A Survey of Web services Orchestration and Choreography with Formal Models

Damjan Kovač*, Denis Trček

University of Ljubljana, Faculty of Computer and Information Science, Laboratory of e-media, Tržaška cesta 25, SI-1001 Ljubljana, Slovenia

Abstract

The service-oriented architecture (SOA) promises a new generation of information systems applications based on a new set of standards for enabling self-describing interoperable Web services. Web service orchestration and choreography are both concerned with the composition of Web services to meet the needs of business processes. There are two important standards for modeling and implementing workflows and business processes based on Web services: BPEL follows the orchestration paradigm, and WS-CDL covers the choreography. This paper gives a formal methods focused survey of BPEL and WS-CDL languages. Basic language constructs are presented as core building blocks for business processes. From the perspective of composing Web services to execute business processes, the orchestration is a more flexible approach compared to the choreography. For that reason we concentrate on some BPEL orchestration aspects using formal models: error, event, and compensation handling using extended version of $\pi$-calculus, correlation mechanisms using process algebra, and control flow constructs using Petri nets. All of the formal models allow the analysis and verification of BPEL processes what gives a good starting point for better understanding and further investigations on selected BPEL issues. This could also contribute towards the improvement of the quality of BPEL specification, the applicability of BPEL itself and the implementation of real orchestration engines.

Key words: SOA, Web Services, Orchestration, Choreography, BPEL, WS-CDL, Correlation, Control flow

* Corresponding author. Tel.: +386-1-4768921; fax: +386-1-4264647

Email addresses: damjan.kovac@fri.uni-lj.si (Damjan Kovač), denis.trcek@fri.uni-lj.si (Denis Trček).

Tel.: +386-1-4768918

Preprint submitted to Elsevier

January 9, 2011
1 Introduction

The most recent answer to the integration challenge of existing and legacy business applications is the Service Oriented Architecture (SOA) on the basis of Web services technologies. The general view of the SOA sees different business applications of different legacy and new developed applications in a standard way through the Web services. Developing the Web services and exposing their functionalities is not sufficient. The further step is a need to compose these functionalities in the right order to define business process, which will make use of exposed functionalities.

This is where the BPEL (Business Process Execution Language) [1,12] becomes important. BPEL is a language for the definition and execution of business processes using existing Web services. BPEL enables the top-down realization of SOA through composition, orchestration, and coordination of Web services. So it provides a straightforward way to compose several Web services into new composite services (business processes). BPEL is built on the foundation of XML and Web services; it uses an XML-based language that supports the Web services technology stack, which includes SOAP [26], WSDL [30], UDDI [17], WS-ReliableMessaging [3], WS-Addressing [28], WS-Coordination [13], and WS-Transaction [14,15].

Web services can be composed in two ways – by an orchestration and a choreography:

- Orchestrations refer to a composed business process that may use both internal and external Web services to fulfill its task. The business process is controlled by one of the agent in the system. The process is described at the message level, i.e. in terms of message exchanges and execution order.
- Choreography addresses the interactions that implement the collaboration between services. Multiple agents are present where each of them describes its own part in interaction.
- Orchestration and choreography address different perspectives. Orchestration is focused on the internal behavior of a business process (process model). Choreography is focused on the external perspective, looking at process interaction (interaction model). These perspectives are essential aspects of the SOA.

There are two different ways of describing business processes that support orchestration and choreography:

- Executable processes allow specifying the exact details of business processes. They follow the orchestration paradigm and can be executed by an orchestration engine.
- Abstract business protocols allow a specification of the public message ex-
change between parties only. They do not include the internal details of process flows and are not executable. They follow the choreography paradigm.

BPEL follows the orchestration paradigm. Choreography is covered by other standards, such as WSCI (Web Service Choreography Interface) [27] and WS-CDL (Web Services Choreography Description Language) [29]. It is worth mentioning that the choreography has not gained support from the industry which would be comparable to BPEL.

BPEL is a workflow language that contains a number of primitive and structured activities. This leads to possibility of complex BPEL processes which are hard to analyze, because the semantics of BPEL is not always clear as its specification is presented in an informal way. Important feature of BPEL is that it supports the stateful, long-running interactions involving many parties. As a consequence there are many aspects that should be considered: concurrency, synchronization, correlations, events, faults and compensations handling [21,9]. Researchers attempt to offer theoretical and formal foundations of those aspects; there are many formal mechanisms and methods for analyzing and solving open issues in BPEL, e.g., process algebras [24,25,9,19,33], timed automata [10,11,21], and Petri nets [4,18,6].

This paper presents a brief overview of orchestration and choreography languages with some formal methods. It is organized as follows. Section 2 describes the semantics overview of BPEL orchestration language. Section 3 gives a brief overview of WS-CDL choreography language. In section 4 some BPEL issues are pointed out with their formal representation. Section 5 provides some concluding remarks.

2 BPEL overview

BPEL [1] is an XML-based specification language for business processes orchestration through the interaction of different, existing or possibly dynamically emerging Web services. Business processes can be described in two ways. An executable business process models actual behavior of a participant in a business interaction. Business protocols, in contrast, use process descriptions that specify mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. The process descriptions for business protocols are called abstract processes. BPEL is meant to be used to model the behavior of both executable and abstract processes [1].

BPEL process definition can be executed by an orchestration engine, which is the central coordinator. The engine reads the BPEL document and invokes the
necessary Web services in the specified order, be it sequentially or in parallel. The process itself will be offered as a new Web service to the clients and can be invoked in the same way. BPEL can express a conditional behavior. For example, an invocation of a Web service can depend on the value of a previous invocation. It can also constructs loops, declares variables, copies and assigns values, defines fault handlers. By combining all the constructs a complex process processes can be defined in an algorithmic manner.

BPEL is built on top of the WSDL [30] language for describing the interface of Web services. A Web service interface is specified in terms of port types, operations, and messages (e.g., in an object-oriented area would correspond to interface types, the method names, and method types). As far as BPEL is concerned, the port types are list of operations that are SOAP [26] based messages (e.g., XML data records). A BPEL process can be synchronous (use one, or more request-response operation) or asynchronous (use one or more one-way operation). A synchronous BPEL process blocks the client until the process finishes and returns a result to the client. An asynchronous process does not block the client as it uses a callback to return the result (if any).

The structure of BPEL process is described by the following fragment of XML Schema [32,16]:

```xml
<element name="process" type="bpws:tProcess"/>
<complexType name="tProcess">
<complexContent>
<extension base="bpws:tExtensibleElements">
<sequence>
<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="eventHandlers" type="bpws:tEventHandlers" minOccurs="0"/>
<group ref="bpws:activity"/>
</sequence>
<attribute name="name" type="NCName" use="required"/>
<attribute name="targetNamespace" type="anyURI" use="required"/>
<attribute name="queryLanguage" type="anyURI"/>
<attribute name="expressionLanguage" type="anyURI"/>
<attribute name="suppressJoinFailure" type="bpws:tBoolean" default="no"/>
<attribute name="enableInstanceCompensation" type="bpws:tBoolean" default="no"/>
<attribute name="abstractProcess" type="bpws:tBoolean" default="no"/>
</complexType>
</complexType>
```

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

A BPEL process consists of steps. Each step is called an activity. BPEL sup-
ports *basic* (atomic actions) and *structured* (for concurrency and synchronization) activities. The XML Schema fragment describes activity elements [9]:

```xml
<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 next subsections describe the main basic and structured activities, the event, fault and compensation handling mechanisms. Some syntactical details are omitted; elements are expressed using BNF (Backus-Naur Form) formalisms instead of XML Schema. Frequency operators *elem?, elem+, elem* mean respectively zero or one, one or more, zero or more occurrences of the element *elem*.

### 2.1 Basic activities

The following subsections describe some basic (primitive) BPEL activities [9,7,8], which are used for message exchange. Basic activities are: *invoke, receive, reply, throw, compensate, empty, terminate* (*exit* in BPEL 2.0 [12]), *assign*, and *wait*.

#### 2.1.1 Invoke

The activity represents an invocation of a Web service operation, which can be synchronous or asynchronous accordingly with the interaction modality used by invoked service. The invoke activity is defined as follows:

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

In short, *partnerLink*, *portType* and *operation* are used to denote the specific operation supplied by a certain partner, the end-point of the invoked operation and the transport protocol for SOAP requests (HTTP, SMTP,...);
inputVariable and outputVariable denotes the variables passed as input to and as output from the service.

2.1.2 Receive

The activity waits for a request and has to specify the interacting partner in terms of its role in the process, port type and operation. The syntax is as follows:

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

2.1.3 Reply

The activity is used to generate a response. There is an important constraint that the reply activity should always be preceded by a receive activity for the same partner, port type and operation. Syntax is as follows:

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

2.1.4 Throw

Throw is used to explicitly signal an internal fault or an exception, which has to have a globally unique name. When the throw is executed, the name has to be specified as well as some variables containing information about the fault. Faults are caught by fault handlers. The syntax is as follows:

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

2.1.5 Compensate

The activity is used to invoke the compensation handler (to undo the effects of already completed activities), which are associated to scopes. Compensation handlers can be invoked only by a fault handler or by a compensation handler associated with outer scopes. If no compensation handler is defined, the default handler compensates all the children scopes. The syntax is:

```xml
<compensate scope="ncname"? standard-attributes>
  standard-elements
</compensate>
```
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 message exchange between process instances involved in business process. 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

Sequence defines sequential composition of activities. A sequential activity contains one or more activities that are executed in the order in which they are listed within the sequence element. The sequence activity finishes when the final activity in the sequence has completed. The syntax:

```xml
<sequence standard-attributes>
  standard-elements activity+
</sequence>
```

2.2.2 Flow

Flow allows parallel execution of basic activities. The flow construct provides concurrency and synchronization. A flow finishes when all of the parallel activities it contains have completed. The simplest use of this activity is equivalent to a nested concurrency construct. The syntax is defined as it follows:

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

2.2.3 Links

This construct allows setting up certain constraints on the elements inside a flow activity. The control link – constraint specifies some order on the execution of the parallel activities. Using links a flow can be viewed as a graph – 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. A
control link with its name defines a connection between two activities: one defined as a source and one defined as a target. Both, the source and the target, must define explicitly their role in the syntax. The source activity may also specify a transition condition that is a Boolean expression over the process variables; if it is not defined it is intended to be true. A control link between activities A and B indicates that B cannot start before A has either completed or has been skipped. B can only be executed if its associated join condition evaluates to true; otherwise B is skipped. The join condition is expressed in terms of tokens carried by control links leading to B. Each token can take positive (true) or negative (false) value. An activity X propagates a token with a positive value along an outgoing link L if and only if X was executed and the transition condition associated to L evaluates to true. The process by which positive and negative tokens are propagated along controls links, causing activities to be executed or skipped, is called dead path elimination [18]. Control links can cross the boundaries of most structured activities but they cannot cross a scope.

2.2.4 Switch

This construct represents a case-switch construct for implementing branches. The activity consists of an ordered list of one or more conditional branches defined by case elements that can be followed by an optional otherwise branch. The case branches are evaluated in the order in which they appear. The condition is Boolean expression of process variables. The first branch whose condition is true defines the activity to be performed by the switch. The whole activity completes when the selected activity completes. The syntax:

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

It is worth mentioning that the new BPEL 2.0 [12] renamed switch activity to if activity with equivalent semantics. The if syntax is as follows:

