An Analysis of Web Services Workflow Patterns in Collaxa

Martin Vasko and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Wien, Austria
{e0025379@stud3.tuwien.ac.at | dustdar@infosys.tuwien.ac.at}

Abstract. Web services have a substantial impact on today's distributed software systems, especially on the way they are designed and composed. Specialization of different services is leading to a multitude of applications ultimately providing complex solutions. The interaction and modeling aspects of Web services is increasingly becoming important. Based on the needs for Web services conversations, process modeling, and composition, a variety of languages and technologies for Web services composition have evolved. This case study is focused on a systematic evaluation of the support for workflow patterns and their BPEL (Business Process Execution Language for Web Services) implementation in Collaxa, a leading BPEL process modeling and enactment engine for Web services processes.

Keywords: Workflow patterns, composition, BPEL

1 Introduction

The Business Process Execution Language for Web services (BPEL) [1] is an XML-based flow language that defines how business processes interact within or between organizations. The initial BPEL 1.0 specification was jointly proposed by IBM, Microsoft, BEA in August, 2002 and updated in May 2003 by version 1.1. It supports compensation-based business transactions as defined by the WS-Transaction specification. Business Processes specified in BPEL are fully portable between BPEL-compliant environments. BPEL is a block-structured programming language, allowing recursive blocks, but restricting definitions and declarations to the top level. The language defines *activities* as the basic building elements of a process definition. *Structured activities* prescribe the order in which a collection of activities take place. Ordinary sequential control between activities is provided by sequence, switch, and while. Concurrency and synchronization between activities is provided by the flow constructor. Nondeterministic choice based on external events is provided by the pick constructor. It contains handlers for events including message events (onMessage with portType, operation and partner) and timed events such as duration or deadline. Process instance-relevant data (containers) can be referred to in routing logic and expressions. BPEL defines a mechanism for catching and handling faults similar to common programming languages, like Java. One of the key aspects of Service Oriented Architectures is the support of dynamic finding and binding of services at

runtime (e.g. in a repository such as UDDI). However, in BPEL the notion of dynamic finding and binding is not supported directly. These activities need to be modeled explicitly (as activities, i.e. Web services).
Furthermore, BPEL allows describing relationships (third party declaration) how services interact (what they offer) by introducing *Partner*[1] *Link Types* (PLNK), with a collection of roles, where each role indicates a list of portTypes. At runtime, the BPEL runtime (execution) engine has to deal with the binding. BPEL also supports the notion of compensation and fault handling. Both concepts are based on the concept of *scopes* (i.e. units of compensation or fault). BPEL creates process instances implicitly, i.e. whenever instances receive a message, an instance is created. This is different to many workflow systems, which identify process instances by their ID. In the case of BPEL any “key field”, such as an invoice number in an order fulfillment scenario, could be used for this purpose. The BPEL middleware has to deal with the issue of finding the suitable instance (or creating one if required). This mechanism is called *message correlation*.

The remainder of the paper is structured as follows. Section 2 introduces an example for a process model consisting of Web services. Section 3 analyses the Workflow patterns as suggested in [5] and the implementation found in Collaxa. Finally, section 4 provides remarks and summarizes the workflow pattern support.

2 A Supply Chain process model

The workflow patterns analyzed in this paper are discussed using a model of a Supply Chain process. The service consists of three different processes, each of them communicating with each other. The first process is a logistics process shown in Figure 1. It starts with an order placement as input, which can be generated through an interface for Web services. In the next step, the customer data is validated through a synchronous call to an external service such as a customer database. After user data processing, the decision between direct assembly and distributed processing has to be done. In case of direct assembly the shipment can follow as the next step. If the assembly has to be done before shipment, the process is stalled until all parts have arrived and have been compiled. Otherwise, the compilation can be done after shipment at the final destination. If this is the case, the service is stalled after shipment, until all components have arrived. The logistics control is responsible for the decision where the product has to be built. This brings us to the second process which is active in the Supply Chain process model.

