Institute of Architecture of Application Systems

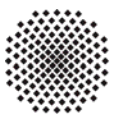
On Visualizing and Modelling BPEL with BPMN

David Schumm, Dimka Karastoyanova, Frank Leymann, Jörg Nitzsche

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{schumm, karastoyanova, leymann, nitzsche}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{SchummKLN09,
  author    = {David Schumm and Dimka Karastoyanova and Frank Leymann and
               Jörg Nitzsche},
  title     = {On Visualizing and Modelling BPEL with BPMN},
  booktitle = {IEEE Proceedings of the 4th International Workshop on
               Workflow Management, IWWM2009,
               4-8 May 2009, Geneva, Switzerland},
  year      = {2009},
  pages     = {80--87},
  doi       = {10.1109/GPC.2009.12},
  publisher = {IEEE Computer Society}
}
```

Universität Stuttgart
Germany

On Visualizing and Modelling BPEL with BPMN

David Schumm, Dimka Karastoyanova, Frank Leymann, Jörg Nitzsche
University of Stuttgart
{david.schumm, dimka.karastoyanova, frank.leymann, joerg.nitzsche} @iaas.uni-stuttgart.de
http://www.iaas.uni-stuttgart.de/

Abstract

The advantages of the process-based approach to implementing applications lead to the development of notations for modelling business processes and languages for enacting them in a process engine for the purpose of process automation. Currently the Business Process Modeling Notation (BPMN) is typically used for modelling business processes and the Business Process Execution Language (BPEL) is used as the process execution format. Both languages differ in purpose, expressivity and operational semantics. Recently it has been shown that there is no complete bi-directional mapping between BPMN and BPEL and transformations have been defined between the two formalisms. However, these transformations lead to more complex models in both, BPEL and BPMN, and enable a roundtrip for only a limited number of scenarios. In this paper we show how BPEL processes can be modelled using the graphical aspect of BPMN in order to facilitate modelling of executable processes using BPMN while avoiding model transformations.

1. Introduction

BPMN [5] aims at creating a standardized bridge for the gap between the business process design and the process implementation. Present research and tooling however show that this aim has not yet been achieved. In the area of process implementation the language BPEL [6] is of special interest, as it is the standard for process implementation in service-oriented architectures. When dealing with automated transformation from BPMN to BPEL one might lose the ability to manage, understand and monitor the generated code and its execution, as the current transformation techniques produce ”less readable code” [7]. We observe a number of problems: when the full spectrum of BPMN is allowed for process modelling without any restrictions, a transformation can result in a GoTo-like code structure in BPEL, as BPMN supports the usage of arbitrary cycles. A further unsolved issue is the missing support for a process lifecycle management that addresses the usually required IT-refinement of the BPEL code that is generated out of a BPMN diagram. The IT-refinement might necessitate a subsequent roundtrip transformation from manipulated BPEL code back to BPMN in order to ensure the preservation of already conducted work. This roundtrip transformation should result in a synchronized process model similar to the original one, yet the available transformation techniques are not convertible. Another problematic aspect of a BPMN to BPEL transformation is that it is hard to give a proof, whether or not the employed transformation technique is correct, because of vague specification of the execution semantics of some icons like the ’Inclusive OR-Gateway’.

BPEL on the other hand has clearly specified execution semantics but the specification does not deal with the graphical representation. This results in numerous tools for modelling BPEL processes, each with a distinct graphical representation. Hence a process modelled in BPMN and the visual representation of a BPEL process are hard to compare.

In this paper we present an approach that allows the visualization and modelling of BPEL processes using BPMN as graphical notation, without requiring any explicit model transformation and thus avoiding the problems mentioned above. We argue that by defining a mapping from BPEL constructs to graphical BPMN elements it is possible to visualize a BPEL process using BPMN diagrams. However, the modelling of BPEL processes with BPMN ”look and feel” has some restrictions as described later in the paper.

This paper is structured as follows. Background information is provided in Sect. 2. In Sect. 3, related

work is discussed. Our approach for the visualization and modelling of BPEL with BPMN is presented in Sect. 4. In Sect. 5 a prototypic implementation of our approach is shown. Finally, conclusions and outlook are provided in Sect. 6.