```xml
<if standard-attributes>
  standard-elements
  <condition expressionLanguage="anyURI">bool-expr</condition>
  activity
  <elseif>*</n
  <condition expressionLanguage="anyURI">bool-expr</condition>
  activity
  </elseif>
  <else>?</n
  activity
  </else>
</if>
```
2.2.5 Pick

This construct performs 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. When the pick activity has accepted an event, the other events are no longer accepted. Events can be message events handled with the onMessage activity or alarm events handled with the onAlarm activity [12]. The pick activity waits the occurrence of one of the defined events and performs the associated child activity. If more than one of the events occurs then the selection depends on which one occurred first. When the child activity completes then the pick activity completes. The syntax is as follows:

\[
\text{<pick createInstance="yes|no"? standard-attributes>}
\text{standard-elements}
\text{<onMessage partnerLink="ncname" portType="qname" operation="ncname" variable="ncname"?>+}
\text{<correlations>?}
\text{<correlation set="ncname" initiate="yes|no"?>+}
\text{</correlations>}
\text{activity}
\text{</onMessage>}
\text{<onAlarm (for="duration-expr" | until="deadline-expr")>*
activity
</onAlarm>}
\text{</pick>}

2.2.6 Scope

Scopes are hierarchically organized parts into which a complex business process can be divided. Scope provides a behavior context for each activity inside. Each scope has a primary activity that defines the behavior of a scope for normal execution. The primary activity of the scope can be basic activity such as invoke, or it can be a structured activity such as sequence or flow. If the primary activity of a scope is a structured activity, it can have many nested scopes, in which the nesting depth is arbitrary. The scope is shared among all the nested activities. A scope can also have nested scopes with arbitrary depth.

Besides activities, scopes can also define local correlation sets, fault, compensation, and event handlers. Variables definition inside a scope holds until it is active and they are visible only within the defined scope. 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 compensation handler, are in essence concerning with different stages of the computation. In particular, fault handler is the mechanism used during execution of the scope activities, while the compensation is used after the successful termination of
the scope. Fault handlers attached to a scope handle faults of all the nested activities of the scope. Faults not caught in a scope are rethrown to the parent scope. Scopes in which faults have occurred are considered to have ended abnormally even if a fault handler has caught the fault and not rethrown it. The syntax is as follows:

```xml
<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.

### 2.2.7 Compensation handler

This construct enables a process to undo the effect of a successfully completed scope [18]. The compensation handler is allowed to be performed when the scope terminates successfully – if no fault has been thrown during the execution and all the activities have been performed. It is allowed to access the variables state as they appear at its termination as to interact with other services. Compensation handlers are defined by the following syntax:

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

When the compensation activity is executed for a given scope, the compensation handler of this scope will be executed when it is available. This may involve the execution of the compensation handlers associated to only the subscopes of the above given scope. The compensation handler can be invoked by using the `compensate` activity (Section 2.1.5), which names the scope for which the compensation is to be performed, that is, the scope whose compensation handler is to be invoked. A compensation handler for a scope is available for invocation only when the scope completes normally [1,12].
2.2.8 Fault handler

This construct defines reactions to internal or external faults that occur during the execution of a scope. The catch element handles a fault specified by a fault name. The catchAll element is able to capture any fault. The throw activity (Section 2.1.4) signals faults and interrupts the normal execution of the scope activating the relative fault handler. If it is not defined, a default handler executes. The default handler compensates all the children scopes and the pending fault is rethrown to the parent scope as the whole BPEL process is considered as scope by specification [1,12]. The scope that manages the signaled fault has not the normal termination and its compensation handler is not allowed to be performed. The syntax of the fault handler is the following:

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

2.2.9 Event handler

The handler defines event-action rule associated with a scope. An event handler is enabled when its associated scope is under execution and may execute concurrently with the main activity of the scope. When an occurrence of the event associated with an enabled event handler is registered (this may be a timeout with onAlarm construct or a message receipt with onMessage construct), the activity of the handler is executed while the scope’s main activity continues its execution. Messages that are caught by an event handler are consumed and it is not allowed that a message sent from a certain partnerLink, portType and operation can be simultaneously consumed by more than one event handler (the same holds true for the receive activity). Event handlers can be defined by the following syntax:

```xml
<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">*
        activity
      </correlation>
    </correlations>
    activity
  </onMessage>
  <onAlarm for="duration-expr"? until="deadline-expr"?>*
    activity
  </onAlarm>
</eventHandlers>
```
3 WS-CDL overview

As explained in the previous sections BPEL follows the orchestration paradigm of Web services composition where the central process takes control over the involved Web services and coordinates the execution of different operations (describes process model). The choreography on the other hand does not rely on the central coordinator. Each involved Web service knows exactly when to execute its operations and whom to interact with. The choreography is a collaborative effort focused on exchange of messages from a global point of view (describes interaction model) where all participating services are treated equally [2]. All participants of the choreography need to be aware of the business process, the operations to execute, the messages to exchange, and the timing of message exchanges.

Choreography is covered by standards such as WSCI (Web Services Choreography Interface) [27] and more novel WS-CDL (Web Services Choreography Description Language) [29,2]. We focus on WS-CDL description because it can be actually used in conjunction with BPEL in modeling and implementing business processes. The purpose of the WS-CDL is to define multi-party contracts, which describe the externally observable behavior of Web services and their clients (usually other Web services) by describing the message exchanges between them. WS-CDL is an XML-based language standard. As we consider some open BPEL orchestration aspects in the following sections, only the basic overview of WS-CDL is given.

It is worth noting the fundamental difference between the concept of choreography and the concept of behavioral interface (i.e. BPEL abstract process). The choreography focuses on interactions seen from the global viewpoint, while behavioral interfaces focus on communication actions seen from the viewpoint of one of the participants [33]. This difference is orthogonal to control flow and the same control flow constructs could be used for describing choreographies and interface behaviors. WS-CDL and BPEL could share the same set of control flow constructs. Choreography models are inherently design-level artifacts and are not intended to be directly executed (like BPEL orchestration models).

3.1 Choreography Package

A WS-CDL choreography description is contained in a package and is essentially a container for a collection of activities that may be performed by one or more of the participating Web services. The behavioral aspect of the message exchanges is described in one or more choreographies (in the Choreography-
Notation section) in terms of activities. There are three types of activities in WS-CDL, namely control-flow activities, workunit activities and basic activities. The syntax of the root package is:

```xml
<package name="ncname" author="xsd:string"? version="xsd:string"? targetNamespace="uri"
xmlns="http://www.w3.org/2004/12/ws-chor/cdl">
  informationType*  
token*  
tokenLocator*  
roleType*  
relationshipType*  
participantType*  
channelType*  
Choreography-Notation*
</package>
```

3.2 Control-flow activities

These activities are block-structured in the sense they enclose a number of sub-activities. They specify ordering rules of activities within that choreography. The control-flow activities include sequence, parallel, and choice. The detailed syntax is described by [9,2].

A sequence activity describes one or more activities that are executed in sequential order – in the same order that they are defined. A parallel activity describes one or more activities that can be executed in concurrent order. This activity completes successfully when all defined activities performing work within it complete successfully.

A choice activity enables the selection of only one activity among the set of alternatives. The choice activity has two types of choices: (a) data-driven choice, which is based upon a Boolean condition on data variables and (b) event-driven choice that depends on the occurrence of one among a set of competing events. Such events may be the consequence of an interaction or of an action in the choreography that results in certain variables being populated.

3.3 Workunit activity

A workunit activity prescribes the constraints that have to be fulfilled for making progress and thus performing actual work within choreography. It describes the conditional and repeated execution of an activity. It has several parts, including a reference to the enclosed activity, a guard, a block, and a repetition condition that are modeled as Boolean expressions. The guard and repeat conditions are used to determine whether or not the enclosed activity is executed one or more time. The execution of workunit activity will start with the evaluation of the guard condition; if it is true, the enclosed activity
is executed; otherwise it is skipped. If the activity is executed, and once is completed, the repeat condition is evaluated and depending on the result, the activity can be executed again. Thereby it provides a mean for structured loops.

The **block** condition is used to determine if the **guard** and **repeat** conditions should wait for variables to become available before being evaluated. A **workunit** completes successfully when all its enclosed actions complete successfully. The syntax is as follows:

```
<workunit name="ncname"
    guard="xsd:boolean XPath-expression"?
    repeat="xsd:boolean XPath-expression"?
    block="true|false"?
    Activity-Notation
</workunit>
```

### 3.4 Basic activities

These activities represent request/response actions for local activities and invocations of external services. There are five kinds of the basic activities: **interaction**, **perform**, **assign**, **noaction** and **silentaction**.

The activities **noaction** and **silentaction** describe points in a choreography where one participant (role) performs no action or performs an action that does not affect the rest of choreography. The **assign** activity is used to transfer the value of one variable to another variable within a participant.

The **perform** activity is used to call another choreography to be performed within the context of the execution choreography. The called choreography may be defined within the same package as the caller, or it may be from a completely separate package that has been imported. The **perform** activity has a binding mechanism to allows the caller’s variables to be bound to free variables in the called choreography.

The most important element of WS-CDL is the **interaction** activity [2]. It describes an exchange of information between participants with a focus on the receiver of the information. An interaction may specify either a request, a response, or a request-response type of information exchange. Its description has three main parts corresponding to (i) the participants involved (element **participate**); (ii) the information being exchanged; and (iii) the channel for exchanging the information. Information is described by variables, which are used to represent three different types of information: an application-dependent information (e.g., product code), a state information (e.g. order received), and a channel information. Variables contain values and have an information type. More specific details and syntax is described by [29,2,33].
4 Formal aspects of orchestration

An orchestration language must support activities from both communicating with other Web services and handling workflow semantics. As presented in the previous sections BPEL provides basic and structured activities to address these two issues. Basic activity can be seen as a basic 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. BPEL covers most of the business process requirements [20]: asynchronous interactions, flow coordination, business transaction activity and management.