The logistics control process receives input data from the logistics process. Based upon the total costs, calculated for the three different possibilities of assembly (before, after shipment or no direct assembly) it decides, which decision tree has to be followed. After execution of this sequence, the result of the best solution has to be found. It will be returned to the calling process and is from now on the headed execution target.

[1] was called Service Link Type in BPEL4WS 1.0

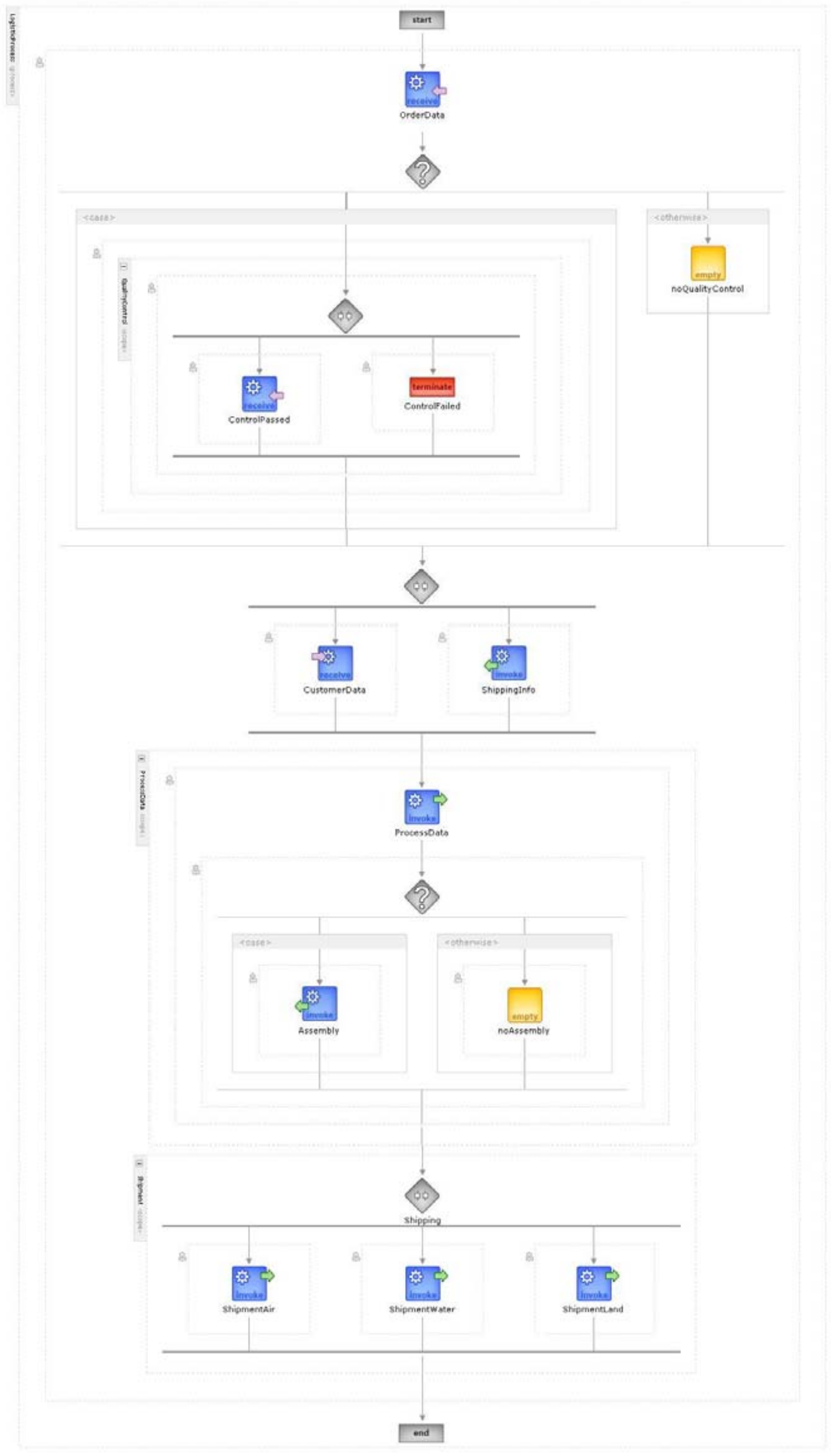
start
receive
OrderData
<case>
<otherwise>
empty
noQualityControl
QualityControl
receive
ControlPassed
terminate
ControlFailed
receive
CustomerData
invoke
ShippingInfo
ProcessData
invoke
ProcessData
<case>
<otherwise>
invoke
Assembly
empty
noAssembly
Shipment
Shipping
invoke
ShipmentAir
invoke
ShipmentWater
invoke
ShipmentLand
end

