Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Logic and Algebraic Programming 70 (2007) 96–118
THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING
www.elsevier.com/locate/jlap

A pi-calculus based semantics for WS-BPEL

Roberto Lucchi, Manuel Mazzara*

Dipartimento di Scienze dell'Informazione, Università degli Studi di Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy

Abstract

Recently, the term Web services orchestration has been introduced to address some issues related to Web services composition, that is the way of defining a complex service out of simpler ones. Several proposals for describing orchestration for business processes have been presented in the last years and many of these languages make use of concepts as long-running transactions and compensations for coping with error handling. WS-BPEL 2.0, the most credited candidate for becoming a standard, provides three different mechanisms allowing to cope with abnormal situations: exception, event and compensation handling. This complexity makes it difficult to formally define the framework, thus limiting the formal reasoning about the designed applications. In this paper we advocate that three different mechanisms for error handling are not necessary and we formalize a novel orchestration language based on the idea of event notification as the unique error handling mechanism. To this end, we formally define the three BPEL mechanisms in terms of our calculus. It is possible to take advantages of this formal description in two ways. Firstly, this language represents by itself a proposal of simplification for WS-BPEL 2.0 including an unambiguous specification. Secondly, an implementor of an actual WS-BPEL 2.0 orchestration engine could implement simply this single mechanism providing all the remaining ones by compilation. With this attempt we intend to give a concrete contribute towards the improvement of the quality of the BPEL specification, the applicability of BPEL itself and the implementation of real orchestration engines. Finally, as a case study we consider some of the hundreds of open issues met by the WS-BPEL designers and we propose a solution making use of the experience gained developing our algebra.

Keywords: Concurrency theory; Business process; Web services; Composition

1. Introduction

Service Oriented Computing (SOC) [17] is an emerging paradigm for distributed computing and e-business processing that finds its origin in object-oriented [7] and component computing [42]. One of the main goals of SOC is enabling developers in building networks of integrated and collaborative applications, regardless of both the platform where the application or service runs (e.g., the operating system) and the programming language used to develop them.

Web services is a set of technologies supporting SOC. It provides a platform on which we can develop applications taking advantage of the Internet infrastructure. A Web service, specifically, supports its operations by associating any functionalities to specific access points available over the network in such a way that they can be exploited, in turn, by other services. The W3C [45] official definition of Web service is the following:

* Corresponding author. Tel.: +39 051 2094974; fax: +39 051 2094510.
E-mail addresses: lucchi@cs.unibo.it (R. Lucchi), mazzara@cs.unibo.it (M. Mazzara).

1567-8326/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2006.05.007

> "A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols."

The Web services framework is essentially based on three technologies: SOAP, WSDL and UDDI. SOAP [39] (Simple Object Access Protocol) is a protocol defining how services have to interact with each other. It is based on XML [49] (eXtensible Markup Language) and describes the documents structure to be used for invoking services. By basing the specification on XML makes it possible to overcome the problems arising when integrating different operating systems, object models and programming languages. In order to achieve flexibility, SOAP abstracts away from the underlying transmission protocol (typically HTTP, SMTP, FTP, ...). WSDL [12] (Web Services Description Language), instead, describes the operations supplied by services, including expected parameters and return values. Finally, since it is necessary to have a central market place where publishing WSDL documents to allow other parties to find and use them, UDDI (Universal Description, Discovery and Integration) [43] has been introduced.

1.1. Web services orchestration

Web services technologies provide the way to build complex services out of simpler ones. Then, the composition can be, in turn, exposed as a service in the same way of basic components. In order to support services composition the so called orchestration languages have been proposed.

A *business process orchestration* consists of the aggregation of Web services by programming business rules or patterns governing services interactions, and has the ability to reuse the created aggregations [20]. Business rules can be seen as the ingredient that sequences, coordinates, and manages interactions among Web services. To program a complex cross-enterprise workflow task or business transaction, for example, it is possible to logically chain discrete Web service activities into inter-enterprise business processes.

Different organizations are presently working on additional layers which have to deal with workflow based composition of Web services. The most important proposals that have been presented in the last years are: IBM's WSFL [23] and Microsoft's XLANG [3,41]. A more recent proposal, which aims at integrating both WSFL and XLANG, is Web Services Business Process Execution Language [1] (WS-BPEL or BPEL for short). It is a workflow-based programming language that describes sophisticated business processes that orchestrate Web services. It allows for a mixture of block and graph-structured process models, thus making the language expressive at the price of being complex. So far, BPEL represents the most credited candidate to become a future standard in the field of Web services orchestration. For this reason it deserves to be investigated and considered as a touchstone for any further effort in this field.

An orchestration language must support activities for both communicating with other Web services and handling workflow semantics. One can think of a basic activity as a component that interacts with something external to the process itself. In contrast, structured activities manage the overall process flow by specifying the order in which composing activities have to run. Business process orchestration requirements – as presented in literature [34] – involve asynchronous interactions, flow coordination, business transaction activity and management. More in detail, orchestration languages for Web services should exhibit the following behaviours:

- Sequence
- Parallel
- Conditional
- Send/Receive to/from other WS on typed ports (WSDL)
- Invocation of WS
- Error handling

In this paper we will investigate with particular attention the BPEL recovery framework, relatively to the last point above. BPEL provides three different mechanisms to deal with abnormal situations: exception, event and compensation handling.[1] Fault handler and compensation handler are thought to be used at different stages of computation: the former during the execution of an activity while the latter when it has been completed. While fault (and sometimes event)

[1] It is worth to note that the BPEL event handling mechanism was not specifically designed for error handling. Although it is not a goal of this work, we consider that the proposed language still allows for all the remaining usages of the original mechanism.

handlers, are provided by typical concurrent programming languages, compensation handlers can be considered a peculiarity of orchestration languages. Compensations are related with long running web transactions.

Web transactions are transactional interactions between organizations [24]. They are long running, therefore classic ACID properties of transactions are not enough, since they make sense only when trusted parties are involved over short periods of time and they are not appropriate in a loosely coupled world of autonomous trading partners where hard locking of local resources is unfeasible. So, since business applications require transactional support beyond the classical approach, BPEL introduces the concepts of long running transaction and compensation as most existing orchestration languages do when describing loosely coupled activities. Compensations are actions which attempt to "reverse" the effects of previous actions; they are carried within a transaction when it needs to be cancelled. The meaning of reversing effects depends on each application. For ACID transactions in a DBMS, the transaction coordinator and the resource controlled by it know at any time all the uncommitted updates and have full control over the order in which they must be reversed. In the case of business transactions, however, the compensation behavior is itself a part of the business logic and must be explicitly specified.

1.2. Contribution of the paper

BPEL is not equipped with formal semantics (as other existing proposals) and, since it includes a large number of aspects, it is difficult to formally reasoning on processes behavior.

In light of this observation, the semantics of a BPEL fragment is formally addressed in this paper. In particular, we focus our efforts on the specification of event, fault and compensation handlers behavior. Besides, we advocate that three different mechanisms for error handling are not necessary and we formalize a novel orchestration language, $\texttt{web}\pi_\infty$, which is based on the idea of event notification as the unique error handling mechanism. Our belief is supported also by different works recently proposed by other researchers who are using similar mechanisms (see for example the extensions of the CORBA transactional system reported in [15]).

$\texttt{web}\pi_\infty$ is obtained by extending the π-calculus with a transactional construct composed by two processes, say P and Q, and identified by a unambiguous event name, say x. During the normal execution of P, the specific event x can occur, raised by a parallel thread. If this happens P is terminated and Q starts.

We show how BPEL mechanisms for error and event handling can be programmed atop our calculus. Moreover, in order to show the adequacy of $\texttt{web}\pi_\infty$ as orchestration language the main activities of BPEL are encoded into $\texttt{web}\pi_\infty$. As a consequence, we indirectly provide a formal semantics also for basic and structured activities of BPEL.

We consider that such a proposal represents a significant contribution in two directions. Firstly, $\texttt{web}\pi_\infty$ represents by itself a simplification of WS-BPEL 2.0 including an unambiguous specification, thus making possible to formally reason on orchestration processes. Secondly, an implementor of an actual WS-BPEL 2.0 orchestration engine could implement simply this single mechanism providing all the remaining ones by compilation.

Finally, a couple of the most interesting open issues met by BPEL designers have been taken into consideration. In particular, we show how a solution for these issues can be programmed in $\texttt{web}\pi_\infty$, thus proving that there exists a expressiveness gap between the two languages. This result highlights the suitability of our proposal for orchestration scenarios.

1.3. Related work

In this paper we mainly refer to BPEL, the most likely candidate to become a standard. Other languages have been presented, among them WS-CDL [19], another workflow-based composition language which claims to be in some relation with fusions [35] and solos [21]. In particular, WS-CDL is built atop the Global Model formalism by Nickolas Kavantzas (as presented in [18]) which is based on the Explicit Solos Calculus [13], the theory underlying the Fusion Machine, a virtual machine implementing in a distributed manner the π-calculus.

Other papers discussing the formal semantics of compensable activities in this context are: the work by Hoare [16] which is mainly inspired by XLANG, the calculus of Butler and Ferreira [10] which is inspired by BPBeans [11], the πt-calculus [6] considering BizTalk and the work [9] dealing with short-lived transactions in BizTalk. The work in [8] also presents the formal semantics for a hierarchy of transactional calculi with increasing expressiveness.

In this paper we only consider the fragment of BPEL concerning concurrency, synchronization, events, faults and compensation handling. Besides computational constructs, another aspect which deserves to be investigated are

correlation sets. Since Web services allow us simple forms of interactions (one-way or request–response) such mechanism provides a mean for correlating several interactions. Such an orthogonal issue has been separately investigated in [44].

Concluding, some authors believe that time should be introduced both at the model level and at the protocols and implementation levels [22,3–5,26]. The specification of XLANG, for instance, provides a notion of timed transaction as a special case of long running activity. Also BPEL can exhibit a similar behaviours using timers. This is a very appropriate feature when programming business services which cannot wait forever for other parties reply.