BPEL provides different mechanisms to deal with some particular situations. Abnormal situations are covered by exception, event and compensation handling. Fault handler is used during the execution of an activity while the compensation handler when it has been completed. Compensations are related with long running transactions as web transactions can be considered as transactional interactions between organizations. The ACID properties of transactions are not enough as they make sense only when trusted parties are involved over short period of interaction time. SOA presumes loosely coupled world of autonomous trading partners where hard locking of local resources is unfeasible. BPEL introduces the concept of long running transactions as compensation and could be one of the possible methods of implementing WS-BusinessActivity [15] specification, which heavily relies on WS-AtomicTransaction [14], and WS-Coordination [13] specifications. Thus the compensation is an important aspect in BPEL and the topic of further research. But due to its complexity there is no work currently in automatic verification of BPEL that supports compensation completely – as stated by [21].

BPEL is not equipped with formal semantics and since it includes a large number of aspects, it is difficult to formally reason on process behavior. There are many issues that are a subject of a research and investigation using different formal methods. Some of the actual BPEL issues as part of process behavior are the following:

- dynamism of activities, concurrency, death paths and deadlocks elimination, time analysis of the process, reachability of activities, behavior specification [22,4,11,18];
- correlation mechanisms as concurrent behavior of process instances [24,21];
- describing relationships and semantics of process elements using ontology [23];
- modeling control flows, formal semantics of the composition operators, verification of different properties, detection of inconsistence, timed analysis
properties [6,21];
• error, event, and compensation handling mechanisms [9].

The list of these issues is quite long. We concentrate on some of them and try to formally describe them with various formal methods: π-calculus, process algebras, Timed Automata, and Petri nets.

4.1 Error, event and compensation handling

As explained, BPEL provides three different mechanisms, which allow dealing with abnormal situations: fault, compensation and event handling. Documentation shows ambiguities especially about the interdependencies among these mechanisms. We present a model with the smallest set of operators, which meet behaviors presented above. A process algebra π-calculus [10] is used which is appropriate for the orchestration purposes. Its syntax represents processes, parallel composition of processes, synchronous communication between processes through channels, creation of fresh channels, replication of processes, and nondeterminism. A process is an abstraction of an independent thread of control. A channel is an abstraction of the communication link between two processes.

Although the π-calculus seems to offer a convenient framework for orchestration, it does not support any transactional mechanism. An extension of the π-calculus is taken into consideration to support transactions [9].

The syntax of processes relies on an infinite set \( \mathcal{N} \) of names which are ranged over by \( u, v, w, x, y, z \). Tuple (vector) of names \( \{u_1, \ldots, u_n\} \) is denoted by \( \tilde{u} \), \( n \) is understood from the context. \( I \) is a finite nonempty set of indexes. The process \( P \) can be defined as follows:

\[
P ::= 0 \quad \text{Inaction}
\]

\[
| \quad P \mid Q \quad \text{Parallel composition}
\]

\[
| \quad \sum_{i \in I} P_i \quad \text{Alternative composition}
\]

\[
| \quad x(u).P \quad \text{Input}
\]

\[
| \quad \pi(u).P \quad \text{Output}
\]

\[
| \quad (x)P \quad \text{Restriction}
\]

\[
| \quad \text{if } (x = y) \text{ then } P \text{ else } Q \quad \text{Conditional}
\]

\[
| \quad !P \quad \text{Replication}
\]

\[
| \quad \langle |P; Q| \rangle_x \quad \text{Transaction}
\]
As shown, $P$ can be void process $0$. $P \mid Q$ denotes a process composed of $P$ and $Q$ running in parallel. Alternative composition $\sum_{i \in I} P_i$ indicates a process that behaves like one of $P_i$ (e.g., in binary summation $P_1 + P_2$ like $P_1$ or $P_2$).

An input $x(u).P$ (also called positive prefix) denotes a process that waits to read a name $u$ from the channel (port) $x$ and then behaves like $P$; to be more specific $x(u).P$ inputs an arbitrary name $z$ at port $x$ (e.g., a message $\overline{x}(z)$ that equals $\overline{x}(z,0)$ and then behaves like $P(z/u)$ – it means a substitution of free occurrence $u$ with $z$). An output $\overline{x}(u).P$ (also called negative prefix) denotes a process that sends the name $u$ along the channel (port) $x$ and then behaves like $P$. A restriction $(x)P$ behaves as $P$ except that inputs and messages on $x$ are prohibited. A conditional branch is based on name equality.

A replication $!P$ denotes an infinite number of copies of $P$ running in parallel; as a consequence a replicated input can be defined: $!x(u).P$ consumes $\overline{x}(w)$ and behaves like $P(w_i/u_i)!x(u).P$. A transaction process is an extension to $\pi$-calculus; $\langle|P;Q|\rangle_x$ behaves as the body $P$ until a transaction abort $\overline{x}(\epsilon)$ is signaled and in this case behaves as the compensation $Q$. The input $x(u).P$ and restriction $(x)P$ are binders of names $u$ and $x$. The scope of these binders is the process $P$; $\text{bn}(P)$ is a set of bound names of $P$ and $\text{bn}_i(P)$ is set of bound names of $P$ with inputs and replicated inputs as binders. The free names $\text{fn}(P)$ of $P$ are just those names which occur in $P$ not bound either by an input or by a restriction. We write $\text{fn}(P_1, P_2, \ldots) = \text{fn}(P_1) \cup \text{fn}(P_2) \cup \ldots$. The following equivalence can be given: $\text{fn}(\langle|P;R|\rangle_x) = \text{fn}(P) \cup \text{fn}(R) \cup \{x\}$.

If we want to express BPEL constructs as $\pi$-calculus processes we need to consider only processes from a set $\text{Process}$ that contains well-formed $\pi$-calculus processes. The definition follows.

**Definition 4.1.1** A process is well-formed if the two conditions are met:

(a) Received names cannot be used as subjects of input or of replicated inputs – if a process $P$ contains $u(v)$ or $!u(v)$ then $u \notin \text{bn}_i(P)$.

(b) 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 $P_1 \mid \langle|P;Q|\rangle_x \mid P_2 \mid \langle|R;S|\rangle_y \mid P_3$ for some $P_1$, $P_2$, and $P_3$ then $x \neq y$.

The first property says that if a name is received by a certain process it does not use that name to perform input on it. This means that a situation where different Web services support the same operation can be avoided. The second property is used to avoid ambiguity on scope names reflecting BPEL itself that is no any transaction is univocally identified.

The semantics of the language is consisting of the laws, which govern the static

\footnote{Note that an output does not bind a name.}
relations between processes and the laws that rule their interaction. The first step is achieved by defining a static structural congruence relation over the processes. It consists of collection of axioms that allow minor manipulation on the process structure.

**Definition 4.1.2** The structural congruence is the least congruence satisfying the abelian monoid laws for parallel composition (associativity, commutativity, \(0\) as identity element) with respect to the following axioms:

(a) Scope laws
\[
(u)0 \equiv 0, (u)(v)P \equiv (v)(u)P
\]
\[
P \parallel (u)Q \equiv (u)(P \parallel Q) \text{ if } u \notin \text{fn}(P)
\]
\[
\langle |(z)P; Q| \rangle_x \equiv \langle |P; Q| \rangle_x \text{ if } z \notin \text{fn}(Q) \cup \{x\}
\]

(b) Transaction laws
\[
\langle |0; Q| \rangle_x \equiv 0
\]
\[
\langle |\langle |P; Q| \rangle_y | R; R'| \rangle_x \equiv \langle |P; Q| \rangle_y | \langle |R; R'| \rangle_x
\]

(c) Floating laws
\[
\langle |z(u) | P; Q| \rangle_x \equiv z(u) | \langle |P; Q| \rangle_x
\]

The first transaction law defines committed transactions that are equivalent to \(0\) and cannot fail anymore. The second law moves transactions outside parent transactions but they may still affect children (nested) transactions by means of transaction names. Floating law moves messages outside the transaction thus the messages are moving towards their inputs independently.

The laws that describe dynamic interactions between processes are defined by reduction relation which definition follows.

**Definition 4.1.3** The reduction relation \(\rightarrow\) is the least relation that satisfies the following axioms and rules:

\[
\pi_i(v) | \sum_i x_i(u_i).P_i \rightarrow P_i\{v/u\} \quad [\text{COM}]
\]
\[
\pi(x) | !x(u).P \rightarrow P\{v/u\} | !x(u).P \quad [\text{REP}]
\]
\[
\pi() | \langle |P; Q| \rangle_x \rightarrow Q \quad P \neq 0 \quad [\text{FAIL}]
\]
\[
\text{if } (x = x) \text{ then } P \text{ else } Q \rightarrow P \quad [\text{IFT}]
\]
\[
\text{if } (x = y) \text{ then } P \text{ else } Q \rightarrow Q \quad x \neq y \quad [\text{IFF}]
\]

[COM] and [REP] rules model input-output interaction and replication. [FAIL] models transaction failures – when a message on a transaction name is emitted, the body is terminated and the compensation is activated. Transaction aborts are not possible if the transaction is already terminated. [IFT] and [IFF] describe the behavior for conditional branching.
4.1.1 Expressing BPEL semantics

This section describes semantics of BPEL in terms of extended π-calculus. We only take a subset of whole BPEL – some basic and structured activities, and error handling as the primary issue as they can be expressed with parallel and synchronization π-calculus primitives. Because BPEL has not formally defined semantics it is complicated to compare its behavior with the one of π-calculus. A mapping is possible if it is limited to concurrency and synchronization mechanisms of BPEL.

