
<receive partnerLink="agent" operation="OrderRequest"
        createInstance="yes" variable = "brokerQuantity ">
    <correlations>
        <correlation set="BrokerCS" />
     </correlations>
</receive>
...
<assign>
    <copy>
        sellNotice.count:= brokerSold
    </copy>
</assign>
<reply partnerLink="agent" operation="OrderNotice"
        variable = "amountSold">
    <correlations>
        <correlation set="BrokerCS" initiate="yes" />
    </correlations>
</reply>
```

A new instance of the broker process is created by receiving a new order from the agent.

The reply message contains the correlation information that must be presented in subsequent requests that are targeted at this process instance. Each invocation from the agent therefore refers to the same correlation set that is initiated when the reply is sent.

$$Broker(x_b, y_b) = x_b(M) \ \triangleright \ if\ create(M)\ then\ [[p \leftarrow brokerID] : D']D(M, y_b)$$
$$else\ if\ target(M) = self()\ then\ D(M, y_b)\ else\ 0.$$
$$D(M, y_b) = \tau.update(M', (brokerID, createInstance \leftarrow \text{"}false\text{"},$$
$$dest \leftarrow \text{"}agent\text{"}, data \leftarrow amountsold))\overline{y_b}^{rep}\left\langle M' \right\rangle$$

4.4 The Trade Market process

Putting all these constructs together, we can modelize the whole system as the parallel composition of these processes:

$$TradeMarket(y) = (\nu\ x_c, y_c, x_b, y_b)(Customer(x_c, y_c, z) \mid Agent(x_c, y_c, x_b, y_b) \mid Broker(x_b, y_b))$$

Channels x_c, y_c, x_b and y_b represent the channels by which the Agent communicates with its environment, eg, the customers and the brokers. Channel z is used by the customer to signal the correct termition of the transaction.

It is worth to note that, due to rules `c-SPT` and `c-SPF`, the semantics guarantees that correlation sets are unique i.e. they are not initialized twice.

5 Conclusion

In this paper we enhanced the BP-calculus [2] by using function calls and message handling to formalize the correlation concept for orchestration languages in general and the BPEL language in particular. Some aspects of this work need a deeper analysis in order to study some properties of the model and the interactions of correlation sets with handlers and other complex constructs. This work may also integrate a formalization of the WS-Addressing [7] standard in order to deal with name mobility. Finally, we are working to integrate all the concepts presented in this paper to the verification engine based on the BP-calculus that we are developing.

Acknowledgement

The authors would like to acknowledge Sardaouna Hamadou for his contributions to the paper and the anonymous referees for their useful comments.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Symposium on Principles of Programming Languages*, page 104115. ACM Press, 2001.

[2] F. Abouzaid and J. Mullins. A calculus for generation, verification and refinement of bpel specifications. *Electronic Notes in Theoretical Computer Science*, 200(3):43–65, 2008.

[3] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. Sock : A calculus for service oriented computing. In Springer Berlin / Heidelberg, editor, *Service-Oriented Computing ICSOC 2006*, volume 4294/2006 of *Lecture Notes in Computer Science*, pages 327–338, 2006.

[4] Lanese I., Vasconcelos V., Martins F., and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *5th IEEE International Conference on Software Engineering and Formal Methods*, pages 305–314. IEEE, 2007.

[5] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[6] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming, Elsevier press*, 2007.

[7] Gudgin
M., Hadley M., and Rogers T. Web services addressing 1.0 - core. http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/, 2006.

[8] R. Milner. *Communication and Concurrency. Series in Computer Science.* Prentice Hall, 1989.

[9] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus.* Cambridge University Press, Cambridge, UK, 1999.

[10] Oasis. Web service business process execution language version 2.0 specification, oasis standard,. $http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf$, april 2007.

[11] M.H. ter Beek, A. Bucchiarone, and S. Gnesi. Formal methods for service composition. In *In Proceedings of the 3rd South-East European Workshop on Formal Methods (SEEFM'07)*, pages 365 – 78, 2007.

[12] M. Viroli. A core calculus for correlation in orchestration languages. *Journal of Logic and Algebraic Programming*, 70(1):74–95, January 2007.

Appendix

A BPEL schema of the Agent service.

WSDL definitions of messages:

```
<wsdl:message name="CustomerRequest">
    <wsdl:part name="CustomerID" type="CustomerIDType" />
    <wsdl:part name="OrderID" type="OrderIDType" />
    <wsdl:part name="OrderQuantity" type="OrderQuantityType" />
</wsdl:message>

<wsdl:message name="OrderToBroker">
    <wsdl:part name="BrokerID" type="BrokerIDType" />
    <wsdl:part name="BrokerQuantity" type="BrokerQuantityType" />
</wsdl:message>

<wsdl:message name="BrokerResponse">
    <wsdl:part name="BrokerID" type="BrokerIDType" />
    <wsdl:part name="AmountSold" type="AmountSoldType" />
</wsdl:message>

<wsdl:message name="AgentNotice">
    <wsdl:part name="CustomerID" type="CustomerIDType" />
    <wsdl:part name="OrderID" type="OrderIDType" />
    <wsdl:part name="TotalAmountSold" type="TotalAmountSoldType" />
</wsdl:message>
```

Properties and correlation sets

```
<propertyAlias propertyName="CustomerID"
        messageType="SellRequest" part="CustomerID"/>

<propertyAlias propertyName="CustomerID"
        messageType="SellRequest" part="OrderNumber" />

<propertyAlias propertyName="BrokerID"
        messageType="BrokerResponse" part="AccountNumber"/>

<correlationSets>
    <correlationSet name="CustomerCS" properties="CustomerID OrderID" />
    <correlationSet name="BrokerCS" properties="BrokerID" />
</correlationSets>
```

BPEL code for agent service:

```
<sequence>
    <! Receiving the request -->
    <receive partnerLink="customer" operation="sellRequest"
           createInstance="no"  variable = "CustomerQuantity"/>
       <correlations>
           <correlation set="CustomerCS" initiate="yes" />
       </correlations>
    </receive>

    <assign>
        <copy>
           sharesSold:=0
        </copy>
    </assign>
    <assign>
    <copy>
     brokersNumber:=N
    </copy>
    </assign>
    <forEach parallel="yes">
        <startCounterValue>1</startCounterValue>
        <finalCounterValue> N </finalCounterValue>
        <scope>
            <variables>...</variables>
            <partnerLinks>...</partnerLinks>
            <!-- Primary activity  -->
            <sequence>
                <! Invoking a broker instance  -->
                <invoke partnerLink="broker" operation="orderRequest"
        variable="brokerQuantity">
                    <correlations>
                        <correlation set="BrokerCS" initiate="no" />
                    </correlations>
      </invoke>
                <! Receiving a response from the broker  -->
        <receive partnerLink="broker" operation="orderNotice"
         variable="amountSold">
                    <correlations>
                        <correlation set="BrokerCS" initiate="no" />
                    </correlations>
                </receive>
                <assign>
                   <copy>
                      sellNotice.count:= brokerSold
                   </copy>
                </assign>
        <invoke partnerLink="customer" operation="sellNotice"
         variable="AgentNotice">
                   <correlations>
                      <correlation set="CustomerCS" initiate="no" />
                   </correlations>
                </invoke>
        <assign>
                   <copy>
                      sharesSold+=sellNotice.count
                   </copy>
                </assign>
            </sequence>
        </scope>
    </forEach>
</sequence>
```

BPEL code for the Broker:

```
<receive partnerLink="agent" operation="OrderRequest"
       createInstance="yes" variable = "brokerQuantity ">
    <correlations>
        <correlation set="BrokerCS" />
    </correlations>
</receive>
...
<! Replying to the customer  -->
<assign>
   <copy>
      sellNotice.count:= amountSold
   </copy>
</assign>
<reply  partnerLink="customer" operation="sellNotice"
        variable="amountSold">
    <correlations>
        <correlation set="CustomerCS" initiate="no" />
    </correlations>
</reply>
<assign>
    <copy>
        sharesSold+=sellNotice.count
    </copy>
</assign>
```