Fig. 1

3 Workflow patterns

In the previous section an abstract view of a supply chain process was given. Now we take a closer look at workflow patterns that are constructed with basic BPEL elements. The pattern definitions were taken from van der Aalst et al. [5] in order to provide a framework for their systematic evaluation. Focusing on the support for these workflow patterns by Collaxa, we realised the majority of control structures in the logistics process model. More complex workflow patterns are presented in BPEL source code only. The provided figures visualize the explained patterns. To understand the structure of these figures the basic elements are described as follows:
A service is referenced by a simple box. To visualize business flow, connected lines and arrows are used. The used notation is following the Activity Diagrams.

Workflow pattern: Sequence

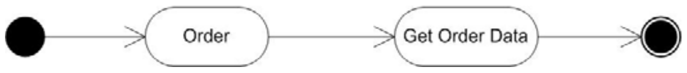

Fig. 2

Two or more activities are processed in a workflow process in the order, in which they were defined. It does not support any kind of parallelism. The Collaxa BPEL engine realizes this with the concepts inherited from XLANG. Listing 1 provides a short excerpt from the source code.

```
Listing 1
<sequence name=”main”>
                <receive name="receiveOrder" partnerLink="client"
portType="tns:SupplyChain" operation="initiate" variable="input"
createInstance="yes"/>
                <receive partnerLink="client" portType="tns:SupplyChain"
name="CustomerData"/><scope>
</sequence>
```

In this example source code, there are two receive statements. The first gets the order data from an external Web service, and the second receives the customer data for verification.

Workflow pattern: Parallel split

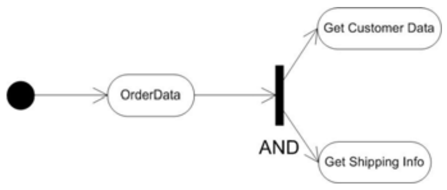

Fig. 3

This pattern defines the structure of a process which is split into several threads of control, all executed in parallel. The order in which they are processed is not defined. This pattern is provided by Collaxa by defining the flow activity, as described in the previous section. The source code which implements this structure is comparable to Listing 1 except, that the sequence statements are enclosed in a flow statement, as described in Listing 2.

```
Listing 2
<flow>
<sequence><receive partnerLink="client" portType="tns:SupplyChain"
name="CustomerData"/></sequence>
<sequence><invoke name="ShippingInfo" partnerLink="client"
portType="tns:SupplyChainCallback" inputVariable="output"/>
</sequence>
</flow>
```

The source code in Listing 2 contains the flow statement, which allows a parallel execution of the two sequences of control. The structure enclosed in the sequences is executed one after another.