1.4. Outline of the paper

The paper is structured as follows. Section 2 presents the BPEL fragment on which we focus our attention. Section 3 is firstly devoted to describe the main requirements an orchestration language should provide and motivates the choice of starting by the π-calculus. Then, our language $\texttt{web}\pi_\infty$ is presented with the relative syntax and semantics. Section 4 describes how main BPEL activities can be encoded in $\texttt{web}\pi_\infty$, while Section 5 describes the behavior of event, fault and compensation handlers and shows how these mechanisms can be programmed. Section 6 presents some open issues in BPEL design and how they can be solved in our calculus. Finally, Section 7 reports some conclusive considerations.

2. WS-BPEL 2.0

BPEL is, in practice, a layer on top of WSDL. The information contained in WSDL documents defines message types and port types which represent the operations supported by the service and the interaction modalities. This information is then used by BPEL for specifying the flow of actions to perform. A BPEL document is an XML-based document that can be executed by an orchestration engine which is the central coordinator. The engine will read the BPEL document and will invoke the necessary Web services in the specified order. The process itself will be offered as a Web service and can be invoked in the same way.

The structure of a BPEL process is described by the following fragment of XML-Schema [50] (attributes are omitted since they do not change the meaning of the parts we are interested in):

```
<element name="process" type="bpws:tProcess"/> <complexType
name="tProcess"> <complexContent>
  <extension base="bpws:tExtensibleElements">
  <sequence>
     <element name="import" type="bpws:tImport" minOccurs="0" maxOccurs="unbounded"/>
     <element name="partnerLinks" type="bpws:tPartnerLinks" minOccurs="0"/>
     <element name="partners" type="bpws:tPartners" minOccurs="0"/>
     <element name="variables" type="bpws:tVariables" minOccurs="0"/>
     <element name="correlationSets" type="bpws:tCorrelationSets" minOccurs="0"/>
     <element name="faultHandlers" type="bpws:tFaultHandlers" minOccurs="0"/>
     <element name="compensationHandler" type="bpws:tCompensationHandler" minOccurs="0"/>
     <element name="terminationHandler" type="bpws:tTerminationHandler" minOccurs="0"/>
     <element name="eventHandlers" type="bpws:tEventHandlers" minOccurs="0"/>
     <group ref="bpws:activity"/>
  </sequence>
</complexType>
```

In a few words, a BPEL process is composed of: (i) the list of partners involved in the process and their role, (ii) the mechanisms to manage events (`eventHandlers`) and abnormal situations (`faultHandlers` and `compensationHandler`), (iii) the business process, that is the activities that must be performed to accomplish the service.

The main goal of this paper is to achieve a deeper understanding of error handling mechanisms. In the following we will describe only the elements relevant to get the goal (activities, scopes and the recovery framework). Activities, in essence, provide a mean for message passing (basic), concurrency and synchronization (structured). The XML-Schema fragment describing the activity elements follows:

```
<group name="activity">
    <choice>
        <element name="empty" type="bpws:tEmpty"/>
        <element name="invoke" type="bpws:tInvoke"/>
        <element name="receive" type="bpws:tReceive"/>
        <element name="reply" type="bpws:tReply"/>
        <element name="assign" type="bpws:tAssign"/>
        <element name="wait" type="bpws:tWait"/>
        <element name="throw" type="bpws:tThrow"/>
        <element name="rethrow" type="bpws:tRethrow"/>
        <element name="terminateexit" type="bpws:tTerminate"/>
        <element name="flow" type="bpws:tFlow"/>
        <element name="switch" type="bpws:tSwitch"/>
        <element name="while" type="bpws:tWhile"/>
        <element name="sequence" type="bpws:tSequence"/>
        <element name="pick" type="bpws:tPick"/>
        <element name="scope" type="bpws:tScope"/>
        <element name="compensate" type="bpws:tCompensate"/>
    </choice>
</group>
```

The following sections are devoted to describe the main activities (divided in basic and structured), the event, fault and compensation handling mechanisms. For the sake of simplicity, we abstract away from some syntactical details. Moreover, to make more intelligible the syntax, elements will be expressed by using a BNF (Backus–Naur Form) formalisms instead of XML-Schema (in the same style of the BPEL specification). For completeness, we just recall the meaning of frequency operators: $el?$, $el+$, $el*$ mean respectively zero or one, one or more, zero or more occurrences of the element el.

2.1. Basic activities

Basic BPEL activities are described in the following sections.

2.1.1. Invoke

Invoke an operation of a Web service. The invocation can be both synchronous and asynchronous accordingly with the interaction modality used by the invoked service. The `invoke` activity is defined as follows:

```
<invoke partnerLink="ncname" portType="qname" operation="ncname"
        inputVariable="ncname"? outputVariable="ncname"?
        standard-attributes>
</invoke>
```

In a few words, `partnerLink`, `portType` and `operation` are used to indicate the specific operation supplied by a certain partner, the access point of the invoked operation and the transmission protocol used to transmit SOAP requests (e.g., SMTP, HTTP, ...) `inputVariable` and `outputVariable` represent instead the variables passed as input to and received as output from the service (if a response is expected), respectively.

2.1.2. Receive

Wait for a request. A receive activity has to specify the interacting partner in terms of its role in the process, port type and supplied operation. The semantics of a process where more than one receive associated with the same partner, portType and operation are simultaneously enabled is undefined. The syntax for `receive` is as follows:

```
<receive partnerLink="ncname" portType="qname" operation="ncname"
         variable="ncname"? createInstance="yes|no"?
         standard-attributes>
  standard-elements
</receive>
```

2.1.3. Reply

It is used to generate a response. A meaningful constraint is that the reply activity must always be preceded by a receive activity for the same partner, port type and operation.

```
<reply partnerLink="ncname" portType="qname" operation="ncname"
       variable="ncname"? faultName="qname"?
       standard-attributes>
  standard-elements
</reply>
```

2.1.4. Throw

It can be used to explicitly signal an internal fault. Any fault is required to have a globally unique name. When the throw is performed, the name has to be specified as well as some variables containing information about the faults. This is done by the following syntax:

```
<throw faultName="qname" faultVariable="ncname"?
standard-attributes>
  standard-elements
</throw>
```

Faults are caught by fault handlers. We will describe fault handlers in Section 2.3.2.

2.1.5. Compensate

The `compensate` activity is used to invoke the compensation handler. The syntax is

```
<compensate scope="ncname"? standard-attributes>
  standard-elements
</compensate>
```

Since compensation handlers are associated to scopes, the relative attribute is used to indicate the target of the activity. It is worth noting that compensation handlers can be invoked only by fault handlers or by compensation handlers associated with outer scopes. If no compensation handler is defined, the default handler compensates all the children scopes.

2.1.6. Empty

It represents a terminated activity.

```
<empty standard-attributes>
  standard-elements
</empty>
```

2.2. Structured activities

Structured activities describe how a business process is created by composing basic activities into complex structures expressing workflow, control patterns, dataflow, faults handling, external events management and coordination of messages exchange between process instances involved in a business protocol.

Main structured activities of BPEL include: (i) ordinary sequential or branching composition (`sequence` and `switch`), (ii) parallel composition and synchronization (`flow`), and (iii) nondeterministic choice (`pick`).

The description of these activities follows.

2.2.1. Sequence

It allows for sequential composition of activities. A sequential activity contains one or more activities that are performed in the order in which they are listed within the `sequence` element. The sequence activity completes when the final activity in the sequence has completed.

```
<sequence standard-attributes>
  standard-elementsactivity+
</sequence>
```

2.2.2. Switch

Case-statement approach. The activity consists of an ordered list of one or more conditional branches defined by `case` elements (followed by an optional `otherwise` branch). The `case` branches are considered in the order in which they appear. The first branch whose condition holds (condition are expressed by boolean expressions) defines the activity to be performed by the switch. If no branch condition holds then the `otherwise` branch is performed if it exists, otherwise the `switch` immediately terminates. The whole activity completes when the selected activity completes.

```
<switch standard-attributes>
  standard-elements
  <case condition="bool-expr">+
    activity
  </case>
  <otherwise>?
    activity
  </otherwise>
</switch>
```

2.2.3. Pick

It is used to perform the nondeterministic execution of one of several paths depending on an external event. The form of a `pick` is a set of branches of the form event-activity where exactly one of these branches will be selected. A branch is selected if the event associated with it occurs. After the pick activity has accepted an event, the other events are no longer accepted by that pick. The possible events are the arrival of some message or an alarm clock based on a timer. Each pick activity must include at least one `onMessage` event. Remember that the semantics of a process in which two ore more receive actions for the same partner, porttype and operation are simultaneously enabled is undefined. The same thing holds in the case of the pick statement. The activity waits the occurrence of one of the defined events and performs the associated activity. If more than one of the events occurs then the selection depends on which one occurred first. If the events occurs almost simultaneously, there is a timing and implementation-dependent race. The activity completes when one of the branches is triggered by the occurrence of its associated event and the corresponding activity completes. The syntax of `pick` element follows:

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partnerLink="ncname" portType="qname"
             operation="ncname" variable="ncname"?>+
    <correlations>?
```

```
      <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
    activity
  </onMessage>
</pick>
```

2.2.4. Flow

Parallel execution of primitive activities. The `flow` construct provides concurrency and synchronization. The most fundamental semantic effect of grouping a set of activities in a flow is to enable concurrency. A `flow` completes when all of the activities it contains have completed. The simplest use of this activity is equivalent to a nested concurrency construct. The `flow` syntax is defined as it follows:

```
<flow standard-attributes>
  standard-elements
  <links>?
    <link name="ncname">+
  </links>
  activity+
</flow>
```

2.2.5. Links

This feature – an inheritance left by WSFL – allows to set up certain constraints on the elements inside a `flow`. This constraint (called link) allows to specify some order on the execution of the parallel activities. Using links a `flow` construct can be viewed as a graph (this was the basic paradigm of WSFL). Nodes are activities and edges are the links expressing the interdependencies among activities, that is the order with which they have to be executed.

Links are used to express synchronization dependencies between activities. Abstracting from the XML-based syntax, a link with its name represents a connection between two activities: one defined as the `source` and one defined as the `target`. Both the source and the target must define explicitly their role in the syntax. The source activity may also specify a transition condition (if the condition is not defined it is intended to be true). For simplicity, we shall not describe crossing-boundary link (i.e. links where the source is inside the flow construct while the target is not, or viceversa. For this kind of link some restrictions holds: for example a link cannot cross a `scope`).

2.3. Scope

Scope provides a behavior context for each activity inside it. Any scope has a primary structured activity which defines its normal behavior and the scope is shared by all the nested activities. Besides activities, scopes might also contain fault, compensation and event handlers. Variables definition inside a scope holds until it is *active*. A scope becomes active when its activities can be executed and terminates when all the activities it contains are completed. It is important to point out that the two mechanisms for dealing with abnormal situations, the fault and the compensation handler, are in essence concerning with different stages of the computation. In particular, fault handler is the mechanism used during the execution of the scope activities, while the compensation is used after the successful termination of the scope. The definition of the `scope` element follows:

```
<scope variableAccessSerializable="yes|no" standard-attributes>
  standard-elements
  <variables>?
    ...
  </variables>
  <correlationSets>?
    ...
```

```
  </correlationSets>
  <faultHandlers>?
    ...
  </faultHandlers>
  <compensationHandler>?
    ...
  </compensationHandler>
  <eventHandlers>?
    ...
  </eventHandlers>

  activity
</scope>
```

Any scope can include fault, compensation and event handlers whose description follows.

2.3.1. Compensation handler

If defined the *compensation handler* contains the activity to be performed in the case we are willing to compensate the activity of the scope. The compensation handler is installed (i.e. it is allowed to be performed) when the scope terminates in a successful way or, in other words, if no fault has been thrown during the execution and all the activities have been performed. It is allowed to consult the variables state as they appear at its termination and to interact with other services. Compensation handlers can be provided by the following syntax:

```
<compensationHandler>?
  activity
</compensationHandler>
```

2.3.2. Fault handler

The `catch` element allows handling a fault specified by a fault name. `catchAll`, instead, is able to capture any fault not specifically handled. Faults are signalled by the `throw` activity which interrupts the normal execution of the scope activating the relative fault handler, if defined. In the opposite case, a default handler is executed.

The default handler compensates all the children scopes and the pending fault is then rethrown to the parent scope (by specification, the whole BPEL process is considered a scope). By definition, the scope managing the signalled fault has not a normal termination and consequently its compensation handler will not be installed. To define a fault handler the following syntax is required:

```
<faultHandlers>?
  <!-- there must be at least one fault handler or default -->
  <catch faultName="qname"? faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

2.3.3. Event handler

Any scope, as well as the whole business process, can be associated with *event handlers*. The life cycle of these handlers is the same of the associated scope (either it terminates successfully or faulty). Any handler is relative to a

particular event (an incoming message or a timeout[2]) and defines the activities to be performed if this event occurs. Moreover, messages that are caught by an event handler are consumed and – in the same way as happens for receive activities – it is not allowed that a message sent from a certain partnerlink, portType and operation can be simultaneously consumed by more than one receive or event handler. Activities performed by an event handler can be anything except `compensate`, that is the invocation of a compensation handler. When an event occurs, the corresponding handler is concurrently executed with the main activity. Other event will also be handled concurrently, even in the case of two identical occurrences.

```
<eventHandlers>?
  <!-- there must be at least one onMessage or
       onAlarm handler -->
  <onMessage partnerLink="ncname" portType="qname"
             operation="ncname"
             variable="ncname"?>*

    <correlations>?
        <correlation set="ncname" initiate="yes|no">+
    </correlations>
    activity
  </onMessage>
  <onAlarm for="duration-expr"? until="deadline-expr"?>*
    activity
  </onAlarm>
</eventHandlers>
```

3. The orchestration calculus $\texttt{web}\pi_\infty$

Although BPEL covers all the aspects presented in the previous section, its current specification is rather involved. A major issue is error handling.

As explained, BPEL provides three different mechanisms which allow us to cope with abnormal situations: fault, compensation and event handling. Documentation shows ambiguities, in particular about the interdependencies among these mechanisms. Therefore, it is difficult to use the language, and we want to clarify this aspect.

Our goal is the definition of a clear model with the smallest set of operators which can meet the behaviours introduced above, offering a reasonable simplicity to the application designers. We start from the π-calculus [32,30,37], a well known process algebra which has been widely studied during the last 15 years. It is simple and appropriate for orchestration purposes. It includes:

- a parallel operator allowing explicit concurrency,
- a restriction operator allowing compositionality and explicit resource creation,
- a recursion or a process definition operator,
- a sequence operator allowing causal relationship between activities,
- an inaction operator which is just a ground term for inductive definition on sequencing,
- message passing and especially name passing operators allowing communication and link mobility.

The main reason to use the π-calculus for formalization is because the so called *Web services composition languages*, like XLANG, WS-BPEL and WS-CDL, claim to be based on it, and they would therefore allow rigorous mathematical treatment. However, no interesting relation with process algebras has really been proved for any of these languages, nor an effective tool for analysis and reasoning, either theoretical or software based, has been released. Therefore, we see a gap that needs to be filled, and we want to address the problem of composing services starting directly from the π-calculus. This is not to say that the π-calculus is superior to other models, such as Petri nets [36].

[2] In this paper we do not deal with temporal aspects, thus this case is not considered.

Although the π-calculus seems to offer an adequate framework for orchestration, by itself it does not support any transactional mechanism. Programming complex business processes with failure handling in terms of message passing only is not reasonable; also, the Web services environment requires that several operations have transactional properties and be treated as a single logical unit of work when performed within a single business transaction. Therefore, below we shall consider a lightweight and conservative extension of the π-calculus that covers transactions. The language we present is a slight extension of the one presented in [27] to analyze a case study of Web services orchestration.

3.1. Syntax

The syntax of $\texttt{web}\pi_\infty$ *processes* relies on countable sets of *names*, ranged over by $x, y, z, u, \ldots$ Tuples of names are written $\widetilde{u}$. We intend $i \in I$ with I a finite nonempty set of indexes.

$P ::=$	
$\mathbf{0}$	(**nil**)
$\mid P \mid P$	(**parallel composition**)
$\mid \sum_i x_i(\widetilde{u_i}).P_i$	(**alternative composition**)
$\mid \texttt{if } (x = y) \texttt{ then } P \texttt{ else } Q$	(**conditional**)
$\mid \overline{x}\,\langle \widetilde{u} \rangle$	(**output**)
$\mid (x)P$	(**restriction**)
$\mid !x(\widetilde{u}).P$	(**lazy replication**)
$\mid \langle\!\vert P \ ; \ P \vert\!\rangle_x$	(**transaction**)

A process can be the inert process $\mathbf{0}$, a parallel composition of processes, an alternative composition consisting of input guarded processes that consumes a message $\overline{x_i}\,\langle \widetilde{w}_i \rangle$ and behaves like $P_i\{\widetilde{w_i}/\widetilde{u_i}\}$, a conditional branch based on name equality, an output $\overline{x}\,\langle \widetilde{u} \rangle$ sent on a name x that carries a tuple of names $\widetilde{u}$, a restriction $(x)P$ that behaves as P except that inputs and messages on x are prohibited, a replicated input $!x(\widetilde{u}).P$ that consumes a message $\overline{x}\,\langle \widetilde{w} \rangle$ and behaves like $P\{\widetilde{w}/\widetilde{u}\} \mid !x(\widetilde{u}).P$, or a (Web) transaction $\langle\!\vert P \ ; \ R \vert\!\rangle_x$ that behaves as the *body* P until a transaction abort $\overline{x}\,\langle\rangle$ is signalled, in this case when it occurs it behaves as the *compensation* R.

Names x in outputs, inputs, and replicated inputs are called *subjects* of outputs, inputs, and replicated inputs, respectively. It is worth to notice that the syntax of $\texttt{web}\pi_\infty$ processes simply augments the asynchronous π-calculus with transaction process.

The input $x(\widetilde{u}).P$, restriction $(x)P$ and replicated input $!x(\widetilde{u}).P$ are binders of names $\widetilde{u}$, x and $\widetilde{u}$, respectively. The scope of these binders is the process P. We use the standard notions of α-equivalence, *free* and *bound names* of processes, noted $\texttt{fn}(P)$, $\texttt{bn}(P)$ respectively. In particular $\texttt{fn}(\langle\!\vert P \ ; \ R \vert\!\rangle_x) = \texttt{fn}(P) \cup \texttt{fn}(R) \cup \{x\}$ and α-equivalence equates $(x)(\langle\!\vert P \ ; \ Q \vert\!\rangle_x)$ with $(z)(\langle\!\vert P\{z/x\} \ ; \ Q\{z/x\} \vert\!\rangle_z)$. We also define $bn_i(P)$ as the set of bound names of P with inputs and replicated inputs as binders.

In this paper we consider only processes in $Process$ which is the set composed of well-formed $\texttt{web}\pi_\infty$ processes, whose definition follows.

Definition 3.1 (*Well-formedness*). A process is well-formed if the two following conditions hold:

(1) (*Output capability of input names*) Received names cannot be used as subjects of inputs or of replicated inputs. More formally if a process, say P, contains $u(\widetilde{v})$ or $!u(\widetilde{v})$ then $u \notin bn_i(P)$.

(2) (*Unicast activation*) Transaction names are distinct. Different transactions cannot share the same activation name and every abort message is able to activate a single compensation. Formally, if a process is of the form $P_1 \mid \langle\!\vert P \ ; \ Q \vert\!\rangle_x \mid P_2 \mid \langle\!\vert R \ ; \ S \vert\!\rangle_y \mid P_3$ for some P_1, P_2 and P_3 then $x \neq y$.

