OPAALS PROJECT

Contract n° IST-034824

WP2: Automatic Code Structure and Workflow Generation from Models

Del2.3 - Extended vocabulary and rule set for an existing scenario

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: IST-034824

Project Acronym: OPAALS

Deliverable N°: D2.3

Due date: August 15, 2008

Delivery Date: November 15, 2008

Short Description: This report concerns the transformation from SBVR model to executable workflows. The work extends the previously proposed approach, described in D2.2, based on SBVR for augmenting its capabilities and expressive power.

Author: T6 ECO (Antonella Filieri, Antonio Margarito), SUAS (Thomas Kurz, Raimund Eder)

Partners contributed: T6 ECO, SUAS

Made available to: OPAALS Consortium and European Commission

Versioning		
Version	**Date**	**Name, organization**
0.1	24/10/2008 (first submission)	T6 ECO,SUAS
0.2	12/11/2008 (second submission)	T6 ECO,SUAS

Quality check

Internal Reviewers: Paul Krause (UNIS), TV Prabhakar (IITK)

Dependences:

Work Packages	WP1, WP2, WP3, WP6, WP10
Partners	University of Surrey (UniS), Indian Institute of Technology Kanpur (IITK), University of Kassel (UniKassel), University of Central England Birmingham (UCE).
Domains	Computer science.
Targets	Computer Science researchers

Table of contents

Figures

Tables

1 Introduction

This work is part of WP2 and contributes to its second-phase activities. The objective of WP2 is to provide a set of guidelines and conceptual/technological means to allow the automatic and semi-automatic generation of software systems from natural language based specifications [1] [2]. Research activities toward such objective will help bridge the gap between unstructured natural language and formal structured languages, thus allowing non-computer scientists to interact with automatic systems in a simple and natural way. One of the main obstacles toward this direction is represented by the intrinsic ambiguity and semantic richness of natural language. Communication among human beings is based on terms. Those terms must be understood by both parties of the communication, and their meaning is generally influenced by the context and by specific rules. Moreover words must be combined in special ways, following a more or less formal syntax.

This work narrows the focus of WP2, assuming a precise and simplified syntax for English language and requiring explicit definition of the meaning of the used terms. In particular the work extends a previously proposed approach based on SBVR (see D2.2 [1] for more details), augmenting its capabilities and expressive power. If on the one hand the starting point is not precisely natural language, but expressions that are far less rich than natural language, on the other hand this scale-down approach helps with subsequent steps in code generation. In particular the target system is represented by an executable workflow, that is the formalisation of a process able to orchestrate already existing software components (e.g. web services, local applications,).

One of the main difficulties that have been faced during this work is the lack of SBVR [3] in supporting temporal logics. This means that it is impossible to totally leverage the SBVR metamodel, using understandable, simple and natural statements, to represent typical workflow models that are heavily based on the sequence of activities and on other temporal concepts.

If on the initial approach of D2.2 [1] we gave a temporal interpretation (see paragraph 2.4 for further explanations) to some first-order logic like expressions [11] (i.e. if <antecedent> then <consequent>), the extension of the target workflow patterns, required the introduction of new keywords (e.g., after, while) into SBVR Structured English [3].

2 XPDL workflow generation

This Section describes the improvement upon the approach to automatic workflow generation from SBVR models described in D2.2 [1]. The approach leverages both the SBVR Metamodel [3] and an ad-hoc defined Workflow Metamodel, as represented in Figure 1. The idea is to represent a workflow model, which is an instance of the given workflow metamodel, in terms of SBVR Structured English (a textual representation notation for SBVR models) and to define an automatic transformation from such textual description to an executable one (that is based on the Workflow Metamodel).