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 by \( \tilde{u} \). The syntax of process \( A \) can be described as follows.

\[
A ::= \text{empty} & \quad \text{Empty} \\
| \text{invoke}(x, \tilde{i}, \tilde{o}) & \quad \text{Synchronous invoke} \\
| \text{invoke}(x, \tilde{i}) & \quad \text{Asynchronous invoke} \\
| \text{receive}(x, \tilde{i}) & \quad \text{Receive} \\
| \text{reply}(x, \tilde{o}) & \quad \text{Reply} \\
| \text{throw}(f, \tilde{o}) & \quad \text{Throw} \\
| \text{compensate}(z, \tilde{o}) & \quad \text{Compensate} \\
| \text{sequence}(A, A) & \quad \text{Sequence} \\
| \text{flow}(A, A) & \quad \text{Parallel flow} \\
| \text{switch}(x = y)A; A & \quad \text{Conditional} \\
| \text{pick}((x, \tilde{i}_1, A), (x, \tilde{i}_2, A)) & \quad \text{Alternative} \\
| \text{scope}_z(A, S_e, S_f, A) & \quad \text{Scope}
\]

This syntax is intuitive and represents an abstraction of the actual BPEL syntax. The scope construct deserves some consideration. The \( S_e \) and \( S_f \) are a finite set of triples of the form \((x, \tilde{u}, A)\) and \((f, \tilde{u}, A)\). That means that the event related to port name \( x \) or the exception \( f \) is handled by the activity \( A \) which receives \( \tilde{u} \) as parameters. The term \( \text{scope}_z(A, S_e, S_f, A) \) defines a scope with name \( z \), where \( A \) represents the process associated with the scope, \( S_e \) is the set of handled events, \( S_f \) is the set of handled exceptions, and \( A_c \) is the compensation activity associated with that scope.

The set \( A_{BPEL} \) contains only well-formed BPEL activities. To give the semantics in terms of π-calculus algebra a function \([\text{[]}]_{\bar{y}(\tilde{u})} : A_{BPEL} \to \text{Process} \) is used [9], which maps BPEL activities into π-calculus processes that execute the BPEL activity and flag out \( \bar{y}(\tilde{u}) \) to signal termination.
The encoding of some basic and structured BPEL activities into π-calculus algebra follows.

**a) Asynchronous invoke (one-way)**

The Web service is invoked by using a channel as service identifier to identify as specific operation of a certain service. Any operation implemented by a Web service can be considered associated to a channel, which is dedicated to receive invocations of the supplied operation. This behavior is translated as follows ($x$ is the end-point of the service):

$$[[\text{invoke}(x, \tilde{i})]]_{\bar{y}(\tilde{u})} = \overline{x \langle \tilde{i} \rangle} | \bar{y}(\tilde{u})$$

**b) Synchronous invoke (request-response)**

This kind of invocation needs two variables; $o$ is identified as the result of invocation. Session name $r$ is introduced to correlate the invocation with its response (the invoked service $S$ uses this channel to reply):

$$[[\text{invoke}(x, \tilde{i}, \tilde{o})]]_{\bar{y}(\tilde{u})} = r(\langle r, \tilde{o} \rangle) . \bar{y}(\tilde{u})$$

**c) Raising an exception**

It is combined 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 has to be terminated. The $\text{throw}(f, \tilde{o})$ is translated as follows:

$$[[\text{throw}(f, \tilde{o})]]_{\bar{y}(\tilde{u})} = r(\langle \overline{\text{throw}()} \rangle | \overline{f}(r, \tilde{o}) | \langle r \rangle . \bar{y}(\tilde{u}))$$

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

**d) Compensate**

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

$$[[\text{compensate}(z, \tilde{o})]]_{\bar{y}(\tilde{u})} = \overline{z \langle \tilde{o} \rangle} | \bar{y}(\tilde{u})$$

**e) Sequential composition**

Let the $(A, A')$ be the sequential composition of activity $A$ and $A'$; the encoding of such activity is as follows.

$$[[\text{sequence}(A', A'')]_{\bar{y}(\tilde{u})} = y'(\langle A' \rangle . \bar{y}(\tilde{u}). [[A'']_{\bar{y}(\tilde{u})}], \ \tilde{v} = \text{fn}(A'')$$

The sequencing of BPEL is not a binder operator like π-calculus. Variables scopes infer different semantic behavior with respect of usual binding of
process algebra. The problem is when sequentially combining two activities, which include some name input on common names. Thus the second activity cannot be just triggered when the first one terminates but passing additional information to bind the free names of the second activity is required.

f) Parallel composition
BPEL allows usage of links to express synchronization dependencies between activities. As noted by [9] it is redundant and the message passing should be used instead.

\[
[[[(\text{flow}(A', A''))]](\tilde{\nu})] = \gamma' y''([[[[A']])(\tilde{\nu})] | [[[A'']])(\tilde{\nu})] | y'(\tilde{\nu}).y''(\tilde{\nu}).\gamma(\tilde{\nu})
\]

g) Scopes
Scopes can be interpreted as the wrapping structure containing BPEL processes. Activities inside can be associated to particular mechanisms to handle asynchronous events and failures. Let \( A_z = \text{scope}_z(A, S_e, S_f, C) \) be a scope. When no fault happens it behaves as the main activity \( A \); 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. Events are handled concurrently with \( A \) while faults interrupt its execution.

In the case when an event \( \pi(\tilde{v}) \) occurs during the normal execution of a main activity \( A \), if a suitable handler \( (x, \tilde{\nu}, B) \in S_e \) is defined for some \( \tilde{\nu} \) and \( B \), the event is consumed and the scope behaves as the parallel composition (flow) of \( A_s \) and \( B\{\tilde{v}/\tilde{\nu}\} \). Here \( \tilde{\nu} \) represents the formal parameters of the handler. If during the execution of a scope an exception \( \text{throw}(f, \tilde{v}) \) occurs two different behaviors are possible, depending whether the current fault is handled or not. In the first case, when \( (f, \tilde{\nu}, B) \in S_f \) for some \( \tilde{\nu} \) and \( B \), the scope behaves as the fault handler \( B\{\tilde{v}/\tilde{\nu}\} \) where \( \tilde{\nu} \) are the names that represent the formal parameters. In the second case the scope behaves like the parallel composition of \( \text{throw}(f, \tilde{v}) \) (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 \( \text{compensate}(z, \tilde{o}) \) is performed, the scope \( A_z \) behaves as \( C\{\tilde{o}/\tilde{\nu}\} \) where \( \tilde{\nu} \) are names substituted by \( \tilde{o} \). If \( \text{compensate}(z, \tilde{o}) \) is performed when \( A \) has not completed, the scope behaves as the default handler compensating all the children scopes.

The encoding of scope in \( \pi \)-calculus can be described as composition by many threads running in parallel, some managing scope activities, some to event, fault, and compensation handlers. Let \( h_f \) and \( h_e \) be functions that, given a scope, return the set of handled faults and handled events:

\[
\begin{align*}
h_f(\text{scope}_z(A, S_e, S_f, C)) &= \{x | (x, \tilde{\nu}, B) \in S_f\} \\
h_e(\text{scope}_z(A, S_e, S_f, C)) &= \{x | (x, \tilde{\nu}, B) \in S_e\}
\end{align*}
\]
We can use a short form $h_f(S_f)$ and $h_e(S_e)$. We also define function $s_n$ that given $A \in A_{BPEL}$ returns the set of scope names contained in $A$.

**h) Event handler**

The definition consists of two parallel processes: the first catching events while the second handling them. 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:

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

$$en_{eh}().\left(\prod_{(x, \tilde{u}, A) \in S_e} !x(\tilde{u}), e_x(\tilde{u}); y_{eh}()\right)_{dies_{eh}} \prod_{(x, \tilde{u}, A_e) \in S_e} !e_x(\tilde{u}), [[[A_e]]]_{y'_{0}}$$

The signal $en_{eh}$ enables the overall process to become responsive to all the events included in $S_e$. The transaction is introduced to disable event handling: when the signal $dis_{eh}$ is raised, events are no longer considered and the termination signal $y_{eh}$ is performed. The actual event handling activities $A_e$ are outside the transaction what ensures that once disabled the whole handler, the activities that are still alive can complete their execution as predicted in BPEL specification. Event handlers are composed in parallel and are specified by the replicated input guarded processes waiting for an incoming message on an event related channel.

**i) Fault handler**

The encoding of this construct is similar to the event handler except for the fact, 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. Let $S_f$ be set of handled faults and $y_{fh}$ a name used to signal fault handler termination, the process describing the fault handling behavior is as follows:

$$FH(S_f, y_{fh}) = (y')$$

$$en_{fh}().\left(\sum_{(f, \tilde{u}, A) \in S_f} f(r, \tilde{u}).[[[A]]]_{y'_{0}} \overline{\text{throw}()} \ y'().(\overline{\text{throw}()} y_{fh}()); y_{fh}(); \overline{y_{fh}}(); \overline{y'_{0}})_{dis_{fh}}$$

Fault handler is encoded by alternative composition of input guarded processes waiting for a message on a relative channel. Any process associated to a fault behaves in a way that, when it terminates, the whole fault handler $FH$ also terminates. When the fault handling completes, the including scope and the activating $\text{throw}(f, \tilde{u})$ are notified. The first by an output on $y_{fh}$, the second by an output on $r$. Then they are terminated.

**j) Compensation handler**

This construct has to be programmed in a different way. It has to be enabled when the scope starts and installed at the end of the execution if no fault happened. 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 for the scope compensation is defined as follows:

\[
CH(A, C, z, y_{ch}) = en_{ch}(\langle z(\tilde{u}).(CC(A, \tilde{u}) | \text{throw } \langle \rangle ); (y')z(\tilde{u}).[[C]]y_{ch} | y_{ch} \rangle \rangle)_{inst_{ch}}
\]

\[
CC(A, \pi) = \prod_{z' \in sn(A)} z'(\pi)
\]

where \(CC(A, \tilde{u})\) is an auxiliary process used to compensate children scopes of \(A\). In the case the handler has been enabled but not yet installed, the behavior means the termination of the scope activity (\(\text{throw } \langle \rangle \)) and the compensation of the children scopes (\(CC(A, \tilde{u})\)). The second case where the compensation handler has been installed consists of the compensation process defined by designers.

Specifying the complete scope construct is composed of the event, fault and compensation handlers and by a process that represents the main activity of the scope. Some other open BPEL issues (e.g. speculative parallelism) can be defined in \(\pi\)-calculus algebra as well [9]. In this section we outlined the semantics of BPEL where some constructs has been formally defined with the main focus on BPEL error recovery framework. This framework includes event, fault and compensation handlers, which are programmed by exploiting the event-based mechanisms based on \(\pi\)-calculus language. One important property of such language is mobility: it is possible to transmit channel names that can be used by any process receiving them. This aspect is essential in the formalization of BPEL and plays an important role in the formalization of interaction with request-response services. Another approach of formalizing most of BPEL constructs (including fault handler) is with \(\mu\)-BPEL and Timed-Automata (TA) language [21] where some time-related properties can be verified.

### 4.2 Correlation

A basic characteristic of a business process is that its global task is divided into different sessions, called process instances, where each is responsible for carrying on a separate service or work for each client. Service instances must be stateful processes and for that purpose the variables are defined that carry XML data values, messages, and they also define the state of each service instance. Variables can also contain partner links, which are abstract references to other services; similar as in \(\pi\)-calculus where channels are used to exchange the names of channels [9].