The first of these properties guarantees that if a name is received by a certain process it does not use that name to perform input on it. Such a condition is strictly related with the fact that the access points of Web services, that is the operations, are available on specific channels. In this way we avoid a situation where different services support the same operation. The second is intended to avoid ambiguity on scope names reflecting BPEL itself, that is no any transaction is univocally identified.

3.2. The semantics

Now we shall give the semantics for the language in two steps, following the approach of Milner [31]. This approach consists in separating the laws which govern the static relations between processes from the laws which rule their interactions. We shall achieve this by defining firstly a static *structural congruence* relation over processes. A structural congruence relation equates all processes we will never want to distinguish and it is introduced via a collection of axioms that allow minor manipulation on the processes structure. This relation is intended to express some intrinsic meanings of the operators, for example that parallel composition is commutative. Secondly, we shall define the way in which processes evolve dynamically by means of an operational semantics. In this way we simplify the statement of the semantics just closing with respect to $\equiv$, i.e. closing up to structural congruence.

Definition 3.2. The *structural congruence* $\equiv$ is the least congruence satisfying the abelian monoid laws for parallel (associativity, commutativity and $\mathbf{0}$ as identity) closed with respect to α-renaming and the following axioms:

1. Scope laws:

$$(u)\mathbf{0} \equiv \mathbf{0}, \quad (u)(v)P \equiv (v)(u)P,$$
$$P \mid (u)Q \equiv (u)(P \mid Q), \quad \text{if } u \notin \mathtt{fn}(P)$$
$$\langle\!|(z)P \;;\; Q|\!\rangle_x \equiv (z)\langle\!|P \;;\; Q|\!\rangle_x, \quad \text{if } z \notin \{x\} \cup \mathtt{fn}(Q)$$

2. Transaction laws:

$$\langle\!|\mathbf{0} \;;\; Q|\!\rangle_x \equiv \mathbf{0}$$
$$\langle\!|\langle\!|P \;;\; Q|\!\rangle_y \mid R \;;\; R'|\!\rangle_x \equiv \langle\!|P \;;\; Q|\!\rangle_y \mid \langle\!|R \;;\; R'|\!\rangle_x$$

3. Floating law:

$$\langle\!|\overline{z}\,\langle\widetilde{u}\rangle \mid P \;;\; Q|\!\rangle_x \equiv \overline{z}\,\langle\widetilde{u}\rangle \mid \langle\!|P \;;\; Q|\!\rangle_x$$

The scope laws are standard while novelties regard transaction and floating laws. The law $\langle\!|\mathbf{0} \;;\; Q|\!\rangle_x \equiv \mathbf{0}$ defines committed transactions, namely transactions with $\mathbf{0}$ as body. These transactions, being committed, are equivalent to $\mathbf{0}$ and, therefore, cannot fail anymore. The law $\langle\!|\langle\!|P \;;\; Q|\!\rangle_y \mid R \;;\; R'|\!\rangle_x \equiv \langle\!|P \;;\; Q|\!\rangle_y \mid \langle\!|R \;;\; R'|\!\rangle_x$ moves transactions outside parent transactions, thus flattening nested transactions. Notwithstanding this flattening, parent transactions may still affect children transactions by means of transaction names. The law $\langle\!|\overline{z}\,\langle\widetilde{u}\rangle \mid P \;;\; R|\!\rangle_x \equiv \overline{z}\,\langle\widetilde{u}\rangle \mid \langle\!|P \;;\; R|\!\rangle_x$ floats messages outside transactions, thus modelling the fact that messages are particles that independently move towards their inputs. The intended semantics is the following: if a process emits a message, this message traverses the surrounding transaction boundaries until it reaches the corresponding input. In case an outer transaction fails, recoveries for this message may be detailed inside the compensation processes.

The dynamic behavior of processes is defined by the reduction relation.

Definition 3.3. The *reduction relation* $\rightarrow$ is the least relation satisfying the following axioms and rules, and closed with respect to $\equiv$, $(x)\_$, $\_ \mid \_$, and $\langle\!| \_ \;;\; Q|\!\rangle_x$:

$$\overline{x_i}\,\langle\widetilde{v}\rangle \mid \textstyle\sum_i x_i(\widetilde{u_i}).P_i \;\rightarrow\; P_i\{\widetilde{v}/\widetilde{u_i}\} \quad \text{(COM)}$$
$$\overline{x}\,\langle\widetilde{v}\rangle \mid !x(\widetilde{u}).P \;\rightarrow\; P\{\widetilde{v}/\widetilde{u}\} \mid !x(\widetilde{u}).P \quad \text{(REP)}$$
$$\overline{x}\,\langle\rangle \mid \langle\!|P \;;\; Q|\!\rangle_x \;\rightarrow\; Q \qquad P \not\equiv \mathbf{0} \quad \text{(FAIL)}$$
$$\mathtt{if}\ (x = x)\ \mathtt{then}\ P\ \mathtt{else}\ Q \rightarrow P \quad \text{(IFT)}$$
$$\mathtt{if}\ (x = y)\ \mathtt{then}\ P\ \mathtt{else}\ Q \rightarrow Q \qquad x \neq y \quad \text{(IFF)}$$

Rules (com) and (rep) are standard in process calculi and models input–output interaction and lazy replication. Rule fail models transaction failures: when a transaction abort (a message on a transaction name) is emitted, the corresponding body is terminated and the compensation activated. On the contrary, aborts are not possible if the transaction is already

terminated, namely every thread in the body has completed its own work. It is worth noting that we consider terminated a transaction when its main activity is equal to **0**, thus it is not in a terminated state if the main activity is a process P which behaves as **0**, in the sense that it is semantically equivalent to **0**. For instance $(x)x(\widetilde{u})$ behaves as **0** but it is not considered terminated thus the transaction can react to the fail signal. Finally (ift) and (iff) reflect the intended behavior for conditional branching.

4. Expressing WS-BPEL 2.0 semantics

In this section our goal is to give the semantics of WS-BPEL 2.0: we will do this in terms of $\mathtt{web}\pi_\infty$. This effort is intended to show the flexibility of the algebra with respect to orchestration purposes when compared with real languages. Practically, we limit our work to a subset of the whole BPEL. We consider mainly basic activities, structured activities and error handling, letting Correlation Sets, Partner and data handling as future work. Preliminary results appeared can be found in [28], while an application of those results in [14].

A consistent feeling regarding significant relations between π-calculus and orchestration languages is spreading, both in the academia and in the industry [38]. Although all this hype, to formally compare such languages is not trivial.

BPEL for example cannot completely deal with name mobility if not coupled with another specification, WS-Addressing [47], which is just a W3C member submission. Nevertheless, another meaningful difference is that WS-BPEL combines computational and concurrent constructs while π-calculus is essentially focused on concurrency.

In particular, BPEL allows both global name and global states over computation and message passing. Furthermore it combines sequencing with concurrency allowing synchronization patterns. In these cases interleaving and name binding create confusing behavior with respect the natural π-calculus semantics.

Let us consider the following example where we use the symbol ; to express the sequential composition of BPEL:

$$((x(u).P \mid Q);\ R \mid S);\ T$$

Normally, the π-calculus semantics allows the binding of u simply over P according to the rule

$$\overline{x}\,\langle\widetilde{v}\rangle \mid x(\widetilde{u}).P \quad \rightarrow \quad P\{\widetilde{v}/\widetilde{u}\}$$

and the closure over parallel composition.

Differently, the sequencing of BPEL is not a prefix operator like this and would allow the substitution over all the scope where names are defined including R, S, T and even the remaining part of Q at the moment of the receiving. This is a typical global state semantics of sequential programming languages that, when combined with concurrent flow makes very intricate to model such behavior with the π-calculus.

Furthermore, BPEL has not a formally defined semantics and this significantly complicates the understanding and the attempt of comparing its behavior with the one of the π-calculus. All these points, in our opinion, should convince us to be very careful when associating process algebra and workflow languages (particulary BPEL which has been the main touchstone of our investigation).

Anyway, a mapping is possible if it is limited to concurrency and synchronization mechanisms of BPEL, paying attention to these few complication which will be adequately remarked also in the following.

4.1. Core BPEL syntax

Now let us give the formal machinery necessary to express the encoding. Firstly we present a syntax for the core language we intend to consider in the following. As said, we are abstracting from several aspects of the language focusing on workflow and failure handling. Furthermore we want to abstract form verbose XML syntax.

The syntax of BPEL *processes* relies on countable sets of *port names*, ranged over by $x, y, z, u, i, o, \ldots$, *fault names*, ranged over by f, g, h and *scope names*, ranged over by $z, z', \ldots$ Tuples of names are written $\widetilde{u}$.

$$
\begin{array}{lll}
A ::= & & \\
 & \texttt{empty} & \textbf{(empty)} \\
| & \texttt{invoke}(x_s, \tilde{i}, \tilde{o}) & \textbf{(synch invoke)} \\
| & \texttt{invoke}(x_s, \tilde{i}) & \textbf{(asynch invoke)} \\
| & \texttt{receive}(x_s, \tilde{i}) & \textbf{(receive)} \\
| & \texttt{reply}(x_s, \tilde{o}) & \textbf{(reply)} \\
| & \texttt{throw}(f, \tilde{o}) & \textbf{(throw)} \\
| & \texttt{compensate}(z, \tilde{o}) & \textbf{(compensate)} \\
| & \texttt{sequence}\ (A, A) & \textbf{(sequence)} \\
| & \texttt{flow}\ (A, A) & \textbf{(parallel)} \\
| & \texttt{switch}\ (x = y)\ A; A & \textbf{(conditional)} \\
| & \texttt{pick}\ ((x, \tilde{i_1}, A), (x, \tilde{i_2}, A)) & \textbf{(alternative)} \\
| & \texttt{scope}_z(A, S_e, S_f, A) & \textbf{(scope)}
\end{array}
$$

This syntax is quite intuitive and it simply represents an abstraction for the actual BPEL syntax. Sequence, parallel, conditional and alternative are here composed by only two activities. Clearly, the composition can be extended to an arbitrary number of activities by nesting multiple statement.