2. Background

The *Business Process Modeling Notation (BPMN)* is meant for use by business users who create the initial drafts of processes, technical developers who are responsible for implementing processes and business people who manage and monitor processes [5]. This notation is deemed appropriate to communicate a business process to a huge audience. However, BPMN lacks clearly defined execution semantics and related work shows that BPMN and BPEL have a different expressivity [11] and that a transformation from BPMN to BPEL is only possible for a limited number of scenarios and produces unmanageable code in certain cases [7]. The freedom of drawing arbitrary process graphs provided by BPMN often results in process models that are not transformable into executable models. For this reason the vendors offering BPM solutions that support BPMN for process modelling usually only support a subset of BPMN in order to be able to produce executable code for the process execution. In its current version (1.1) BPMN is not suitable for creating an executable process, unless a few restrictions on the language are imposed, as discussed in [10]. Otherwise the modelled processes have to be partially transformed and refined manually by human beings that again makes the lifecycle of evolving processes hard to manage.

The *Business Process Execution Language for Web Services (BPEL)* enables the definition of business processes as coordinated sets of Web Service interactions (Orchestration) recursively into new aggregated Web Services. Both, Intra Enterprise Integration and Integration with Business Partners can be achieved by orchestrating applications, using BPEL processes. BPEL has clearly specified execution semantics and strong support in the industry so that it is the de-facto language in its area, but due to the fact that BPEL is a XML-based language the process models in this format tend to be quite 'verbose' and not easily readable by human beings. In order to leverage the creation and maintenance of BPEL processes the tool vendors have created graphical process modelling tools for BPEL, but as the definition of a graphical notation for BPEL is out of scope of the specification, each vendor uses its own graphical notation. This hinders the communication between technical developers and business analysts, which is often referred to as Business-IT-Gap, but it also impedes the communication among technical developers that use tools from different vendors. The usage of BPMN as graphical notation for BPEL processes could narrow this gap.

The language BPEL and the notation BPMN have a *conceptual mismatch* in their underlying concepts. A means to compare the expressiveness of computer languages that are related to processes and their implementation in IT are Workflow Patterns [11]. The comparison of the language expressiveness of BPEL and BPMN [9] using Workflow Patterns shows some of the most important differences. For instance BPMN allows to draw arbitrary cycles, and also language constructs for modelling an unsynchronized merge of parallel paths and global data semantics. Another fundamental difference that can not be captured by the Workflow Pattern analysis is the underspecification of the execution semantics of BPMN, though there is recent work in this area [1]. Also the mismatching concept of token flow in BPMN and control flow in BPEL makes a mapping challenging. In some works the mismatch is reduced to the block structure in BPEL compared to the graph structure in BPMN. This is of minor importance, as the <flow> construct and the usage of <link> elements provide for graph-like structures [14], yet with the limitation that the creation of cycles is not allowed in order to make Dead-Path-Elimination [3] work.

3. Related Work

Existing work mainly addresses the transformation of BPMN diagrams into BPEL processes in order to make them executable. Most notably in this area is the work of Ouyang et al. [7] where the transformation is accomplished by multiple phases, matching of patterns and graph transformations. The mapping is still incomplete, problems in this transformation result from the possible usage of arbitrary cycles, unsynchronized merge of parallel paths and the semantics of the Inclusive OR-gateway. Thus only a subset of the language constructs is allowed in order to be able to transform a BPMN diagram into BPEL code completely. The works that address the transformation from BPEL to BPMN either do not cover the whole language [8, 13] or do not allow for round tripping [12]. A product in this area is [4], which provides a BPMN visualization, yet without taking the graph structure of BPEL into account (<flow> construct with <links>) to define control dependency.