Correlation sets are introduced to identify interactions that are related to a
given process instance, which is necessary for correct order of dispatched messages between the various concurrent sessions. Each correlation set contains the properties that are aliases for parts of messages and are late-bound constants – once initialized its content will never change. At the given time each process instance has a number of active correlation sets, and the values associated to all their properties uniquely identify the process instance. As an example, all the messages related to a given client request must have the same unique identifier (e.g., client ID).

4.2.1 Case study

As a reference case study we consider a simple example that demonstrates how incoming messages are delivered to the correct process instance by the BPEL engine using correlation mechanisms.

The example deals with correlation sets. The BPEL process provides two operations: `setValue` and `getValue`. The string value passed to the `setValue` operation is returned when the `getValue` operation is invoked. The call to `setValue` creates a new instance of the BPEL process, which is identified by the identifier passed in with the request message. This request message is stored with the process instance until the `getValue` operation is invoked.

To ensure that the request message for the `getValue` operation is directed to the correct process instance, the same identifier value must be used in the request message. The operation `setValue` can be invoked multiple times each time with different identifier to create multiple process instances. The `getValue` operation can be invoked using different identifiers to retrieve the values passed in by each of the set operations.

Only one partner link is specified representing the customer service with two operations `setValue` and `getValue`. Four types of exchange messages are used: `setValueRequest` that is composed of an identifier and a value part, `setValueResponse` with an identifier part, `getValueRequest` that has a single identifier part, and `getValueResponse`, which has a single return value part.

To use the correlation sets a property identifier is declared that represents the value for correlating request messages with process instances. Next two properties alias elements are declared, one for each message type (`setValueRequest` or `getValueRequest`), which specify how the value is extracted from the message. These elements specify the message type, the part, and the XPath [31] query to use. Four variables are used in BPEL process for storing request messages (`setValueRequestMessage`, `getValueRequestMessage`) and response messages (`setValueResponseMessage`, `getValueResponseMessage`).

One correlation set with name `instanceID` is defined that contains the property identifier. This correlation set is then referenced from within the two
receive activities. The reference to the correlation set on the first receive has an attribute `initiate="yes"` which indicates that the correlation set is initialized with the value from the incoming message. This value remains unchanged for the lifetime of the process instance. The reference to the correlation set on the second receive activity ensures that only messages with a matching value for the identifier property are passed to that process instance. If a request message for `getValue` operation does not contain an identifier value that matches one of the active processes, it is rejected.

The logic of the BPEL process can be described in the following pseudo code:

```pseudo
Receive setValue request /* create process instance & initializes correlation set */
Reply with request identifier
Receive getValue request /* with matching correlation set value */
Initialize getValue Reply
Reply to getValue
```

The whole BPEL process is defined as follows. Some XML code is omitted for the sake of brevity.

```xml
<?xml version="1.0" ?>
<bpws:process name="CorrelationSample" targetNamespace="http://tempuri.org/CorrelationSample"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://demo.com/Correlation.xsd1"
xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
xmlns:ns="http://tempuri.org/Correlation.wsdl"
xmlns:ns1="http://tempuri.org/Correlation.xsd1"
xmlns:tns="http://tempuri.org/Correlation.wsdl">
<plnk:partnerLinkType name="Correlation">
  <plnk:role name="ServiceProvider">
    <plnk:portType name="tns:Correlation"/>
  </plnk:role>
</plnk:partnerLinkType>
<bpws:property name="identifier" type="xsd:string"/>
<bpws:propertyAlias messageType="tns:setValueRequest" part="request"
  propertyName="tns:identifier" query="/xsd1:SetValue/Identifier"/>
<bpws:propertyAlias messageType="tns:getValueRequest" part="request"
  propertyName="tns:identifier" query="/xsd1:GetValue/Identifier"/>
<bpws:partnerLinks>
  <bpws:partnerLink myRole="ServiceProvider" name="client" partnerLinkType="ns:Correlation"/>
</bpws:partnerLinks>
<bpws:variables>
  <bpws:variable messageType="ns:setValueRequest" name="setValueRequestMessage"/>
  <bpws:variable messageType="ns:setValueResponse" name="setValueResponseMessage"/>
  <bpws:variable messageType="ns:getValueRequest" name="getValueRequestMessage"/>
  <bpws:variable messageType="ns:getValueResponse" name="getValueResponseMessage"/>
</bpws:variables>
<bpws:correlationSets>
  <bpws:correlationSet name="instanceID" properties="ns:identifier"/>
</bpws:correlationSets>
```
To express basic BPEL correlation constructs a process language has to be introduced. We rely on language defined by [25], which is consisted of mapping BPEL syntax to abstract language syntax to provide an operational semantics that describe the actual behavior and state of a business process at a given
time. [24] presents a similar but a more powerful formal process language construction and correlation analysis.

Let \( u \) be a meta-variable ranging over integer and Boolean values, \( v \) over variables of all values, and \( p \) over properties that form correlation sets. Meta-variable \( l \) ranges over Web services end-points that can be conceptually described as triples of partner link, an action name, and a direction; they identify an interaction channel in a unique way. Given meta-variable \( x \), \( \overline{v} \) indicate sequences of \( x \) elements such as \( \langle x_1, x_2, x_3, \ldots, x_k \rangle \), \( x_i \) is the \( i \)-th element of one such sequence, \( x \in \overline{v} \) is used to indicate that \( x = x_i \) for some \( i \), and the symbol \( \circ \) is used for empty sequence. The syntax of the language is as follows.

\[
\begin{align*}
  w & ::= v \mid p & \text{Variables and properties} \\
  e & ::= \text{any} \mid u \mid v \mid f(\overline{v}) & \text{Expressions} \\
  s & ::= 0 & \text{Empty specification} \\
  & \mid \text{send}(l, v, w) & \text{Send a message} \\
  & \mid \text{recv}(l, v, w) & \text{Receive a message} \\
  & \mid \text{while}(e)\{a\} & \text{Iteration} \\
  & \mid \text{switch}(e)\{a : a\} & \text{Switching structure} \\
  & \mid \text{assign}(v, e) & \text{Variable assignement} \\
  & \mid \text{source}(\lambda) \mid \text{target}(\lambda) & \text{Internal links operation} \\
  & \mid a; a \mid (a \parallel a) \mid a + a & \text{Sequence, flow, and choice} \\
  a & ::= s & \text{Activity}
\end{align*}
\]

Meta-variable \( w \) ranges over both variables and properties; their assignment to a value defines the current state of a given process instance. Expressions can be the \textit{any} identifier, which may have any value in non-deterministic way.

With the syntax language definition is quite simple to map BPEL XML elements into the abstract syntax [25]. The part, which needs more accurate description, is the management of correlations. Activities \textit{send} and \textit{recv} have extra argument \( \overline{w} \). Abstract syntax represents messages as sequence of values; thus \( \overline{v} \) indicates variable holding messages, and \( v_i \) indicates the corresponding variable over its \( i \)-th subpart or field. As example the one-way invoke activity without correlation

\[
<\text{invoke} ... \text{inputVariable="varName"} ... /></\text{invoke}>
\]

is expressed as \textit{send}(\( l, \overline{v}_{\text{varName}}, \overline{v}_{\text{varName}} \)). The first parameter is the message receiver end-point, the second is the variables that will contain the received message, and the third contains information on correlations; when no correlations exists this parameter is exactly the same as the second.

Let the SOAP message has type \textit{msg}, which is composed of three parts \textit{part1},
part2, and part3, it is denoted by variable \( \tau = \langle v_1, v_2, v_3 \rangle \) and let us suppose that the following correlation set is specified inside the invoke activity:

\[
<\text{invoke ...}>...<\text{correlations}> <\text{correlation set}="cName" ... > </\text{correlations}> </\text{invoke}>
\]

By the definitions of correlation sets

\[
<\text{correlationSet name}="cname" \text{ properties}="p1 p2"> \\
<\text{bpws:propertyAlias propertyName}="p1" \text{ messageType}="msg" \text{ part}="part1" ...> \\
<\text{bpws:propertyAlias propertyName}="p2" \text{ messageType}="msg" \text{ part}="part2" ...>
\]

the name \( \text{cname} \) represents a correlation set made of the properties \( p1 \) and \( p2 \), which are linked to the first and second part of message \( \text{msg} \). The correlation set denotes those process instances that exchange messages with given values in the first and second part of the \( \text{msg} \) messages. The proper abstract syntax is now \( \text{send}(l, \langle v_1, v_2, v_3 \rangle, \langle p_1, p_2, v_3 \rangle) \). The third parameter \( \langle p_1, p_2, v_3 \rangle \) indicates that while the third part of the message is not related to any correlation set, it is exactly the variable \( v_3 \), the first two parts are properties \( p_1 \) and \( p_2 \) and are correlated to the process instance. If in the current process instance \( p_1 \) is bound to value \( u_1 \) and \( p_2 \) to \( u_2 \), then the first and second part of the message are automatically set to \( u_1 \) and \( u_2 \). In the opposite case the \( \text{recv}(l, \langle v_1, v_2, v_3 \rangle, \langle p_1, p_2, v_3 \rangle) \) only a message carrying \( u_1 \) and \( u_2 \) in the first and second part can be received by the process instance.

The sample BPEL process defined in Section 4.2.1 can be expressed in the abstract syntax. Four endpoints are used: \( l_s \) represents an action \text{setValue} invoked by the client, \( l_s' \) is used to send a response to the client after the invocation of \text{setValue}, \( l_g \) represents an action \text{getValue} invoked by the client, and \( l_g' \) is used to send a response after invocation of \text{getValue}. The symbol \( \tau^s \) is used for a variable \text{setValueRequestMessage}, containing a request message, which is composed by two parts \( v^s_1 \) and \( v^s_2 \) representing Identifier and Value. The symbol \( \tau^s' \) is used for a variable \text{setValueResponseMessage} containing a part \( v^s_1' \) representing Identifier. Then, the symbol \( \tau^g \) represents a variable \text{getValueRequestMessage}, containing a part \( v^g_1 \) for Identifier, and the symbol \( \tau^g' \) is used for a variable \text{getValueResponseMessage} with a part \( v^g_1' \) representing Value. A property \( p \) of the correlation set is used for Identifier. The abstract syntax of the example is directly mapped from the BPEL specification as follows:

\[
\text{recv}(l_s, \tau^s, \langle p, v^s_2 \rangle); \\
\text{assign}(v^s_1', v^s_1); \\
\text{send}(l_s', \tau^s', \tau^s'); \\
\text{recv}(l_g, \tau^g, \langle p \rangle); \\
\text{assign}(v^g_1', v^g_1); \\
\text{send}(l_g', \tau^g', \tau^g')
\]
This activity sequence is more compact as actual BPEL specification.