Workflow pattern: Synchronization

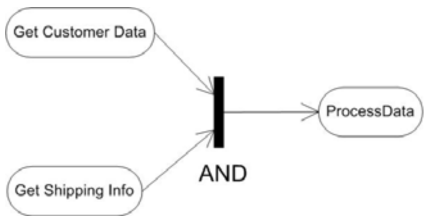

Fig. 4

This pattern is implemented mostly by the use of receive statements. The execution of the process continues when all data concerning the customer is collected. After that, the processing of the data will be started. In the actual example, the data for shipment and customers are collected. If both are finished, the ‘normal’ flow of the business process commences. This structure is shown in Listing 3. To show the synchronizing structure of this example, a scope is starting immediately after the flow statement, concerning the previous pattern to be executed.

```
Listing 3
<flow>
        <sequence><receive name="CustomerData"/></sequence>
        <sequence><invoke name="ShippingInfo"/></sequence>
</flow></sequence>
<scope name="ProcessData">
        <sequence><invoke name="ProcessData"/>
```

Workflow pattern: Exclusive choice

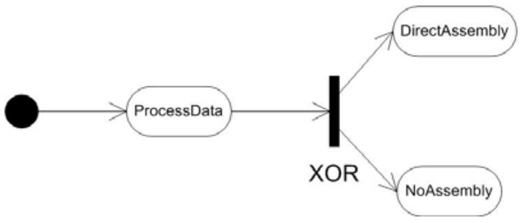

Fig. 5

The exclusive choice structure defines a point in the business workflow, where a certain condition based on a decision in the flow is taken. This workflow pattern is best implemented with the switch statement of BPEL. In Listing 4 the execution of a switch statement is shown. This BPEL code fragment decides between ‘direct assembly’ or ‘no assembly’.

```
Listing 4
<switch>
        <case>
        <sequence><invoke partnerLink="client"
portType="tns:SupplyChainCallback" name="Assembly"/></sequence>
        </case>
        <otherwise>
        <sequence><empty name="noAssembly"/></sequence>
        </otherwise>
</switch>
```

Workflow pattern: Simple Merge

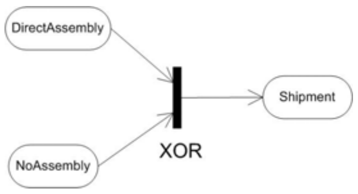

Fig. 6

This pattern defines a point in the flow of execution, where two or more alternative branches come together. It is important to mention that the simple merge pattern does not support any kind of synchronization, which means, that none of the alternative processes is ever executed in parallel. This pattern is only supported in an indirect way by Collaxa.

```
Listing 5
<sequence name="assembly">
        <switch>
        <case><sequence><invoke partnerLink="client"
portType="tns:SupplyChainCallback" name="Assembly"/></sequence></case>
        <otherwise><sequence><empty name="noAssembly"/></sequence></otherwise>
        </switch>
</sequence>
<invoke name="Shipment"/></sequence>
```

In Listing 5 a simple merge pattern is provided. Notice, that this is not the only possibility to realize this functionality. The sequences contain one invoke statement and an empty statement. The decision is made by the logistics control, but in both cases, the alternative sequence is never chosen.

Workflow pattern: Multi Choice

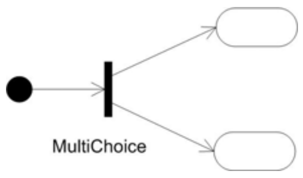

Fig. 7

In contrast to the exclusive choice this pattern defines a point in the workflow, where a number of branches can be chosen. It is *not supported* by the Collaxa BPEL engine. Furthermore, it is rather complicated to achieve a similar pattern.

Workflow pattern: Synchronizing Merge

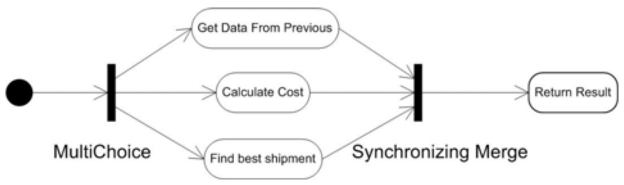

Fig. 8

This pattern marks a point in the process execution, where several branches merge into a single one. If one or more processes are active, the flow is triggered until these processes are finished.