4. Visualizing and Modelling BPEL with BPMN

The basic idea of our approach is to make use of graphical representations instead of transformation techniques for creating BPEL processes. We will show that it is possible to visualize and model a BPEL process using the icons, connecting elements and semantics defined in BPMN. The basic principle is to use BPEL as the metamodel for the process definition. For each construct in BPEL a mapping to its graphical representation in BPMN is defined, based on the BPMN specification and the transformation patterns provided in [7]. Model transformations are not applied in this approach, instead the graphical representation directly reflects the underlying BPEL code. This approach imposes a few restrictions on BPMN, as only those BPMN icons can be used for modelling which have a counterpart in the BPEL metamodel. It is also not possible to freely place some constructs on the modelling space, for example Gateways can only be used in combination, such as split/join or fork/merge. Those restrictions also have to be imposed even when model transformation techniques are employed, in order to get manageable code. So from this point of view this limitation is similar to those in the existing approaches.

Three fundamental principles guide the algorithm for the visualization. The first one is to always produce a closed system, so any split is finally being joined, any fork is finally being merged and any start is transitively connected to an end, in order to account for the concept of token flow. The second principle is to make use of a recursive definition for the representation of activities. From the implementation point of view that means the function for the visualization of an activity can call other visualization functions in order to build up and integrate the graphical representation of nested activities. For example the visualization function of an <if> activity will call the visualization function of a <sequence> activity, if a <sequence> is contained within the <if> that is visualized. The third principle is to stick as close as possible to the semantics described in the BPMN specification and its interpretation in the BPMN-to-BPEL patterns by Ouyang et al. [7] in order to have a meaningful and clear graphical visualization of the code and its execution semantics.

Overall it is possible for almost every BPEL activity or other construct to define a mapping to a set of BPMN icons. The mapping for basic activities is shown in table 1, the mapping of structured activities is described in table 2, and the mapping of other BPEL constructs is presented in table 3. Some minor problems arise when the requirement of closed systems should always be met, as for the <exit>, the <throw> and the <rethrow> activity no corresponding Intermediate Events exist in BPMN. However a correct mapping is also possible using End Events.

The approach is best explained using several examples. For space limitations we only present the mapping of a simple structured activity (<if>) in section 4.1, the structured <scope> activity is presented in section 4.2 and the <flow> activity is discussed in section 4.3 in detail.

```
<if>
   <condition>if_condition</condition>
   <receive name="Receive"/>
   <elseif>
      <condition>elseif_condition</condition>
      <reply name="Reply"/>
   </elseif>
   <else>
      <assign .../>
   </else>
</if>
```

Listing 1. Code of an <if> activity in BPEL

Table 1. Mapping of BPEL Basic Activities

BPEL Construct	BPMN Mapping
<assign>	Task or Sub-Process with Tasks for the copy constructs
<compensate>	Throwing Intermediate Compensation Event
<compensateScope>	Throwing Intermediate Compensation Event
<empty>	Task
<exit>	Throwing Intermediate Terminate Event
<extensionActivity>	Vendor-specific rendering
<invoke>	Task or Sub-Process (if <faultHandlers> and/or <compensationHandler> is defined)
<receive>	Task or Catching Intermediate Message Event
<reply>	Task or Throwing Intermediate Message Event
<rethrow>	Throwing Intermediate Error Event
<throw>	Throwing Intermediate Error Event
<validate>	Task
<wait>	Catching Intermediate Timer Event

Table 2. Mapping of BPEL Structured Activities

BPEL Construct	BPMN Mapping
<flow>	Introduces additional Gateways, visualization of links as sequence flow connections, links in and out of structured activities have impact on other renderings
<forEach>	Sub-Process with forEach Marker, early completion condition as attached Catching Intermediate Conditional Event
<if>	The branches of the <if> construct are surrounded by XOR-Gateways, the 'else' branch has a default sequence flow connection
<pick>	The branches are surrounded by gateways, at the entrance a complex event gateway, at the end an XOR gateway, the contained constructs have a separate mapping