4.2.3 Operational semantics

To provide an operational semantics describing the actual behavior of orchestrating Web services, the presented abstract syntax is needed to represent the state of a business process at the given time. Abstract syntax has to be extended with run-time aspects. The symbol for activity $a$ is extended as [25]:

$$a ::= 0 | (\overline{w} \rightarrow \overline{u})s | !a | a \parallel_P a$$

Notation $(\overline{w} \rightarrow \overline{u})$ is called a store and it indicates that in the activity specification $s$, variable of property $w_i$ is associated to value $u_i$ for any $i$. Notation $(\overline{w} \rightarrow \overline{u})(\overline{w} \rightarrow \overline{u})$ has the same meaning as $(\overline{w}, \overline{w} \rightarrow \overline{u}, u)$. Symbol $!a$ indicates that the activity $a$ can be replicated (spawned) and then executed infinite times; the operator $\parallel_P$ is used for parallel composition of process instances while the operator $\parallel$ indicates parallel composition different activities inside the same process instance. While stores are used to model the current state of a process instance (in terms of variables and correlation sets), replication is used at the top level to deal with process instances. If $s$ is a BPEL activity, then $!(\circ \rightarrow \circ)s$ is the initial state of the business process that replicates a new process instance with initially empty store each time the first message of each work session is received.

To define operational semantics some congruence rules [24,25] (similar as in section 4.1.1.) have to be specified in abstract syntax as shown below:

$$!a \equiv !a \parallel_P a \ (\overline{w} \rightarrow \overline{u})0 \equiv 0 \quad [\text{CREP}]$$
$$a \parallel_P 0 \equiv a \quad a \parallel_P a' \equiv a' \parallel_P a \quad a \parallel_P (a' \parallel_P a'') \equiv (a \parallel_P a') \parallel_P a'' \quad [\text{CPAR}]$$
$$s \parallel 0 \equiv s \quad s \parallel s' \equiv s' \parallel s \quad (s' \parallel s'') \equiv (s \parallel s') \parallel s'' \quad [\text{CFLW}]$$
$$s + 0 \equiv s \quad s + s' \equiv s' + s \quad s + (s' + s'') \equiv (s + s') + s'' \quad [\text{CCHO}]$$
$$s; 0 \equiv s \quad s; s' \equiv s; (s'; s'') \equiv s; (s'; s'') \quad [\text{CSEQ}]$$

To differ variables from properties a partial operator $\oplus$ is introduced; it models the store update enforcing the idea that properties are constants: into a process the value of correlation set identifies the process identity that never changes. Operator is defined as follows:

$$(\overline{w} \rightarrow \overline{u}) \oplus (v \rightarrow u) \equiv (\overline{w} \rightarrow \overline{u})(v \rightarrow u)$$
$$(\overline{w} \rightarrow \overline{u})(p \rightarrow u) \oplus (p \rightarrow u) \equiv (\overline{w} \rightarrow \overline{u})(p \rightarrow u)$$
$$(\overline{w} \rightarrow \overline{u}) \oplus (p \rightarrow u) \equiv (\overline{w} \rightarrow \overline{u})(p \rightarrow u) \text{ if } p \notin \overline{w}$$

The first definition denotes that variable assignment always updates the store,
the second and third that updating property assignment are allowed only if an equivalent assignment already happens in the store. This definition is used to enforce the idea that messages sent and received by a given process instance must suit the correlation set; the notation \((p \to u) \oplus (p \to u')\) is meaningless when \(u \neq u'\). The notation for the stores is simplified as we write \((\overline{w} \to \overline{u}) \oplus (\overline{w'} \to \overline{u'})\) as a short form for \((\overline{w} \to \overline{u}) \oplus (\overline{w'_1} \to \overline{u'_1}) \oplus (\overline{w'_2} \to \overline{u'_2}) \oplus \cdots \oplus (\overline{w'_k} \to \overline{u'_k})\), where \(k\) is the size of \(\overline{w'}\) and \(\overline{w'}\).

Evaluation of expression \(e\) under the store \((\overline{w} \to \overline{u})\) if defined by the notation \(\{\overline{w}/\overline{u}\}e\) with the semantics [25]:

\[
\{\overline{w}/\overline{u}\} u \cong u \quad \{\overline{w}/\overline{u}\} w_i \cong u_i \quad \{\overline{w}/\overline{u}\} f(\overline{v}) \cong f(\{\overline{w}/\overline{u}\} \overline{v}) \\
\{\overline{w}/\overline{u}\} any \cong u \quad v \notin \overline{w} \Rightarrow \{\overline{w}/\overline{u}\} v \cong u
\]

As shown in order, values are evaluated to themselves, variables and properties to their associated values, functions by propagating the evaluation to their parameters, the \(any\) identifier or a variable not already initialized to any value \(u\). The notation for an evaluation is again simplified by using \(\{\overline{w}/\overline{u}\} \overline{v}\).

Operational semantics is defined by a labeled transition system \((A, \to, I)\), where \(A\) is the set of activities (states of modeling system), ranged over by meta-variable \(a\), and \(I\) is the set of labels representing interactions, ranged over by meta-variable \(\alpha\) with syntax:

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

Presented transition system is used to find out how the state of the business process evolves as interactions with the partners occur; notation \(a \stackrel{\alpha}{\longrightarrow} a'\) indicates an activity (state) \(a\) moving to \(a'\) by interaction \(\alpha\). \(\tau\) is the silent action that represents internal computation inside the business process; \(l!\overline{u}\) represents the business process sending a message to the end-point \(l\) specifying a message \(\overline{u}\), and similarly \(l?\overline{u}\) represents receiving a message from the end-point \(l\). Some rules defining transition relation \(\to\) are shown below.

\[
\begin{align*}
\text{PAR} & \quad a || p a'' \stackrel{\alpha}{\longrightarrow} a'' || p a' \quad \text{if} \quad a \stackrel{\alpha}{\longrightarrow} a'' \\
\text{SEQ} & \quad (\overline{w} \to \overline{u})(s_0; s_1) \stackrel{\alpha}{\longrightarrow} (\overline{w'} \to \overline{u'})(s'_0; s_1) \quad \text{if} \quad (\overline{w} \to \overline{u})s_0 \stackrel{\alpha}{\longrightarrow} (\overline{w'} \to \overline{u'})s'_0 \\
\text{ASG} & \quad (\overline{w} \to \overline{u})\text{assign}(v, e) \stackrel{\tau}{\longrightarrow} (\overline{w} \to \overline{u}) \oplus (v \to \{\overline{w}/\overline{u}\} e)0 \\
\text{REC} & \quad (\overline{w} \to \overline{u})\text{recv}(l, \overline{v}, \overline{w}_p) \stackrel{l\overline{w}_p}{\longrightarrow} (\overline{w} \to \overline{u}) \oplus (\overline{v} \to \overline{u'})(\overline{w}_p \to \overline{u'})0 \\
\text{SND} & \quad (\overline{w} \to \overline{u})\text{send}(l, \overline{v}, \overline{w}_p) \stackrel{l\{\overline{w}/\overline{u}\}\overline{v}}{\longrightarrow} (\overline{w} \to \overline{u}) \oplus (\overline{w}_p \to \{\overline{w}/\overline{u}\}\overline{v})0
\end{align*}
\]

For better understanding of operational semantics, we can use it to describe our correlation example (see Section 4.2.1) and its behavior. For a brevity, we denote the abstract specification shown in Section 4.2.2 by \(s_C\) composed by \(s_s\) and \(s_g\).
\[ s_C := \text{recv}(l_s, \overline{v}^s, \langle p, v_2^s \rangle); s_{sa}; s_{ss}; s_{gr}; s_{gs} \]

\[ s_{sa} := \text{assign}(v_1^s, v_1^s) \]

\[ s_{ss} := \text{send}(l_s', \overline{v}'^s, \overline{v}'^s) \]

\[ s_{gr} := \text{recv}(l_g, \overline{v}^g, \langle p \rangle) \]

\[ s_{ga} := \text{assign}(v_1^g, v_2^g) \]

\[ s_{gs} := \text{send}(l_g', \overline{v}'^g, \overline{v}'^g) \]

As an example we consider the following evolution:

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

- At some time, a client sends a request for \texttt{setValue} operation of the kind \( \langle \text{proc}_1, \text{val}_1 \rangle \) asking to set value \( \text{val}_1 \):

  \[
  a_0 \xrightarrow{l_s? \langle \text{proc}_1, \text{val}_1 \rangle} a_0 \parallel p \ a_1
  \]

  \[
  a_1 = (\overline{v}^s \rightarrow (\text{proc}_1, \text{val}_1))(p \rightarrow \text{proc}_1)s_{sa}; s_{ss}; s_{gr}; s_{ga}; s_{gs}
  \]

  This is obtained first by replicating-spawning a new process instance by congruence rule \[ \text{CREP} \] that receives the request by operational rule \[ \text{REC} \], which is executed in the context of \[ \text{SEQ} \]. The result of this execution is that the store will contain the variable \( v^s \) for the received message, while the correlation set property \( p \) defining the identifier of the process instance is bound to \( \text{proc}_1 \).

- A new client sends the request \( \langle \text{proc}_2, \text{val}_2 \rangle \) asking to set value \( \text{val}_2 \); a new process instance is created similarly to the previous case:

  \[
  a_0 \parallel p \ a_1 \xrightarrow{l_s? \langle \text{proc}_2, \text{val}_2 \rangle} a_0 \parallel p \ a_1 \parallel p \ a_2
  \]

  \[
  a_2 = (\overline{v}^s \rightarrow (\text{proc}_2, \text{val}_2))(p \rightarrow \text{proc}_2)s_{sa}; s_{ss}; s_{gr}; s_{ga}; s_{gs}
  \]

  Two process instances \( a_1 \) and \( a_2 \) are currently active, each with own state of the store denoted by different \texttt{instanceID}.