Workflow pattern: Multi – Merge

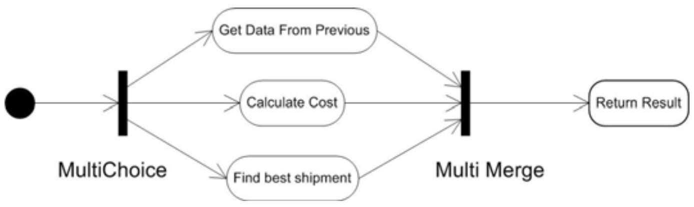

Fig. 9

This pattern joins two or more different workflows without synchronization together. This means that results, processed on different paths, are passed to other activities in the order in which they are received. This pattern is neither supported by XLANG, nor by WSFL. Listing 6 provides a mixture between the two workflow patterns synchronizing - merge and multi – merge. The two sequences are always executed in parallel. The whole logic is enclosed by a loop statement. The pattern which is realized by this code depends on the problem to solve. It may have a synchronizing behavior, or alternatively, a multi merge one.

```
Listing 6
<switch>
<case><sequence>
        <flow>
                <sequence><invoke name="Get Data From Previous"/></sequence>
                <sequence><invoke name="Calculate Cost"/></sequence>
        </flow></sequence></case>
<otherwise><invoke name="Find best shipment"/></otherwise>
</switch>
<reply name="Return Result"/>
```

Workflow pattern: Discriminator

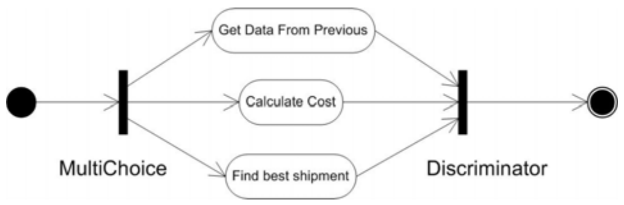

Fig. 10

This pattern describes a point in the execution flow of the system which waits for completion of an incoming branch, before executing a subsequent flow of control. As the subsequent process is activated, all other incoming branches are ignored. When all incoming branches are triggered, this structure resets itself to accept new incoming processes. This workflow pattern is comparable with the previous pattern. Note, that this structure is not supported by the BPEL language. Moreover, there does not exist a structured activity which can provide a workaround. A simple join condition in an OR condition is not suitable, because of the evaluation of the results. In this condition, always both or more results are evaluated before the execution path continues to the following activity. This restriction of BPEL makes it impossible for the Collaxa Engine to support this pattern, neither direct, nor as a workaround.

Workflow pattern: Arbitrary Cycles

This pattern defines a point in the business process, where a portion of the process has to be "visited" repeatedly. There have to be no restrictions on the number, location, and nesting of these points. This pattern is not supported in BPEL. Although the while statement supports simple loop structures, it is not possible to jump into the loop flow in an arbitrary way. As in the previous pattern, the Collaxa engine does not support this pattern. This may be a possible extension for future BPEL specifications.

Workflow pattern: Implicit Termination

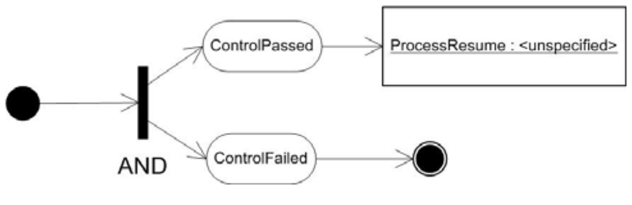

Fig. 11

An executed sub process is terminated, when there is nothing left to do. In BPEL the flow statement realizes this pattern. This activity awaits the occurrence of one set of events.

```
Listing 7
<scope name="QualityControl">
<sequence>
        <flow>
        <sequence><receive name="ControlPassed"/></sequence>
        <sequence><terminate name="ControlFailed"/></sequence>
        </flow>
</sequence>
</scope>
```

The sample code in Listing 7 demonstrates the QualityControl process. It is enclosed by a flow statement, which provides the parallel functionality. Depending on an external decision by a quality controller, the process flow can be approved or denied.