The scope construct deserves some considerations. Here S_e (resp. S_f) is a finite set of triples of the form $(x, \widetilde{u}, A)$ (resp. $(f, \widetilde{u}, A)$). Informally, $(x, \widetilde{u}, A)$ (resp. $(f, \widetilde{u}, A)$) means that the event related to port name x (resp. the exception f) is handled by the activity A which receives as parameters $\widetilde{u}$.

The term $\texttt{scope}_z(A, S_e, S_f, A_c)$ defines a scope, whose name is z, where A represents the process associated with the scope, S_e the set of handled events, S_f the set of handled exceptions and, finally, A_c the compensation activity associated with that scope.

In the following we consider only BPEL activities in A_{BPEL} which is the set composed of well-formed activities. The definition of well-formedness for activities is essentially the same used for webπ processes and can be obtained simply by rephrasing syntactical constraints defined in Definition 3.1 (e.g., scope names unequivocally identify scopes).

4.2. Semantics

In this section we give the semantics in terms of our algebra. To this end we define the function $[\![\ ]\!]_{\overline{y}\,\langle\tilde{u}\rangle} : A_{BPEL} \rightarrow Process$ which maps BPEL activities into webπ_∞ processes. Such processes execute the BPEL activity and flag out $\overline{y}\,\langle\tilde{u}\rangle$ to signal termination. The role played by $\tilde{u}$ will be clarified later. Shortly, the tuple is used as a way to pass received names in sequenced activities in the continuation passing style encoding (see Section 4.4.1). It is worth noting that this expedient does not completely solve the problem of variables state alignment (e.g., in parallel composed processes the problem still holds) but it supports value passing in the most significant cases (sequential composition). Considering shared global variables is beyond the scope of this paper, anyway it is well-known in the literature that global accesses can be avoided in favour of message passing and viceversa. Message passing can be used to implement shared memory: a global state can be a particular process which receives read and write messages and operates the required tasks; and message passing can be done using shared variables as communication channels [40].

In the following we compositionally define this function. We assume the names ranged over $y, y', \ldots$ as fresh names.

4.3. Basic activities

Let us consider basic activities.

4.3.1. Empty

The empty behavior <empty/> has a natural mapping in **0**. Although such an activity do nothing, as will be clear in the following we have to signal it has been enabled (i.e. terminated) to support sequential composition. We express the encoding as

$$[\![\texttt{empty}]\!]_{\overline{y}\,\langle\tilde{u}\rangle} = \overline{y}\,\langle\tilde{u}\rangle$$

4.3.2. Asynchronous (one-way) invoke

BPEL allows asynchronous invocation of services available over the network. Differently from the complete syntax (see Section 2.1.1) here we are abstracting from some details of the invoked service by using a channel as service identifier (i.e. identifying a specific operation of a certain service). In particular, any operation implemented by a Web service can be considered associated to a channel which is dedicated to receive invocations of the supplied operation. We translate this behavior as follows:

$$[\![\texttt{invoke}(x_s, \tilde{i})]\!]_{\overline{y}\,\langle\tilde{u}\rangle} = \overline{x_s}\,\langle\tilde{i}\rangle \mid \overline{y}\,\langle\tilde{u}\rangle$$

where x_s is the access point of the service. It is worth noting that, since we are considering the asynchronous invoke, the encoding does not guarantee that when the termination is signalled the invocation has completed.

4.3.3. Synchronous (request–response) invoke

Let us consider synchronous invocation of services: This kind of invocation needs two variables, we identify as o the result of invocation. Here we need to introduce a *session name* r, which is a fresh name, to correlate the invocation with its response (in particular, as we will show in the following, the invoked service S uses this channel to reply):

$$[\![\texttt{invoke}(x_s, \tilde{i}, \tilde{o})]\!]_{\overline{y}\,\langle\tilde{u}\rangle} = r(\overline{x_s}\,\langle r, \tilde{i}\rangle \mid r(\tilde{o}).\overline{y}\,\langle\tilde{u}\rangle)$$

4.3.4. Supplying services

Accordingly with `invoke` activity defined above, x_s represents a specific operation supplied by a service. In $\texttt{web}\pi_\infty$ the same specification looks like:

$$[\![\texttt{receive}(x_s, \tilde{i})]\!]_{\overline{y}\,\langle\tilde{u}\rangle} = x_s(r, \tilde{i}).\overline{y}\,\langle\tilde{u}\rangle$$

where the name x_s has to be publicly available (i.e. a global name) and r is the channel name to be used to perform the `reply` activity if it is expected (in this case the `invoke` have to be programmed in a synchronous way accordingly with Section 4.3.3 where r is a fresh name passed by the invoker).

4.3.5. Supplying response

Services with request–response interaction modality use `reply` activity to transmit the result of invocation. We specify this behavior similarly to the `invoke` activity:

$$[\![\texttt{reply}(x_s, \tilde{o})]\!]_{\overline{y}\,\langle\tilde{u}\rangle} = \overline{x_s}\,\langle\tilde{o}\rangle \mid \overline{y}\,\langle\tilde{u}\rangle$$

4.3.6. Raising an exception

Raising an exception in our model consists of an output message on a failure channel that activates the handler responsible for catching it, and by another output that signals to the scope that it has to be terminated (scopes and fault handlers will be programmed in the following section). The $\texttt{throw}(f, \tilde{o})$ is programmed as follows:

$$[\![\texttt{throw}(f, \tilde{o})]\!]_{\overline{y}\,\langle\tilde{u}\rangle} = r(\,\overline{throw}\,\langle\rangle \mid \overline{f}\,\langle r, \tilde{o}\rangle \mid r().\overline{y}\,\langle\tilde{u}\rangle)$$

where f represents the exception name while $\tilde{o}$ the value passed to the handler. The fault signal is composed of two outputs: one is caught by the fault handler ($\overline{f}\,\langle r, \tilde{o}\rangle$) which performs an output on r when it completes its activity, and the other one ($\overline{throw}\,\langle\rangle$) by the scope that terminates its activity. The $\texttt{throw}(f, \tilde{o})$ completes when the fault has been handled.

4.3.7. Compensate

Compensation mechanism is programmed simply by performing an output on a channel, whose name is the fault name, with parameters that have to be passed to the compensation handler as it follows:

$$[\![\texttt{compensate}(z, \tilde{o})]\!]_{\overline{y}\,\langle\tilde{u}\rangle} = \overline{z}\,\langle\tilde{o}\rangle \mid \overline{y}\,\langle\tilde{u}\rangle$$

4.4. Structured activities

Let us consider structured activities.

4.4.1. Sequential composition

Let `sequence` (A, A') be the sequential composition of activity A and A'; the encoding of such an activity into $\texttt{web}\pi_\infty$ follows.

$$[\![\texttt{sequence}\ (A', A'')]\!]_{\overline{y}\langle\tilde{u}\rangle} = y'([\![A']\!]_{\overline{y'}\langle\tilde{v}\rangle} \mid y'(\tilde{u}).[\![A'']\!]_{\overline{y}\langle\tilde{u}\rangle}) \quad \text{where } \tilde{v} = \texttt{fn}(A'')$$

At this point, there is a subtle point which deserves more attention. As previously mentioned, the sequencing of BPEL is not a binder operator like the π-calculus. In particular, variables scopes infer different semantical behavior with respect to usual binding of process algebra. The problem arises when sequentially combining two activities which include some name input on common names. In these cases the encoding requires more attention. For this reason we cannot simply trigger the second activity when the first one terminates but we require to pass additional information to bind the free names of the second activity. In this paper we are claiming that any statement trying to assimilate workflow languages like BPEL and π-like languages should be carefully analyzed and need a deeper investigation where theoretical considerations play a central role.

4.4.2. Parallel composition

A crucial aspect of workflow languages for Web services is the opportunity of composing in parallel many activities. Concurrency allows the versatility needed by the scenario but introduces some semantical complications of primary importance.

BPEL allows the usage of links in flow activity. Links are used to express synchronization dependencies between activities. This feature of BPEL is a legacy of WSFL and, in our opinion, represents another source of redundancy and confusion for a workflow language. We intend to code this simply by message passing. Here, to keep the specification small we ignore this aspect.

We express the semantics of this construct as

$$[\![\texttt{flow}\ (A', A'')]\!]_{\overline{y}\langle\tilde{u}\rangle} = y'y''([\![A']\!]_{\overline{y'}\langle\tilde{u'}\rangle} \mid [\![A'']\!]_{\overline{y''}\langle\tilde{u''}\rangle} \mid y'(\tilde{u'}).y''(\tilde{u''}).\overline{y}\langle\tilde{u}\rangle)$$

4.4.3. Conditional composition

Conditional composition in BPEL follows a case-statement approach. We present the basic case of two alternatives with a boolean condition on name equalities which are naturally encoded by the conditional statement of our algebra. Let A be the conditional composition of A' and A'', the specification in $\texttt{web}\pi_\infty$ of A is the following:

$$[\![\texttt{switch}\ (x = x')\ A'; A'']\!]_{\overline{y}\langle\tilde{u}\rangle} = \texttt{if}\ (x = x')\ \texttt{then}\ [\![A']\!]_{\overline{y}\langle\tilde{u}\rangle}\ \texttt{else}\ [\![A'']\!]_{\overline{y}\langle\tilde{u}\rangle}$$

4.4.4. Alternative composition

Abstracting over service details (we simply write x_i for a specific service) the alternative composition can be obtained by the pick statement. Pick allows nondeterministic execution of one of several paths based on an external event, this is the typical choice operator of the π-calculus. Let A be the alternative composition of A' and A'', the specification in $\texttt{web}\pi_\infty$ of A is the following:

$$[\![\texttt{pick}\ ((x_1, \tilde{i_1}, A), (x_2, \tilde{i_2}, A))]\!]_{\overline{y}\langle\tilde{u}\rangle} = x_1(\tilde{i_1}).[\![A']\!]_{\overline{y}\langle\tilde{u}\rangle} + x_2(\tilde{i_2}).[\![A'']\!]_{\overline{y}\langle\tilde{u}\rangle}$$

5. Scope

Scopes can be thought as the wrapping structure containing BPEL processes. Activities inside scopes can be associated to particular mechanisms to handle asynchronous events and failures. In this section our aim is to investigate scopes semantics. To this end, we firstly describe our understanding of the scope behavior, then we describe how it can be programmed in $\texttt{web}\pi_\infty$.