<repeatUntil>	The activity nested within is connected to surrounding XOR Gateways by sequence flow connection, default sequence flow connects the gateways for the loop, alternatively a Sub-Process with a Loop Marker can be used
<scope>	Sub-Process with attached handlers, see figure 2
<sequence>	Nested activities are connected by sequence flow connections
<while>	The activity nested within is connected to surrounding XOR Gateways by sequence flow connection, normal sequence flow connects the gateways for the loop, alternatively a Sub-Process with a Loop Marker can be used

Table 3. Mapping of other BPEL constructs

BPEL Construct	BPMN Mapping
<catch>	Mapping realized by the surrounding <faultHandlers> construct, using sequence flow connection
<catchAll>	Mapping realized by the surrounding <faultHandlers> construct, using default sequence flow connection
<compensation Handler>	Sub-Process with Catching Compensation image as Marker
<correlationSets>	Data object or some other artefact or none
<documentation>	Annotation
<extensions>	Data object or some other artefact or none
<faultHandlers>	Sub-Process with Error Marker, the mapping uses the same structure as for if: each <catch> is one branch, the <catchAll> branch is connected by default sequence flow
<import>	Data object or some other artefact or none
<onEvent>	Catching Intermediate Message Event
<onAlarm>	Catching Intermediate Timer Event
<partnerLinks>	Pools, connected to the sending and receiving activities using message flow connections
<process>	Pool with artifacts, integrates contained handlers and artifacts
<terminationHandler>	Sub-Process with a feasible marker (not foreseen in BPMN)
<variable>	Data object or some other artefact or none

4.1 Example: <if>

As first example we discuss the <if> activity that is used for conditional branching, comparable to the statement in common structured programming languages. The XML-code in listing 1 shows an <if> activity in BPEL 2.0 syntax with an if-branch, one elseif-branch and an else-branch. For a better readability the code has been simplified, for example the attributes of the interaction activities partnerLink, operation and variable have been omitted. The BPMN snippet in figure 1 shows the corresponding visualization of this code.

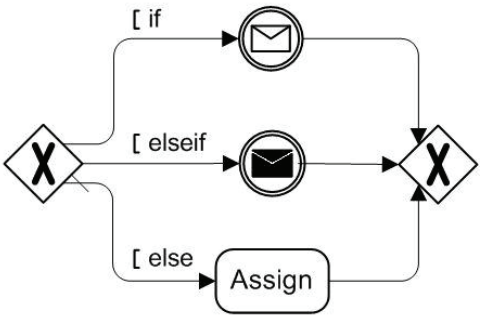


Figure 1. Visualization of an <if>activity in BPMN

The principles of the mapping algorithm result in the BPMN snippet in figure 1. The <if> block is represented by XOR-Gateways which are surrounding and embedding the nested branches. For each of the conditional branches the corresponding graphical representation is used, which is in this case a Receiving Message Intermediate Event for the <receive> activity, a Throwing Message Intermediate Event for the <reply> activity and a Task for the <assign> activity. The else-branch is connected by using a default flow connection, annotations can be used to make the representation clearer.

4.2. Example: <scope>

A far more complex example is the visualization of the <scope> activity which is shown in listing 2. This activity can be modelled as a Sub-Process in BPMN, as shown in figure 2. The artifacts like variables can be visualized by Data Objects, in this example grouped in the lower left corner of the outer Sub-Process in the figure. For some constructs various graphical representations are possible, <partnerLink>s for example could also be modelled as Pools instead of Data Objects. The outer Sub-Process in the figure

```
<scope ...>
   <partnerLinks>...</partnerLinks>
   <messageExchanges>...</messageExchanges>
   <variables>...</variables>
   <correlationSets>...</correlationSets>
   <faultHandlers>...</faultHandlers>
   <compensationHandler>...</compensationHandler>
   <terminationHandler>...</terminationHandler>
   <eventHandlers>...</eventHandlers>
   Main Activity
</scope>
```

Listing 2. Code of a <scope> activity in BPEL

points out the visibility and the scope of the <scope> artifacts like variables, which are included in the <scope> activity. The inner Sub-Process holds the main activity of the <scope> activity. This inner Sub-Process also includes the <eventHandlers>, that are installed and running as long as the nested main activity is running. This construct is placed inside the inner Sub-Process, as faults that might occur during execution of these Event Handlers are caught by the Fault Handler of the <scope> activity that is located outside this Sub-Process and thus valid for the Event Handlers. The other Handlers of the <scope> are attached to the boundary of the inner Sub-Process as Catching Intermediate Events. Those events interrupt the normal processing of the scope, which is equivalent to the execution semantics in BPEL.