Workflow pattern: Multiple Instances without Synchronization

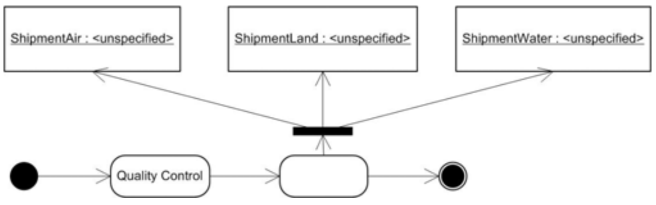
ShipmentAir : <unspecified>
ShipmentLand : <unspecified>
ShipmentWater : <unspecified>
Quality Control

Fig. 12

Multiple Instance (MI) without synchronization defines the creation of multiple instances within an activity, all of them independent of each other. In addition, they may be able to execute in parallel.

```
Listing 8
<while condition=”terminate”>
        <invoke name=”ShipmentAir”></invoke>
        <invoke name=”ShipmentWater”></invoke>
        <invoke name=”ShipmentLand”></invoke>
</while>
<receive name=”ShipmentAir” createInstance=”yes”></receive>
<receive name=”ShipmentWater” createInstance=”yes”></receive>
<receive name=”ShipmentLand” createInstance=”yes”></receive>
```

The code in Listing 8 invokes multiple instances of shipments in the while loop. To create an instance, each time the process is invoked, the `createInstance` attribute has to be set to “yes”. Notice, that this BPEL code is a theoretical approach and does not correspond to the visual representation in figure 1. It is an alternative solution, to better demonstrate this workflow pattern.

Workflow pattern: Multiple Instances with Synchronization

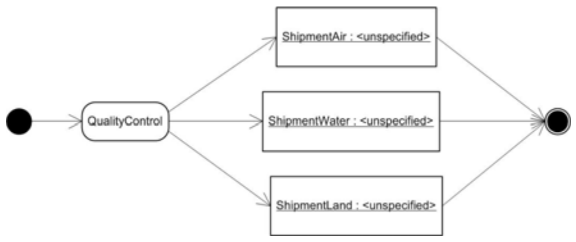
QualityControl
ShipmentAir : <unspecified>
ShipmentWater : <unspecified>
ShipmentLand : <unspecified>

Fig. 13

In contrast to the previous pattern, MI with synchronization means, that all instances, which are created in the scope of an action, are synchronized before they proceed with the workflow. This pattern is separated into several different structures, which differ from each other in the way the processes are instantiated. The first structure holds information about the number of instances at design time. In the second workflow pattern, the number of created processes is known at some stage of run time, but before the instantiation of the processes. The last pattern concerning MI with Synchronization does not know the number of instances to be created. New processes are created as they are needed, as long as no more instances are required. If the number of instances is known at compile time, the synchronization is comparable to the previous pattern, except for the need of an enclosing flow statement, which handles the synchronization.

```
Listing 9
<flow name="Shipping">
        <sequence><invoke name="ShipmentAir"/></sequence>
        <sequence><invoke name="ShipmentWater"/></sequence>
        <sequence><invoke name="ShipmentLand"/></sequence>
</flow>
```

The BPEL code sample in Listing 9 explains a possible solution for multiple instances, where the number of instances is known at run time. The three instances of Shipments are created and, enclosed within the flow statement, are synchronized.