- The execution of both process instances stops at the \( s_{gr} \) activity that waits for \texttt{getValue} request message. The evolution of the process \( a_1 \) is the follow-
ing (it is similar for $a_2$):

$$a_0 \parallel p \ a_1 \xrightarrow{\tau}$$

$$a_0 \parallel p \ (\mathbf{v}^s \rightarrow \langle \text{proc}_1, \text{val}_1 \rangle)(p \rightarrow \text{proc}_1)\text{assign}(v_1'^{\prime}, v_1')$$

$$\text{send}(l_g', \mathbf{v}^{s'}, \mathbf{v}^s); s_{gr}; s_{gs} \xrightarrow{\tau}$$

$$a_0 \parallel p \ (\mathbf{v}^s \rightarrow \langle \text{proc}_1, \text{val}_1 \rangle)(p \rightarrow \text{proc}_1)(v_1'^{\prime} \rightarrow \text{proc}_1)$$

$$\text{send}(l_g', \mathbf{v}^{s'}, \mathbf{v}^s); s_{gr}; s_{gs} \xrightarrow{l_g'!(\text{proc}_1)}$$

$$a_0 \parallel p \ (\mathbf{v}^s \rightarrow \langle \text{proc}_1, \text{val}_1 \rangle)(p \rightarrow \text{proc}_1)(v_1'^{\prime} \rightarrow \text{proc}_1)s_{gr}; s_{ga}; s_{gs}$$

The process instance waits now for the request message. The incoming message is necessarily associated to the proper process instance due to the fact that it is allowed only within the process instance providing compatible correlation sets.

- The client sends the request for `getValue` operation in the form $\langle \text{proc}_1 \rangle$ asking to get proper stored value from the appropriate process instance $a_1$ identified by the identifier $\text{proc}_1$. The evolution of the process state moves forward:

$$a_0 \parallel p \ (\mathbf{v}^s \rightarrow \langle \text{proc}_1, \text{val}_1 \rangle)(p \rightarrow \text{proc}_1)(v_1'^{\prime} \rightarrow \text{proc}_1)\text{recv}(l_g, \mathbf{v}_g, \langle p \rangle)$$

$$s_{ga}; s_{gs} \xrightarrow{l_g?!(\text{proc}_1)}$$

$$a_0 \parallel p \ (\mathbf{v}^s \rightarrow \langle \text{proc}_1, \text{val}_1 \rangle)(p \rightarrow \text{proc}_1)(v_1'^{\prime} \rightarrow \text{proc}_1)(\mathbf{v}_g \rightarrow \langle \text{proc}_1 \rangle)$$

$$\text{assign}(v_1'^{\prime}, v_2'); s_{gs} \xrightarrow{\tau}$$

$$a_0 \parallel p \ (\mathbf{v}^s \rightarrow \langle \text{proc}_1, \text{val}_1 \rangle)(p \rightarrow \text{proc}_1)(v_1'^{\prime} \rightarrow \text{proc}_1)(\mathbf{v}_g \rightarrow \langle \text{proc}_1 \rangle)$$

$$(v_1'^{\prime} \rightarrow \text{val}_1)\text{send}(l_g', \mathbf{v}^{s'}, \mathbf{v}^s); l_g'!(\text{val}_1) \xrightarrow{\tau} a_0$$

When the client receives the response value $\text{val}_1$, the process instance $a_1$ terminates because it has no further activities to perform. The same evolution would take place for process instance $a_2$. If a request message for a `getValue` operation arrives, which does not contain an identifier value (e.g. different from $\text{proc}_1, \text{proc}_2$) that matches one of the active process instances, the error occurs and the request is rejected by the BPEL server.

The presented example shows a fragment how to formalize BPEL process language and express it with proper operational semantics with the concurrent programming languages of well known process algebraic constructs. This allows defining a core formal language for BPEL orchestration language. The correlation mechanisms are modeled in a process algebra, which provides some heterogeneous features and precise modeling an important portion of BPEL.
4.3 Control flow

The kernel of BPEL consists of simple communication primitives that may be combined using control-flow constructs expressing sequence, branching, parallelism, synchronization, concurrency, etc. Analysis of these dynamic aspects and their properties is not easy without formalization of BPEL control flow. The constructs found in BPEL, especially those related to control flow, are close to those found in workflow definition languages. In the area of workflow, it has been shown that Petri nets [4,18,6] provide an appropriate foundation for static analysis and verification. Thus the analysis can be performed on BPEL processes by translating them to Petri nets and apply existing Petri net-based analysis techniques and tools.

There are many issues that can be analyzed with Petri nets – dead locks, detection of unreachable activities, reachability of specific states, path coverage, liveness, soundness, and reductions of BPEL workflow nets. Several approaches are presented how to translate BPEL constructs to Petri nets (High level Petri nets with colored tokens [4], Petri net based algebra [6], plain Petri transition/place nets [18]) that are used as an intermediate representation where different theoretical techniques and tools can be applied (WofBPEL [18], pose++ [4]).

4.3.1 Petri nets

A Petri net (also known as a place/transition net or P/T net) is one of several mathematical representations of discrete distributed systems. A Petri net is a directed, connected, and bipartite graph in which each node is either a place or a transition [6]. A directed arc connects place with transition. Tokens occupy places. Transition is enabled when there is at least one token in every place connected to a transition. Any enabled transition can fire removing one token from every input place, and depositing one token in each output place.

Therefore, a Web service is basically a partially ordered set of operations. It is quite straightforward to map it into a Petri net. Operations are modeled by transitions and the state of the service is modeled by places. The directed arcs between places and transitions are used to specify causal relations. A Petri net that represents the behavior of a Web service contains one input place for absorbing information (with no incoming arcs) and one output place for emitting information (no outgoing arcs). Such Petri net is a good model [6] to manage definition of the composition operators and the analysis as the verification of certain properties (reachability, deadlocks, liveness). We can define formal representation of Petri net and Web service as a foundation for definition of the formal semantics.
Definition 4.3.1 A service net is a labeled place/transition Petri net represented by a tuple $SN = (P, T, W, i, o, ℓ)$ where:

- $P$ is finite set of places,
- $T$ is finite set of transitions that represents the operations of the service,
- $W \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs (flow relation),
- $i$ is the input place with $\bullet i = \{ x \in P \cup T \mid (x, i) \in W \} = \emptyset$,
- $o$ is the output place with $\bullet o = \{ x \in P \cup T \mid (o, x) \in W \} = \emptyset$,
- $ℓ : T \to A \cup \{ \tau \}$ is labeling function where $A$ is a set of operation names; $\tau$ denotes a silent operation, $\tau /\in A$.

Silent operations are transition firings that cannot be observed and they are used to distinguish between external and internal behavior of the service. The formal definition of a Web service follows.

Definition 4.3.2 A Web service is a tuple $S = (Name, Desc, Loc, URL, CS, SN)$ where:

- $Name$ is the name of the service, used as its unique identifier,
- $Desc$ is description of the service (what the service offers),
- $Loc$ is the location of the server where the service is located in,
- $URL$ is the address for the invocation of the Web service,
- $CS$ is a set of its component services; if $CS = \{ Name \}$ then $S$ is the basic service otherwise is a composite service,
- $SN = (P, T, W, i, o, ℓ)$ is the service net modeling the dynamic behavior of the Web service.

The place $i$ is the initial marking of the service $S$ (only $i$ contains a token). The execution of service $S$ starts when a token is in the place $i$ and finishes when a token reaches the place $o$.

4.3.2 Modeling Web services

Web service has a specific task and may depend on other Web services. Composition of two or more services generates a new service providing the original individual behavioral logic and a new collaborative behavior to accomplish a new composite task.

The algebra is presented that allows the creation of new Web services using existing ones as building blocks. The set of Web services can be defined by
the basic grammar [6] in BNF notation:

\[
S ::= \epsilon \quad \text{Empty service} \\
| X \quad \text{Service constant} \\
| S \otimes S \quad \text{Sequence} \\
| S \oplus S \quad \text{Alternative (choice)} \\
| S \diamond S \quad \text{Unordered sequence} \\
| \mu S \quad \text{Iteration} \\
| S \parallel S \quad \text{Parallelism} \\
| S \parallel c S \quad \text{Parallelism with communication}
\]

The algebra guarantees that each result of an operation on services is a service (closure property) to which algebra operators can be applied again. This can be used to build more complex services by aggregating and reusing existing services through declarative expressions of the service algebra. It is important to note that service composition applies to syntactically different services. Places and transitions of the composed services must be disjoint for proper composition. Let \(S_i = (\text{Name}_i, \text{Desc}_i, \text{Loc}_i, \text{URL}_i, \text{CS}_i, \text{SN}_i)\) with \(\text{SN}_i = (\text{P}_i, \text{T}_i, \text{W}_i, \text{i}_i, \text{o}_i, \text{ℓ}_i)\) for \(i = 1 \ldots n\) be \(n\) Web services such that \(\text{P}_i \cap \text{P}_j = \emptyset\) and \(\text{T}_i \cap \text{T}_j = \emptyset\) for \(i \neq j\).

**a) Sequence**

This operator allows the execution of two services \(S_1\) and \(S_2\). \(S_1\) must complete before \(S_2\) can start. This is typically the case when a service depends on the output of the previous service.

**Definition:** The service \(S_1 \otimes S_2\) is defined as a tuple \((\text{Name}_1, \text{Desc}_1, \text{Loc}_1, \text{URL}_1, \text{CS}_1, \text{SN}_1)\) where:

- \(\text{Name}\) is the name of the new service,
- \(\text{Desc}\) is description of the new service,
- \(\text{Loc}\) is the location of the new service,
- \(\text{URL}\) is the address for the invocation of the new service,
- \(\text{CS} = \text{CS}_1 \cup \text{CS}_2\),
- \(\text{SN} = (P, T, W, i, o, ℓ)\) where:
  - \(P = P_1 \cup P_2\),
  - \(T = T_1 \cup T_2 \cup \{t\}\),
  - \(W = W_1 \cup W_2 \cup \{(o_1, t), (t, i_2)\}\),
  - \(i = i_1\),
  - \(o = o_2\),
  - \(ℓ = ℓ_1 \cup ℓ_2 \cup \{t, τ\}\).

Graphical representation by the Petri net is shown in Fig. 1.
b) Alternative
The alternative operator allows with given two services \( S_1 \) and \( S_2 \) to model the execution of either \( S_1 \) or \( S_2 \), but not both.

**Definition**: The service \( S_1 \oplus S_2 \) is defined as a tuple \((Name, Desc, Loc, URL, CS, SN)\) where:

- Name is the name of the new service,
- Desc is description of the new service,
- Loc is the location of the new service,
- URL is the address for the invocation of the new service,
- \( CS = CS_1 \cup CS_2 \),
- \( SN = (P, T, W, i, o, \ell) \) where:
  - \( P = P_1 \cup P_2 \cup \{i, o\} \),
  - \( T = T_1 \cup T_2 \cup \{t_{i_1}, t_{i_2}, t_{o_1}, t_{o_2}\} \),
  - \( W = W_1 \cup W_2 \cup \{(i, t_{i_1}), (i, t_{i_2}), (t_{i_1}, i_1), (t_{i_2}, i_2), (o_1, t_{o_1}), (o_2, t_{o_2}), (t_{o_1}, o), (t_{o_2}, o)\} \),
  - \( \ell = \ell_1 \cup \ell_2 \cup \{(t_{i_1}, \tau), (t_{i_2}, \tau), (t_{o_1}, \tau), (t_{o_2}, \tau)\} \).