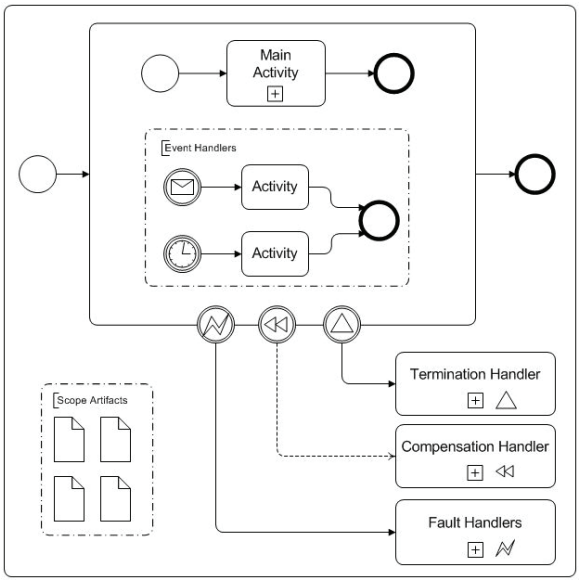


Figure 2. Visualization of a <scope>activity in BPMN

4.3. Example: <flow>

The mapping of the <flow> activity is quite challenging and partially impossible without extending BPMN. This activity can be used to describe control-dependencies in possibly, but not necessarily, parallel paths of the process model. For the definition of these dependencies <link>s are used, each connects two activities and defines thereby a control-dependency. Transition-Conditions can be defined for each <link> in order to model the process in a graph-based manner. Those conditions have to be evaluated before an activity that is target of a <link> can be started. Whether an activity has to be started or skipped can be explicitly defined in the Join-Condition. When this condition evaluates to false, the activity will not be executed, instead the Dead-Path-Elimination (DPE) [3] takes place. Thereby the statuses of all outgoing <link>s are set to false, which could cause again the skipping of other, subsequent activities. This procedure is applied until an activity is finally reached, whose Join-Condition does not evaluate to false. It should be noted, that this is just one situation where DPE can occur, it can also be initiated for example when a certain branch in an <if> activity is chosen and another branch contains outgoing <link>s. In the latter case the status of the outgoing <link>s are set to false, which can again lead to the situation described above, so a Join-Condition might not be met.

The BPMN-to-BPEL transformation in [7] and also other related work for the transformation between BPMN and BPEL such as [8] do not consider making sophisticated use of the <flow> activity, because it introduces complex execution semantics, similar to the semantics of the OR-Gateway in BPMN. This together with mismatching concepts of token flow in BPMN and control flow in BPEL are indeed problematic. We will show in the following, where exactly the problems in the visualization of the <flow> activity are and to what extent they can be solved.

Basically the visualization of BPEL with BPMN uses sequence flow connections to represent <links> within a <flow>. Preferably, and this is allowed according to [5], a different colour is used in order to allow to distinguish between them. The visualization is becoming complex, when <link>s are contained in the process that are entering or leaving structured activities, like for example a <if> activity. Then implicit sequence flow connections introduced by the visualization of the structured activity and those sequence flow connections resulting from the <link> need to be properly merged, as otherwise an unsynchronized merge would be displayed. For the join and split or possibly parallel paths, that result from this constructs, an Inclusive Gateway can be placed before and respectively after an activity with an ingoing or outgoing <link>, in accordance to the semantics described in [5].