Let $A_z = scope_z(A, S_e, S_f, C)$ be a scope. When nothing bad occurs (i.e. no fault happens), it behaves as A (the main activity). As mentioned in Section 2.3, event and fault handlers both play their role during the execution of this activity: they are enabled when A is performed and they are disabled when it terminates. The fundamental difference is that events are handled concurrently with A while faults interrupt its execution.

In the case an event $\overline{x}\langle\widetilde{v}\rangle$ occurs during the normal execution of a main activity A, if an appropriate handler $(x, \widetilde{u}, B) \in S_e$ is defined for some $\widetilde{u}$ and B, the event is consumed and the scope behaves as the parallel composition

(i.e. flow) of A_s and $B\{\widetilde{v}/\widetilde{u}\}$. Here $\widetilde{u}$ represents the formal parameters of the handler. On the other hand, if during the execution of a scope an exception $\texttt{throw}(f, \widetilde{v})$ happens (note that exceptions can be raised only by the main activity) two different behaviours are possible, depending whether the current fault is handled or not. In the first case, that is when $(f, \widetilde{u}, B) \in S_f$ for some $\widetilde{u}$ and B, the scope behaves as the fault handler $B\{\widetilde{v}/\widetilde{u}\}$ where $\widetilde{u}$ are the names which represent the formal parameter. In the opposite case, the scope behaves as the parallel composition of $\texttt{throw}(f, \widetilde{v})$ (the exception is rethrown to the outer scope) and the default handler compensating all the children scopes. Compensation handlers are installed when the activity A terminates without faults. In this case, when $\texttt{compensate}(z, \widetilde{o})$ is performed, the scope A_z behaves as $C\{\widetilde{o}/\widetilde{u}\}$ where $\widetilde{u}$ are names substituted by $\widetilde{o}$. On the other hand, if $\texttt{compensate}(z, \widetilde{o})$ is performed when A has not completed, the scope behaves as the default handler compensating all the children scopes.

5.1. Encoding scopes in $\texttt{web}\pi_\infty$

All the described behaviours are allowed in the event based framework $\texttt{web}\pi_\infty$. In this section we shall show how the mechanisms provided by the BPEL Recovery Framework can be programmed in our language. At the end, it should be immediate to figure out that implementing failure resilient business processes within this single error handling mechanism is both easy and effective. We intend this as a main contribution of this paper.

Following the approach used in the whole paper, we present how to specify the scope activity by means of $\texttt{web}\pi_\infty$ processes. The specification we propose is essentially composed by many threads running in parallel, some managing scope activities, other dedicated to event, fault and compensation handlers. We firstly describe these last three processes.

Let h_f and h_e be functions that, given a scope, return respectively the set of handled faults and handled events. More precisely, $h_f(\texttt{scope}_z(A, S_e, S_f, C)) = \{x \mid (x, \widetilde{u}, B) \in S_f\}$ and $h_e(\texttt{scope}_z(A, S_e, S_f, C)) = \{x \mid (x, \widetilde{u}, B) \in S_e\}$ (when this is not ambiguous we will use shortcuts $h_f(S_f)$ and $h_e(S_e)$). We also define s_n as a function that, given $A \in A_{BPEL}$, returns the set of scope names contained in A. Consider that all these functions are simply syntactical extractor. The following subsections are devoted to describe the three error handling mechanisms, and the concluding subsection reports the encoding of scopes.

5.1.1. Event handler

Let S_e be the set of handled events and y_{eh} a name used to signal the event handler has been disabled, the process implementing the event handling behavior is as follows:

$$EH(S_e, y_{eh}) = (y')(\{e_x \mid x \in h_e(S_e)\})$$

$$en_{eh}().\left(\left(\!\left|\prod_{(x,\widetilde{u},A)\in S_e} !\, x(\widetilde{u}).\overline{e_x}\,\langle\widetilde{u}\rangle \;;\; \overline{y_{eh}}\,\langle\rangle\right|\!\right)_{dis_{eh}} \Bigg| \prod_{(x,\widetilde{u},A_x)\in S_e} !\, e_x(\widetilde{u}).[\![A_x]\!]_{\overline{y'}\,\langle\rangle}\right)$$

The definition consists of two parallel processes: the first catching events while the second handling them. The signal on channel en_{eh} enables the overall process to became reactive to all the events included in S_e. The transactions is introduced to disable event handling: when the signal dis_{eh} is raised, no events are longer considered and the termination signal y_{eh} is performed. It is worth to note that actual event handling activities A_x are outside the transaction. This ensures that, once disabled the whole handler, the activities which are still alive can complete their execution as underlined by the BPEL specification. Event handlers are composed in parallel and are specified by replicated input guarded processes waiting for an incoming message on an event related channel. This allows concurrent management of multiple events. The process executed when a specific event occurs are identified by $[\![A_x]\!]_{\overline{y'}\,\langle\rangle}$, i.e. the translation of the BPEL body activity in $\texttt{web}\pi_\infty$ by means of the function introduced in the previous section. Finally, some consideration on restricted names. Besides y', which is used to signal the termination of any handler (we do not distinguish among different handlers because we are not interested in when they complete), we restrict names e_x for any $x \in h_e(S_e)$ (that we assume not appearing free in A_x for any $x \in h_e(S_e)$) in order to guarantee that the defined handlers can be activated only within EH.

5.1.2. Fault handler

Let S_f be the set of handled faults and y_{fh} a name used to signal fault handler termination, the process implementing the fault handling behavior is as follows:

$$FH(S_f, y_{fh}) = (y')$$
$$en_{fh}().(\!| \sum_{(f,\widetilde{u},A)\in S_f} f(r,\widetilde{u}).(\llbracket A \rrbracket_{\overline{y'}\langle\rangle} \mid \overline{throw}\,\langle\rangle) \mid y'().(\overline{r}\,\langle\rangle \mid \overline{y_{fh}}\,\langle\rangle)\ ;\ \overline{y_{fh}}\,\langle\rangle |\!)_{dis_{fh}}$$

The encoding of such a construct is rather similar to the event handling except for the fact that, when a fault is thrown, the main activity of the scope must be terminated. After the fault handling, the entire scope has to be terminated as well. Thus, fault handlers are encoded by alternative composition of input guarded processes waiting for a message on a relative channel. Any process associated to a fault is programmed in such a way that, when it terminates, the whole FH also terminates. When the fault handling completes, both the including scope and the activating $\texttt{throw}(f, \tilde{u})$ are notified. The first by an output on y_{fh}, the second by an output on r. As a consequence of this notification they will be terminated. Note that, differently from event handlers, when a fault is caught, the process behaves as the mapping of the associated activity in parallel with an output on $throw$. This second parallel activity will be received by the scope which will be terminated.

5.1.3. Compensation handler

The compensation handler has to be programmed in a different way. Indeed it has to be enabled when the scope starts and installed at the end of the execution if no fault happened. Moreover, it can react to a compensate invocation at any time behaving in a way depending on the state of scope execution. Let A and C be scope and compensation activities, z a scope name and y_{ch} a name used to signal handler installation. The process which manages scope compensation is defined as follows:

$$CH(A, C, z, y_{ch}) =$$
$$en_{ch}().(\!|z(\widetilde{u}).(CC(A,\widetilde{u}) \mid \overline{throw}\,\langle\rangle)\ ;\ (y')\, z(\widetilde{u}).\llbracket C \rrbracket_{\overline{y'}\langle\rangle} \mid \overline{y_{ch}}\,\langle\rangle |\!)_{inst_{ch}}$$

where $CC(A, \tilde{u})$ is an auxiliary process used to compensate children scopes of A:

$$CC(A, \tilde{u}) = \prod_{z'\in s_n(A)} \overline{z'}\,\langle\tilde{u}\rangle$$

Accordingly with Section 4.3.7, the compensation handler associated to a scope z has to react to an output on channel z. In the case the handler has been enabled but not yet installed, the behavior is $z(\tilde{u}).(CC(A, \tilde{u}) \mid \overline{throw}\,\langle\rangle)$. Otherwise the handler behaves as $(y')z(\widetilde{u}).\llbracket C \rrbracket_{y'} \mid \overline{y_{ch}}\,\langle\rangle$ where $\overline{y_{ch}}\,\langle\rangle$ signals the scope that the compensation has been properly installed. The first behavior means the termination of the scope activity ($\overline{throw}\,\langle\rangle$) and the compensation of children scopes ($CC(A, \tilde{u})$). The second behavior, instead (the compensation handler has been installed), consists of the compensation process defined by designers.