Workflow pattern: Deferred Choice

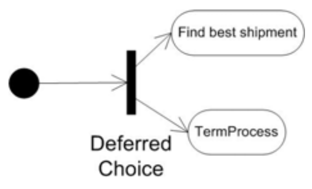

Fig. 14

In contrast to the Multi Choice pattern, this pattern chooses the branch to execute based on an event which is not necessarily available when the branch is reached. The decision, which branch to take is delayed until the suitable event has occurred. This construct is supported by the `<pick>` statement and is used especially for external triggers. We added a pick pattern to the logistics control application, as shown in Listing 10.

```
Listing 10
<sequence>
<pick>
<onMessage><invoke name="Find best shipment"/></onMessage>
<onAlarm><terminate name="TermProcess"/></onAlarm>
</pick>
</sequence>
```

This code extends the logistics control process shipment function. It is dynamically terminated by the terminate statement when a suitable solution was found. Sometimes it is not possible to find the best shipment in linear time, so we have to terminate the flow of control alternatively by a maximum amount of time, which is available for processing. During the consumption of the process the solution is approximated iteratively.

Workflow pattern: Interleaved Parallel Routing

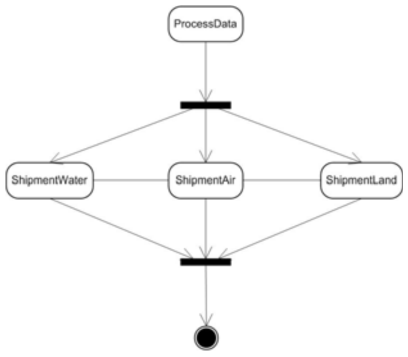

Fig. 15

This workflow pattern defines a point in execution, where a set of activities is processed in a determined order. Each activity is executed only once and the order, in which they are processed, is defined at run-time. This pattern is realized in Collaxa with the `variableAccessSerializable` attribute in the `<scope>` statement. If this attribute is set to true, variable access in this scope is subject to concurrency control. In Listing 11 a BPEL code excerpt from the logistic process is shown.

```
Listing 11
<scope name="Shipment" variableAccessSerializable="yes">
<switch>
        <case><invoke name="ShipmentAir"/></case>
        <case><invoke name="ShipmentWater"/></case>
        <otherwise><invoke name="ShipmentLand"/></otherwise>
</switch>
</scope>
```

Listing 11 contains the process about the physical shipment for the parcels. They will be alternatively routed over sea, land or air. The visual representation in Figure 1 contains the flow statement enclosing the three different shipments to explain multiple instances with synchronization as explained in Figure 14.

Workflow pattern: Milestone

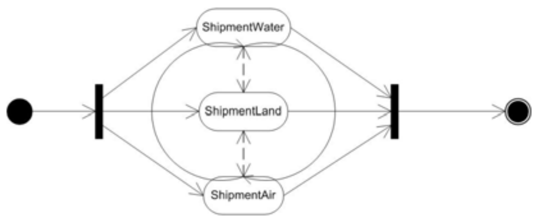

Fig. 16

The Milestone workflow pattern defines a point in the workflow, where a determined milestone has to be reached to enable a given activity. A milestone can also expire which means that the activity will not be enabled. BPEL does not support this pattern directly. BPELJ [4] provides a possible solution for implementing a ‘workaround’. In Listing 12, a solution for this pattern is provided.

```
Listing 12
<bpelj:propertyAlias propertyName=”Token” type=”bpelj:com.supply.Milestone”/>
   <bpelj:extractValue arg=”milestone”>
   milestone.getValue()
   </bpelj:extractValue>
</bpelj:propertyAlias>
<flow name="Shipping">
        <sequence><invoke name="ShipmentAir"/></sequence>
        <sequence><invoke name="ShipmentWater"/></sequence>
        <sequence><invoke name="ShipmentLand"/>
        <input part=”token” variable=”milestone”/>
               <correlations>
         <correlation set=”Milestone” initite=”yes” pattern=”out” parts=”token”>
        </correlations>
        </sequence>
</flow>
```

Listing 12 provides an example for a milestone pattern. In this code, ShipmentLand provides a token, which can be accessed externally by all other processes. If the milestone was achieved, execution will be advanced; otherwise the flow is stalled, until the Milestone will be reached by ShipmentLand. The only thing that is not standard BPEL in this example is the part attribute specified in the correlation element. In future BPEL notations, we expect the concept of *opaque correlations* whose values are chosen by the execution framework. This approach is not realized in Collaxa. Maybe it is subject for future work.