Yet, the BPMN specification postulates that there must not be any implicit flow during the course of normal token flow. But exactly this happens when the DPE that is related to the semantics of the <link>s is integrated into BPMN in a straightforward manner. In order to comply to this rule we want to present a way how to incorporate DPE into BPMN by introducing a new kind of tokens that we call 'dead tokens'. A dead token is generated directly at an activity, whose <joinCondition> is not met. The description of the DPE in [6] dictates, that whole paths of the process graph are successively excluded from the execution (the reflexive transitive closure) until finally a <joinCondition> is reached, that is being evaluated to true or that is not possible to be evaluated at that time. Dead tokens are also created at conditional branchings, as for example in an <if> branch that is not entered, since this branch could possibly contain outgoing <link>s. The semantics of a dead token is that it passes activities, without triggering their execution. They are flowing along a dead path, until either a <joinCondition> is met or the dead branch reaches an end, for example the end of an <if> activity, marked by </if> in BPEL code and a gateway in the visualization in BPMN. Thus the gateway that represents the <joinCondition> can neutralize a dead token and even create a regular token from it. This approach presents on the one hand a possibility how the mismatch between BPEL and BPMN could be narrowed, on the other hand it shows the vast discrepancy between the concept of control flow in BPEL and token flow in BPMN. Even though dead tokens represent a possibility to incorporate the concept of DPE into BPMN, they do also turn BPMN into a more complex notation, containing details that are not known by all users of BPMN.

5. Implementation

The prototypic implementation of our approach is realized by adaption and extension of the Open Source BPEL editor 'Eclipse BPEL Designer' [2]. This tool is written in Java and mainly built on the Eclipse-frameworks EMF and GEF and thus is also realized as a set of Eclipse Plug-Ins. It already provides the functionality of graphically modelling a BPEL process, running static validation and it also has a generic interface to plug in different BPEL runtime-engines. However, the graphical representation of BPEL

constructs is not based on any standard. The adaptation mainly concerned those pieces of code that connect the elements from the model to their graphical representation, which in the case of GEF are called EditParts and Figures. To be able to easily switch on and off certain functionality, like showing partnerLink-constructs as BPMN pools, some core functions of the editor had to be adapted and extended, too [10]. The graphical representation of an example BPEL process from listing 3 is shown in figure 3.

6. Conclusion and Outlook

The possibility to visualize and model a BPEL process using a standardized graphical notation can facilitate the communication of processes between business analysts and IT developers and also among different IT departments using tools from different vendors. The main contribution of this paper is the presentation of an approach to visualize and model a BPEL process without applying complex model transformations that would introduce round-trip problems when the transformed model is being manipulated. The transformation from one model to another is avoided and executable code is directly produced. The fundamental advantage of the presented approach lies in the solely creation of valid process models that are easier to understand than BPEL code and easier to communicate as it uses a standardized notation.

Acknowledgements

The work published in this article was partially funded by the SUPER project[1] under the EU 6th Framework Programme Information Society Technologies Objective (contract no. FP6-026850) and the COMPAS project[2] under the EU 7th Framework Programme Information and Communication Technologies Objective (contract no. FP7-215175).

References

[1] E. Borger and B. Thalheim. A method for verifiable and validatable business process modeling. Advances in Software Engineering, 5316, 2008.

[2] Eclipse BPEL Project. Eclipse BPEL Designer. http://www.eclipse.org/bpel/.

[3] F. Leymann and D. Roller. Production Workflow: Concepts and Techniques. Prentice Hall PTR, 2000.

[4] Netbeans community. Netbeans BPEL Designer. http://enterprise.netbeans.org/soa/.

[5] Object Management Group (OMG). Business Process Modeling Notation Version 1.1 – OMG Standard, 2007.

[6] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution Language Version 2.0 – OASIS Standard, 2007.

[7] C. Ouyang, M. Dumas, A. ter Hofstede, and W. van der Aalst. Pattern-Based Translation of BPMN Process Models to BPEL Web Services. International Journal of Web Services Research, 5(1):42–62, 2008.

[8] J. Recker and J. Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In Proceedings 18th International Conference on Advanced Information Systems Engineering, 2006.

[9] N. Russell, A. ter Hofstede, W. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, BPMcenter.org, 2006.

[10] D. Schumm. Graphische Modellierung von BPEL Prozessen unter Verwendung der BPMN Notation. Diplomarbeit, Universität Stuttgart, IAAS (in German), 2008.

[11] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. Distributed and Parallel Databases, 14(1):5–51, 2003.

[12] M. Weidlich, G. Decker, A. Großkopf, and M. Weske. BPEL to BPMN: The Myth of a Straight-Forward Mapping. In Proceedings of the 16th International Conference on Cooperative Information Systems (CoopIS 2008), 2008.