5.1.4. Scopes

Now we are ready to put together all this machinery to specify the whole scope construct. In the following, in order to make more intelligible the specification, we use as shortcut $\overline{en_{x,y}}\,\langle\rangle$ to denote $\overline{en_x}\,\langle\rangle \mid \overline{en_y}\,\langle\rangle$; similarly for $\overline{dis_{x,y}}\,\langle\rangle$. Let $A_z = \texttt{scope}_z(A, S_e, S_f, C) \in A_{BPEL}$ be a scope and y' a name not occurring free in A; A_z semantics is defined as follows:

$$\llbracket \texttt{scope}_z(A, S_e, S_f, C) \rrbracket_{\overline{y}\langle\rangle} =$$
$$y' y_{eh} y_{fh} y_{ch}(h_f(A_z))(throw)(en_{eh}, en_{fh}, en_{ch}, dis_{eh}, dis_{fh}, inst_{ch})$$
$$EH(S_e, y_{eh}) \mid$$
$$FH(S_f, y_{fh}) \mid$$
$$CH(A, C, z, y_{ch}) \mid$$
$$(\!|\overline{en_{eh,fh,ch}}\,\langle\rangle \mid \llbracket A \rrbracket_{\overline{y'}\langle\rangle} \mid y'().\overline{t}\,\langle\rangle \mid c().(\overline{dis_{eh,fh}}\,\langle\rangle \mid \overline{inst_{ch}}\,\langle\rangle) \mid y_{eh}().y_{fh}().y_{ch}().\overline{y}\,\langle\rangle$$
$$\mid (x_z().(\overline{throw}\,\langle\rangle \mid \overline{dis_{fh}}\,\langle\rangle) + t().\overline{c}\,\langle\rangle)$$
$$\overline{dis_{eh}}\,\langle\rangle \mid StopExecC(A) \mid x_s().\overline{dis_{fh}}\,\langle\rangle \mid y_{eh}().y_{fh}().y'().\overline{y}\,\langle\rangle |\!)_{throw}$$

where $StopExecC(A) = \prod_{s\in ScNames(A)} \overline{x_s}\,\langle\rangle$.

Firstly some considerations about restricted names. By restricting the name $throw$ we allow only activities inside the scope itself to rise an exception. By restricting all handled fault, we instead guarantee that these faults are caught by the handler of this scope: they are propagated only in the case the fault is not handled by the current FH.

The program is the parallel composition of the event, fault and compensation handlers and by a process representing the main activity of the scope. Such a process is a transaction whose meaning is the following: (i) the process executed represents the activity of the scope, (ii) the process associated to the compensation of the transaction is programmed for handling faults.

We proceed first by describing the encoding of the scope activity. It starts by enabling the event, fault and compensation handlers, and by performing the process obtained by encoding the activity programmed in the scope in parallel with other processes whose meaning is

- $(x_z().(\overline{throw}\ \langle\rangle \mid \overline{dis_{fh}}\ \langle\rangle) + t().\bar{c}\ \langle\rangle)$: its meaning is managing the stop of scope activity which parent scopes can invoke in order to stop the execution of children scopes ($\overline{x_z}\ \langle\rangle$). The alternative composition of such functionality with the input guarded process on t guarantees that such an activity is available until $[\![A]\!]_{\overline{y'}\langle\tilde{u}\rangle}$ terminates.
- $y'().\bar{t}\ \langle\rangle \mid c().(\overline{dis_{eh,fh}}\ \langle\rangle \mid \overline{inst_{ch}}\ \langle\rangle)$: it moves when the scope activity terminates, it disable stop handling and then event and fault as well and installs the compensation handlers.
- $y_{eh}().y_{fh}().y_{ch}().\overline{y}\ \langle\rangle$: such process is responsible of signalling the scope termination (in the case the scope completes in a successful way), that is event and fault handlers have been disabled and compensation handlers have been installed.

Finally, in the case of successful termination the process of the transaction terminates thus the entire transaction as well. During the execution of the scope activity events are handled by $EH(S_e, y_{eh})$, faults by $FH(S_f, y_{fh})$ and the compensation by $CH(A, C, z, y_{ch})$.

In the case a fault is thrown during the execution of the scope activity, an output on channel $throw$ (and another one on f which is the signalled fault which is received by FH), which enables the compensation of the $throw$ transaction which terminates the main activity and then behaves as $\overline{dis_{eh}}\ \langle\rangle \mid StopExecC(A) \mid x_s().\overline{dis_{fh}}\ \langle\rangle \mid y_{eh}().y_{fh}().y'().\overline{y}\ \langle\rangle$. Such process disables the event handler, stops the execution of any children scopes and signals the scope termination after having verified that the event handler as well as the fault handler have been disabled and that the entire activity A has been terminated. The process $x_s().\overline{dis_{fh}}\ \langle\rangle$ can move only in one case, that is when the fault is not handled by the scope raising the exception, in this case to disable its fault handler we need to interact with the process handling the $throw$.

Finally, it remains to considerate the case where a compensation is invoked when the main activity is not yet completed. In this case, CH compensates any scope child and signals ($throw$) to the main activity that it has to be terminated.

6. Open issues in WS-BPEL

In this section we present a couple of significant open issues of the hundreds ones in designing WS-BPEL. The open issues list is publicly available on the OASIS web site [33]. Each of the chosen issues to be presented here is firstly described and then a solution is proposed making use of the experience gained developing our algebra. With this attempt we intend to give a concrete contribution towards the improvement of the quality of the BPEL specification, the applicability of BPEL itself and the implementation of real orchestration engines. Furthermore we intend to demonstrate, in real life scenarios, the added value of formal methods.

6.1. Completion condition

We cite the issue 6 of [48]:

> "In BPEL a set of parallel activities is treated as finished if all activities have been completed. In many cases the process does not need to wait for all the concurrent activities to finish for the overall objective to be reached. There should be a way to express the "completion" of the desired objective, causing termination of all "unnecessary" concurrent activities".

In webπ we have an easy way to obtain this behavior which in literature is called speculative parallelism. Speculative parallelism is the parallel composition of several activities, representing alternative ways of achieving a goal, such that, when one completes – the *winner* –, the remaining activities – the *losers* – are abandoned. For the sake of simplicity, we consider only speculative parallelism consisting of two activities (the generalization is trivial) named A and B, respectively. These activities are modelled by two transactions called x_A and x_B, which are inside an outer transaction responsible for deciding the winner and the loser. The definition is as follows:

$$
\begin{array}{l}
(x_A, x_B, resp, ack_A, ack_B) \\
\quad \{\!| \quad \{\!|\overline{req_A}\,\langle\rangle \mid ans_A(); (\overline{resp}\,\langle\langle x_B, ack_A\rangle\rangle \mid ack_A().\mathbf{0}) \;;\; \overline{\overline{cancel_A}\,\langle\rangle}|\!\}_{x_A} \\
\quad\quad \mid \{\!|\overline{req_B}\,\langle\rangle \mid ans_B(); (\overline{resp}\,\langle\langle x_A, ack_B\rangle\rangle \mid ack_B(); \mathbf{0}) \;;\; \overline{\overline{cancel_B}\,\langle\rangle}|\!\}_{x_B} \\
\quad\quad \mid resp(x, y); (\overline{x}\,\langle\rangle \mid \overline{y}\,\langle\rangle) \;;\; \overline{x_A}\,\langle\rangle \mid \overline{x_B}\,\langle\rangle \quad |\!\}_z
\end{array}
$$

The transactions called x_A and x_B send a request to a corresponding service – on names req_A and req_B, respectively – and wait for the answer. On reception of the answer – on names ans_A and ans_B, respectively –, the transactions communicate their end on the private channel *resp*. The message carries two names: the first one is the name of the opposite transaction while the second one is the name of an input where the transaction body is waiting for an acknowledgement. When the outer transaction receives these two names, they are used to cancel the loser and to acknowledge the winner. Each transaction has an associated compensation process able to cancel the task itself. The compensation process of the outer transaction simply invokes the compensations of the inner transactions.

6.2. Dynamic parallel processing

Citing the issue 4 of [48]:

> "BPEL only supports the invocation of a single web service within an invoke activity. In many situations it is required that a particular invoke activity results in the creation of many activity instances where the number of instances is not known at design time but it is calculated at runtime from the contents of a set of data or references. All instances are carried out in parallel and must be synchronized for completion of the activity. Typical example for this type of processing is the sending of a request to a number of services. Processing of such an activity includes fanning out the requests, collecting the results of the requests, and determining the overall (combined) result of the different requests."

The problem could be easily solved in the case we compose such activities in sequence (by using the `while` construct), while to manage a dynamic number of activities in parallel we need to replicate activities and there is no way, in BPEL, to synchronize them and then to implement the phase which collects anche combines the several results. This expressiveness gap is a direct consequence of the fact that BPEL can replicate (i.e. create new instances) activities only starting by a `receive` operation, but it does not provide any mechanisms to synchronize instances, thus preventing to collect the results obtained by the termination of the several instances. On the other hand, webπ$_\infty$ provides the same mechanism to replicate processes (lazy replication). In this case, however, it is possible to synchronize processes by using channels to notify the completion of any instance of the activity to a process which is responsible for collecting the result of any activity instance.

7. Conclusions

In this paper the semantics of a significant fragment of BPEL has been formally defined in term of webπ$_\infty$. In particular, we have shown how the event handler as well as the fault and the compensation handlers, which are the closest ones to error handling, can be programmed by exploiting the event-based mechanisms we propose in our calculus. With this work we intend to give a concrete contribution from the point of view of the orchestration language specifications in terms of unambiguous semantics and of the feasible implementations of BPEL orchestration engines. Finally, we also show how the language we propose is expressive enough to solve some open issues of BPEL, thus proving an expressiveness gap between the two orchestration languages.

The aim of this paper has been analyzing Web services composition and workflow and, in particular, to investigate the WS-BPEL error recovery framework. To get this task feasible we have been forced to simplify the whole language, by considering only the relevant aspects. We intend to complete our investigation in the future considering further details as variables and global states.

We consider this paper as the first step towards the definition of a formal framework for reasoning on WS-BPEL orchestrations. To this end, as future work we intend to define some behavioural equivalences on $\texttt{web}\pi$ that could be used to analyse and compare the behaviour of $\texttt{web}\pi$ orchestrations. Moreover, given that we provide WS-BPEL semantics in terms of $\texttt{web}\pi$, once having defined such an equivalence we can directly inherit a behavioural equivalence relation also for WS-BPEL processes by allowing thus the formal comparison of WS-BPEL processes.