Graphical representation by the Petri net is shown in Fig. 2.

c) Parallelism with communication
The parallel operator allows the concurrent execution of two or more services. Concurrent services can synchronize and exchange information (e.g., BPEL control links).
Definition: Let $C = \{ (\alpha, \beta) \mid (\alpha, \beta) \in T_1 \times T_2 \cup T_2 \times T_1 \}$ be a set of communication elements. The new service $S_1 \parallel_c S_2$ is defined as a tuple $(Name, Desc, Loc, URL, CS, SN)$ where:

- **Name** is the name of the new service,
- **Desc** is description of the new service,
- **Loc** is the location of the new service,
- **URL** is the address for the invocation of the new service,
- **CS** = $CS_1 \cup CS_2$,
- **SN** = $(P, T, W, i, o, \ell)$ where:
  - $P = P_1 \cup P_2 \cup \{i, o\} \cup \{p_i \mid (\alpha_i, \beta_i) \in C\}$,
  - $T = T_1 \cup T_2 \cup \{t, t_o\}$,
  - $W = W_1 \cup W_2 \cup \{(i, t_i), (t, i_1), (t, i_2), (o_1, t_o), (o_2, t_o), (t_o, o)\} \cup \{(\alpha_i, p_i), (p_i, \beta_i) \mid (\alpha_i, \beta_i) \in C\}$,
  - $\ell = \ell_1 \cup \ell_2 \cup \{(t, \tau), (t_o, \tau)\}$.

Given $S_1$ and $S_2$ and $C = \{ (\alpha, \beta), (\lambda, \alpha) \}$, $S_1 \parallel_c S_2$ is graphically represented by the Petri net shown in Fig. 3.

It is worth mentioning that other similar mapping models to Petri net exist. [4] present a high-level Petri net (HPN) with guard expression and colored tokens. Translation of BPEL process into HPN consists of translating an operation (atomic operation level translation) where input and output messages are decomposed based on their data type. Next step is generation of HPNs, which denote relationships on operations such as one input message of one operation, is one output message of another operation (operation cluster translation). Final step is combining HPNs produced by the previous steps into one HPN according to activity relationships defined in BPEL specification. More exhaustive approach is presented by [18]. All of the BPEL constructs (activities, control links, event handlers, fault handling, compensation, and termination) are mapped to plain Petri nets. Labeled transitions are used to model basic activities and events. The labels encode meta-data, which is associated with
the names of message types sent or received. The unlabeled $\tau$-transitions rep-
resent internal actions that cannot be observed by external participant. All of
the mapping approaches to Petri nets allow the usage of existing theoret-
elical results and Petri net analysis techniques (analysis of dynamic properties
related to unreachable activities, conflicting messages, deadlocks, reduction
techniques).

4.3.3 Sample process

We briefly present an example of a process for handling a purchase order that
can be mapped to Petri net using mapping rules described in the previous
section. On receiving the purchase order from a customer, the process initi-
ates three tasks concurrently: a calculation of the final price for the order, a
selection of a shipper, and a schedule of the production with a shipment for
the order. While some of the processing can be performed concurrently, there
are control and data dependencies between the three tasks. In particular, the
shipping price is required to finalize the price calculation, and the shipping
date is required for the complete fulfillment schedule. When the three tasks
are completed, invoice processing can be performed and the invoice is sent to
the customer.

The purchase order process can be described in a BPEL language – some
elements (partner links, fault handlers) are omitted for the sake of simplicity
and clarity. BPEL definition follows.

```xml
<process name="purchaseOrderProcess"
   targetNamespace="http://company.com/ws-bpel/purchaseSample"
   xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/
   xmlns:lns="http://demo.org/wsdl/purchase">
  <!-- partner links definition -->
  <variables>
    <variable name="PO" messageType="lns:POMessage" />
    <variable name="Invoice" messageType="lns:InvMessage" />
    <variable name="POFault" messageType="lns:orderFaultType" />
    <variable name="shippingRequest" messageType="lns:shippingRequestMessage" />
    <variable name="shippingInfo" messageType="lns:shippingInfoMessage" />
    <variable name="shippingSchedule" messageType="lns:scheduleMessage" />
  </variables>
  <!-- fault handlers definition -->
  <sequence>
    <receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="PO" createInstance="yes" />
  </flow>
  <links>
    <link name="ship-invoice" />
    <link name="ship-scheduling" />
  </links>
  <sequence>
```
The process contains three separate sequences that can be executed concurrently: shipping, invoicing, and scheduling. Three basic services with operations are used in the particular sequence:

- shippingService with operation requestShipping,
- invoiceService with operations initiatePriceCalculation, and sendShippingPrice,
- schedulingService with operations requestProductionScheduling, and sendShippingSchedule.

The process itself is exposed as purchaseService with a single operation sendPurchaseOrder. Two additional operations are used to receive callback messages during the asynchronous message exchange: sendSchedule, and sendInvoice. The set of communication element is \( C = \{(sendShippingPrice, requestShipping), (sendShippingSchedule, sendSchedule)\} \). We use parallelism with communication construct to model regular Petri net. The new composite service purchaseService representing business process is defined as shipping \( \parallel \) invoicing \( \parallel \) scheduling = (Name, Desc, Loc, URL, CS, SN):

- **Name** = “Purchase order service”,
- **Desc** = “Process for handling a purchase order”,
- **Loc** = “location 1”,
- **URL** = “http://tempuri.org/purchase/sendPurchaseOrder”,
- **CS** = \{invoiceService, shippingService, schedulingService\},
- **SN** = \((P, T, W, i, o, \ell)\) where:

  - \( P = \{i, t_1, Pship_1, Pship_2, o_1, t_2, Pinv_1, Pinv_2, o_2, i_3, Psch_1, o_3, o\} \)
  - \( T = \{t_1, \tau, requestShipping, sendSchedule, initiatePriceCalculation, sendShippingPrice, sendInvoice, requestProductScheduling, sendShippingSchedule, t_o\} \)
  - \( W = \{(i, t_1), (t_1, i), (i, \tau), (\tau, Pship_1), (Pship_1, requestShipping), (requestShipping, Pship_2), (Pship_2, sendSchedule), (sendSchedule, o_1), (o_1, t_o), (t_1, i_2), (i_2, initiatePriceCalculation), (initiatePriceCalculation, Pinv_1), (Pinv_1, sendShippingPrice), (sendShippingPrice, Pinv_2), (Pinv_2, sendInvoice), (sendInvoice, o_2), (o_2, t_o), (t_o, i_3), (i_3, requestProductScheduling), (requestProductScheduling, Psch_1), (Psch_1, sendShippingSchedule), (sendShippingSchedule, o_3), (o_3, t_o), (sendShippingPrice, p_1), (p_1, requestShipping), (sendShippingSchedule, p_2), (p_2, sendSchedule)\} \)
  - \( \ell = \{\tau, requestShipping, sendSchedule, initiatePriceCalculation, sendShippingPrice, sendInvoice, requestProductScheduling, sendShippingSchedule\} \)

Graphical representation of the purchase order process by the Petri net is shown in Fig. 4.

The result of mapping can be used to perform formal verification and analysis of BPEL processes on the basis of the existing Petri net analysis techniques (reachability, deadlock detection, concurrency, coverage, reduction, and optimization). Thus the BPEL processes can be verified and validated before deploying them into production environments. Several tools for automated analysis of Petri net exist: BPEL2NML, WofBPEL [18], pose++ [4].
5 Conclusion

This paper gives a survey of BPEL orchestration and WS-CDL choreography languages with formal representation of some selected aspects. BPEL and WS-CDL are two important standards for modeling and implementing workflows and business processes based on Web services. BPEL follows the orchestration paradigm that is more flexible approach compared to the choreography. It is becoming de-facto standard for business process modeling and is gaining increasing adoption as a process oriented service composition language. Choreography, which specifies a business process in terms of interactions among the participating parties from a global point of view, has not gained support from the industry comparing to BPEL. The orchestration paradigm follows a central coordinator, which coordinates the execution of different operations on the involved parties.

As BPEL has no defined formal semantics to analyze the behavior of business process this paper focuses on some important orchestration aspects that can be analyzed with different formal methods. First we concentrate on error, event, and compensation handling mechanisms that are formally described in the extended version of $\pi$-calculus algebra. The second important issue of BPEL is the correlation mechanism that allows to relate operations of the same sessions – this is achieved through correlation sets. The correlation mechanism is expressed using a process algebra that consists of abstract syntax and opera-
tional semantics. The algebra allows studying some properties and features of
the correlation. Last important issue of BPEL that is considered in this pa-
paper is a control flow. BPEL consists of simple communication primitives that
can be combined using control-flow constructs to express sequence, branching,
parallelism, synchronization, concurrency, etc. As Petri nets provide an appro-
priate foundation for static analysis and verification of control flow constructs,
the Petri net-based algebra is described and a translation approach into the
algebra from BPEL constructs.

We have presented a subset of possible formal methods, which are the most
suitable for selected orchestration aspects. Our future work could present some
other powerful formal models and methods – Timed Automata [22] for the
timed analysis, Finite State Automata [4] for the behavior specification and
concurrency, Communicating Sequential Processes (CSP) [33] for the behavior
verification of collaboration Web services. Other interesting issues are inves-
tigation of ontologies between process objects and elements [23], integration
of Web services security and trust mechanisms into the existing BPEL and
WS-CDL business process languages.

References

Version 1.1. 5 May 2003. BEA Systems, IBM, Microsoft, SAP AG,
software/dw/specs/ws-bpel/ws-bpel.pdf.


[3] Bilorusets R et al. Web Services Reliable Messaging Protocol (WS-
ReliableMessaging). BEA System, IBM, Microsoft, TIBCO Software, 2005. URL:

High-level Petri Nets. Proceedings of the 10th IEEE International Enterprise
Distributed Object Computing Conference (EDOC ’06). p. 441-444. IEEE
Computer Society; 2006.

Specification: From BPEL to C-BPEL. Electronic Notes in Theoretical Computer

Proceedings of the 14th Australasian database conference (ADC ’03)–Volume 17.