[13] J. Vanhatalo, H. Völzer, and J. Koehler. The Refined Process Structure Tree. Business Process Management, 6th International Conference (BPM 2008), 2008.

[14] O. Kopp, D. Martin, D. Wutke, and F. Leymann. On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages. Modellierung betrieblicher Informationssysteme (MobIS), 2008.

[1] http://www.ip-super.org/
[2] http://www.compas-ict.eu/

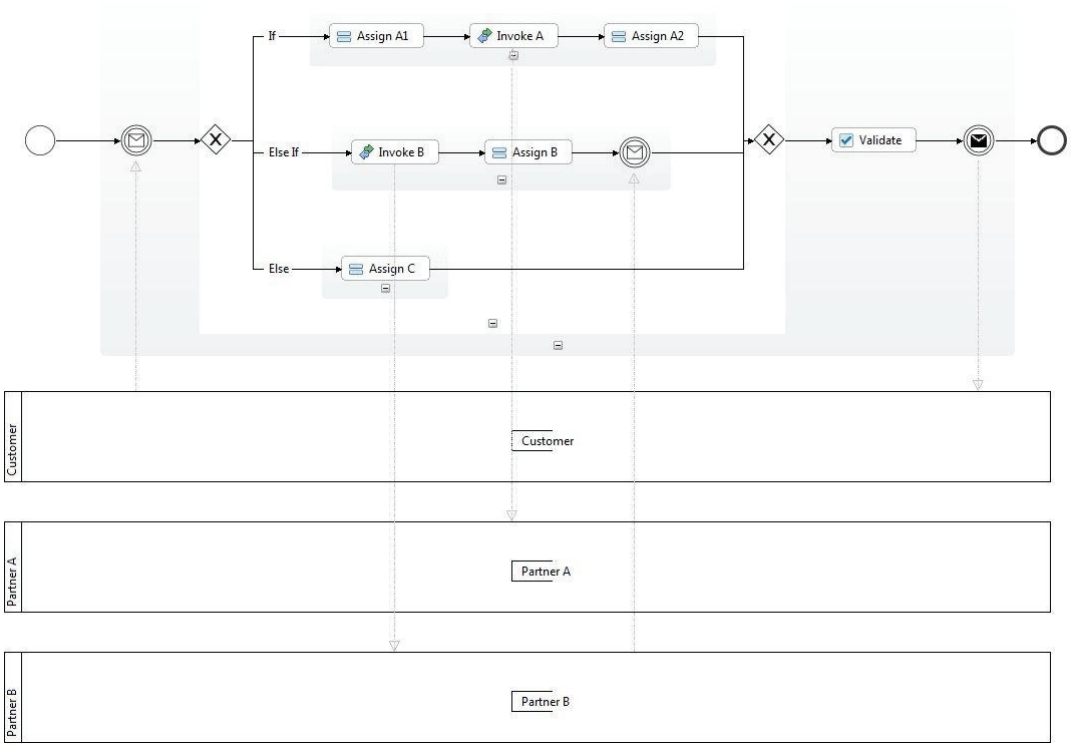


Figure 3. Screenshot of the Visualization and Modelling of BPEL with BPMN

```
<sequence name="main">
   <receive createInstance="yes" name="receiveInput" partnerLink="Customer"/>
   <if name="If">
      <sequence name="Sequence1">
        <assign name="Assign A1" />
        <invoke name="Invoke A" partnerLink="Partner A"/>
        <assign  name="Assign A2"></assign>
      </sequence>
      <elseif>
         <sequence name="Sequence">
            <invoke name="Invoke B" partnerLink="Partner B"/>
            <assign name="Assign B"></assign>
            <receive name="Receive" partnerLink="Partner B"></receive>
         </sequence>
      </elseif>
      <else>
         <sequence>
            <assign  name="Assign C"></assign>
         </sequence>
      </else>
   </if>
   <reply name="Reply" partnerLink="Customer"/>
</sequence>
```

Listing 3. Code of the main activity of the corresponding BPEL process (simplified)