Some considerations about the choice of a π-calculus based language are needed. One peculiarity of such language is mobility: it is possible to transmit channel names that then can be used by any process receiving them. This aspects is essential in the formalization of BPEL and in particular plays an important role in the formalization of interaction with request–response services. Indeed, the invoker must send to the service a channel name to be used to return the response, thus we have to exploit name mobility. Moreover, the importance of mobility grows if we also consider WS-Addressing which makes it possible to dynamically define the access point of the services to be invoked. We consider the $\texttt{web}\pi_\infty$ language expressive enough to describe also WS-Addressing, such a investigation is left as future work.

Other considerations about the formalisms to be used to formally describe the behavior of BPEL, and in general of workflow-based orchestration languages, follow.

Van der Aalst in [46] raises a number of challenges for those advocating the use of π-calculus in the context of workflow. He underlines that:

> In the debate on Petri nets versus π-calculus many players in the “Web Service Composition Languages marketplace” are using demagogic arguments not based on concrete facts. [...] Hopefully, this note will contribute to exposing the people that try to “hype” things like π-calculus only for marketing purposes. Note that the big discrepancy between the “Pi-hype" and reality will not only limit the applicability of Web Service Composition Languages but also discredit a beautiful scientific framework like π-calculus.

One of the challenge raised by van der Aalst is

> “Let the people that advocate BPEL4WS, etc. show the precise relation between the language and some formal foundation. People that cannot do this but still claim strong relationships between their language and e.g., π-calculus only cause confusion.”

In this paper we showed that the gap existing between BPEL and the π-calculus is significant yet not unsurmountable. The main issues we underlined are essentially about mobility and sequencing. On these points the language is not very faithful to the theory creating confusions. Other source of confusion is the fact that BPEL has not a formally defined semantics and this significantly complicates the understanding and the attempt of comparing its behavior with the one of the π-calculus. All these things, in our opinion, should convince readers to be very careful when associating process algebra and workflow languages.

Another challenge we intend to remark is

> Let the people that advocate a particular formal model (e.g., π-calculus) in the context of languages like BPEL4WS, etc. demonstrate the use of analysis methods and tools based on this formal model (in some real life setting).

This is not an evident task. Future developments of our work consist in considering the research directions suggested by [25,29] in developing type systems and contracts for web services composition.

Acknowledgments

The authors would like to acknowledge Cosimo Laneve, Nadia Busi, Claudio Guidi, Andrea Carpineti, Alessandro Perfetti and Enrico Tosi for their comments and contributions to the paper.

References

[1] T. Andrews, F. Curbera, et al., Web service business process execution language, Working Draft, Version 2.0, 1 December 2004.
[2] Microsoft BizTalk Server, Available from: <http://www.microsoft.com/biztalk/default.asp>, Microsoft Corporation.
[3] M. Berger, K. Honda, The two-phase commit protocol in an extended π-calculus, Proceedings of the 7th International Workshop on Expressiveness in Concurrency, EXPRESS'00, The ENTCS Series, vol. 39, Elsevier, 2000.
[4] M. Berger, Towards abstractions for distributed systems, Ph.D. thesis, Imperial College, London, 2002.
[5] M. Berger, Basic theory of reduction congruence for two timed asynchronous pi-calculi, in: Proceedings of CONCUR'04, LNCS 3170, pp. 115–130.
[6] L. Bocchi, C. Laneve, G. Zavattaro, A calculus for long running transactions, in: FMOODS'03, Paris, LNCS 2884, December 2003, pp. 124–138.
[7] G. Booch, Object-Oriented Analysis and Design with Applications, Addison-Wesley, 1993.
[8] R. Bruni, H. Melgratti, U. Montanari, Theoretical foundations for compensations in flow composition languages, in: Proceedings of POPL 2005, 32nd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, ACM press, pp. 209–220.
[9] R. Bruni, C. Laneve, U. Montanari, Orchestrating transactions in join calculus, in: Proceedings of CONCUR'02, LNCS 2421, pp. 321–337.
[10] M. Butler, C. Ferreira, An operational semantics for StAC, a language for modelling long-running businness transactions, in: Proceedings of COORDINATION 2004, LNCS 2949, pp. 87–104.
[11] M. Chessel, D. Vines, C. Griffin, V. Green, K. Warr, Business process beans, System design and architecture document, Technical report, IBM UK Laboratories, January 2001.
[12] E. Christenses, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL 1.1), Available from: <www.w3.org/TR/wsdl>, W3C, Note 15, 2001.
[13] P. Gardner, C. Laneve, L. Wischik, The fusion machine. Theory and practice, CONCUR 2002, LNCS 2421, pp. 321–337.
[14] C. Guidi, R. Lucchi, M. Mazzara, A formal framework for Web services coordination, in: 3rd International Workshop on Foundations of Coordination Languages and Software Architectures, London, Electronic Notes in Theoretical Computer Science, Elsevier, in press.
[15] I. Houston, M.C. Little, I. Robinson, S.K. Shrivastava, S.M. Wheater, The CORBA activity service framework for supporting extended transactions, Softw. Pract. Exper. 33 (4) (2003) 351–373.
[16] T. Hoare, Long-running transactions, Powerpoint presentation available at the <research.microsoft.com/> web pages.
[17] M.N. Huhns, M.P. Singh, Service-oriented computing: key concepts and principles, IEEE Internet Comput. (January) (2005) 75–81.
[18] N. Kavantzas, Aggregating Web services: choreography and WS-CDL, Available from: <http://lists.w3.org/Archives/Public/www-archive/2004Jun/att-0008/WS-CDL-April2004.pdf>.
[19] N. Kavantzas, D. Burdett, G. Ritzinger, Y. Lafon, Web Services Choreography Description Language Version 1.0, 12 October 2004.
[20] R. Khalaf, S. Tai, S. Weerawarana, Web services, the next step: a framework for robust service composition, in: M.P. Papazoglou, D. Georgakopoulos (Eds.), CACM, Special Issue on Service-Oriented Computing, October 2003.
[21] C. Laneve, B. Victor, Solos in concert, Math. Struct. Comput. Sci. 13 (5) (2003).
[22] C. Laneve, G. Zavattaro, Foundations of Web transactions, in: Proceedings of FOSSACS 2005, LNCS 3441, pp. 282–298.
[23] F. Leymann, Web Services Flow Language (WSFL 1.0), Available from: <www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>.
[24] M. Little, Web services transactions: past, present and future, Available from: <www.idealliance.org/papers/dx_xml03/html/abstract/05-02-02.html>.
[25] A. Igarashi, N. Kobayashi, A generic type system for the π-calculus, in: Proceedings of ACM Symposium Conference on Principles of Programming Languages (POPL), 2001, ACM Press, pp. 128–141.
[26] M. Mazzara, Timing issues in Web services composition, in: Second International Workshop on Web Services and Formal Methods (WS-FM 2005), LNCS 367, pp. 287–302.
[27] M. Mazzara, S. Govoni, A case study of Web services orchestration, in: Proceedings of COORDINATION 2005, LNCS 3454, pp. 1–16.
[28] M. Mazzara, R. Lucchi, A framework for generic error handling in business processes, in: First International Workshop on Web Services and Formal Methods (WS-FM), Pisa 2004, Electronic Notes in Theoretical Computer Science, vol. 105, Elsevier.
[29] L.G. Meredith, S. Bjorg, Contracts and types, Commun. ACM 46 (10) (2003).
[30] R. Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, 1999.
[31] R. Milner, Function as processes, Math. Struct. Comput. Sci. 2 (2) (1992) 119–141.
[32] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, J. Inform. Comput. 100 (1992) 1–77.
[33] Oasis, Available from: <www.oasis-open.org/>.
[34] C. Peltz, Web services orchestration and choreography, IEEE Comput. 36 (10) (2003) 46–52.
[35] J. Parrow, B. Victor, The fusion calculus: expressiveness and symmetry in mobile processes, in: Proceedings of LICS'98, IEEE Press, pp. 176–185.
[36] C.A. Petri, Kommunikation mit Automaten, Ph.D. thesis, Technische Universitt Darmstadt, Fakultt Matematik und Physik, 1962.
[37] D. Sangiorgi, D. Walker, The π-Calculus: A Theory of Mobile Processes, Cambridge University Press, 2001.
[38] H. Smith, P. Fingar, Workflow is just a pi process, Unpublished discussion paper, Available from: <www.workflowpatterns.com>.
[39] SOAP – Simple Object Access Protocol, Available from: <www.w3.org/TR/soap>.
[40] A.S. Tanenbaum, Modern Operating Systems, Prentice-Hall, 2001.
[41] S. Thatte, XLANG: web services for businnes process design, Available from: <www.gotdotnet.com/team/xml/wsspecs/xlang-c>, Microsoft Corporation, 2001.
[42] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison Wesley Professional, 2002.
[43] UDDI – Universal Description, Discovery and Integration of Web Services, Available from: <www.uddi.org/specification.html>.

[44] M. Viroli, Towards a formal foundation to orchestration languages, First International Workshop on Web Services and Formal Methods (WS-FM), Electronic Notes in Theoretical Computer Science, vol. 105, Elsevier, 2004.
[45] World Wide Web Consortium (W3C), Available from: <http://www.w3.org>.
[46] W.M.P. van der Aalst, Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype", Unpublished paper.
[47] WS-Addressing, Available from: <http://www.w3.org/TR/2004/WD-ws-addr-core-20041208>.
[48] BPEL Open Issues List, Available from: <http://www.oasis-open.org/apps/group_public/download.php/11285/wsbpel_issues34.html>.
[49] XML – Extensible Markup Language 1.0, Available from: <http://www.w3.org/TR/2000/REC-xml-20001006>.
[50] XML Schema Specification, Available from: <http://www.w3.org/XML/Schema>.