Workflow pattern: Canceling

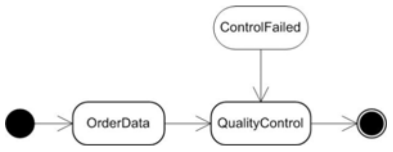

Fig. 17

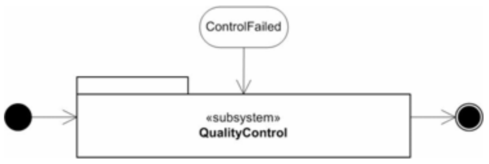

Fig. 18

This pattern is divided into Cancel activity (Figure 17) and Cancel case (Figure 18). The first one terminates a running instance of an activity and the second one leads to the removal of an entire instance.

```
Listing 13
<scope name="QualityControl">
<sequence>
<flow>
        <sequence><receive name="ControlPassed"/></sequence>
        <sequence><terminate name="ControlFailed"/></sequence>
</flow>
</sequence>
</scope>
```

The code in Listing 13 aborts the execution of the process with the terminate action.

4 Conclusion

In this paper we analyzed and systematically evaluated the support for workflow patterns in the Collaxa BPEL engine as defined in [5]. The majority of workflow patterns are supported by this system. Some of them are not even realizable in BPEL, but there are possibilities for workarounds which compensates this detriment. Some statements have restrictions which are known and expressed in the vendors Developer Resource. For example the `<invoke>` statement does not currently support local exception handling. To summarize the usability and design of the Collaxa server is straightforward. In contrast to other implementations the visual BPEL Designer makes the development of workflows easy compared to writing BPEL code directly. However, there does not exist a standard or "agreed upon" visual notation for BPEL. The preferred implementation architectures (JBoss[6] for Collaxa server and Eclipse[7] for BPEL Designer) rank as one of the most prominent of all related open source projects. Table 1 presents a summary of the discussed workflow patterns and their support in Collaxa. An 'X' in the cell indicates if (a) support of the current workflow pattern is given, (b) Collaxa provides possibilities for a workaround or (c) it is not possible to implement this pattern.

Workflow pattern	*Direct support*	*Workaround*	*not supported*
Sequence	X		
Parallel split	X		
Synchronization	X		
Exclusive choice	X		
Simple Merge	X		
Multi choice		X	
Synchronizing Merge		X	
Multi merge			X
Discriminator			X
Arbitrary cycles			X
Implicit termination	X		
MI without synchronization	X		
MI with synchronization	X		
Deferred choice	X		
Interleaved parallel routing	X		
Milestone		X	
Canceling	X		

Table 1. Workflow pattern support in Collaxa

Acknowledgements

This research was supported in part by a research award for *Development Methods for Dynamic Web services Workflows* of the Chamber of Commerce Vienna (Wirtschaftskammer Wien).

5 References

1. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services. http://dev2dev.bea.com/techtrack/BPEL.jsp
2. http://www.collaxa.com
3. http://www.collaxa.com/pdf/cx-bpel-developer-20.pdf
4. Michael Blow, Yaron Goland, Matthias Kloppman et al. *BPELJ: BPEL for Java*, A Joint White Paper by BEA and IBM
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. *Workflow Patterns*, Distributed and Parallel Databases, 14, 5-51, 2003, Kluwer.
6. http://www.jboss.org
7. http://www.eclipse.org