2.1 Workflow and Workflow patterns

Workflow is one of the core/key concepts of this research work. A workflow is closely connected to a business process because it allows representing the details of that process in terms of actual activities, participants, rules etc.
In other words, a workflow permits to point out:

- who are the participants involved in a business process (specifying not only the roles that they can play but also the way in which they can be organized);
- what are the activities included in a business process (specifying not only the participant that performs the activity but also the way in which he can do it) ;
- when and how the elements in the previous two points can interact (specifying, for example, when an activity begins, or ends, or the order in which the participants must execute their tasks, or the duration of each activity and so on).

Using a more rigorous terminology, a workflow consists of process logic and routing rules. The process logic defines the sequence of tasks, while the routing rules express the conditions and criteria that must be followed, as well as deadlines and other business rules implemented by the workflow engine.

Regarding the research field related to the study of workflows, an interesting analysis has been conducted by the joint effort of Eindhoven University of Technology and Queensland University of Technology [9].
Such research work provides a thorough examination of the features that have to be supported by a valid workflow language or a business process modelling language. Workflow specifications have been analysed from a number of different perspectives such as control flow, data, resources, and exceptions handling and a set of workflow patterns have been identified. Using a workflow pattern, which is a specialized form of a design pattern, the main advantage is to provide independence from the implementation technology and at the same time independence from the essential requirements of the domain they were addressed to. These patterns provide the basis for an in-depth comparison of a number of commercially available workflow management systems.
In our work we are interested above all in control flow patterns.
Also based on a subsequent study mainly focused on a control flow based perspective [10], several workflow patterns are taken into consideration in order to evaluate the coverage that different workflow languages provide for these patterns.
BPMN and XPDL were chosen between all available standards.

2.2 Workflow representations: BPMN and XPDL

A workflow model can be expressed in different ways; for example it can be visualized through a graphical diagram in order to better describing the overall structure of the business process.
In this document, BPMN (Business Process Modeling Notation) will be adopted as workflow graphical representation, in particular the latest BPMN 1.1 specifications [4] will be used.
BPMN's goal is to provide a standard notation for process diagrams. Using such a notation ensures consistency and interoperability among different users and modeling tools, so that no matter who created the diagram, the same icons are used to represent the same objects.

The purpose of XPDL is to have an XML format for the storage of BPMN diagrams. If different vendors use XPDL as their file format, they can easily exchange process models.

XPDL is a file format to promote the interoperability of tools. Some process engines run XPDL models, using extensions to add the details necessary for a run-time environment.
So, for this research-work, the adopted executable language is the XML Process Definition Language (XPDL) [5] [6] [7] that provides a file format that supports every aspect of the BPMN process definition notation, including graphical descriptions of the diagram as well as executable properties used at run time.

2.3 The adopted approach

In this sub-section the adopted approach to automatic workflow generation from SBVR models is explained.
Figure 1 provides a visual representation of such an approach. The starting point is the creation of an SBVR Vocabulary, made up of terms and fact types, upon which a Rule Set is built. Both the Vocabulary and the Rule Set are expressed in SBVR Structured English basing on the SBVR metamodel. In particular, the Vocabulary and Rule-set are built according to the SBVR 1.0 specification [3].
The Rule Set, expressed in SBVR SE, defines the workflow model that can be also considered as an instance of the Extended Workflow metamodel.
An ad-hoc parser has been developed in order to transform the textual description of the workflow metamodel in an XPDL file.
The SBVR representation of the workflow model is transformed to the equivalent XPDL representation through the definition of a set of transformation rules that allow the mapping between SBVR Structured English syntactic patterns and the corresponding workflow concepts. Basing on such set of the transformation rules, the parser is able to associate to each element of the workflow model, which is the object of transformation process, the correspondent XPDL tag.

Ultimately, the output produced by the parser in the end of the transformation process is an XPDL document that is equivalent to the original workflow model, which was expressed in SBVR SE.

Note that the XPDL file is also compliant to the XML standard, therefore inheriting all the advantages related to the interoperability guaranteed by the use of this standard.

The obtained XPDL file is then passed to a particular software, called workflow engine which is able to extract from this file the necessary information to manage and execute the workflow representing the business process.

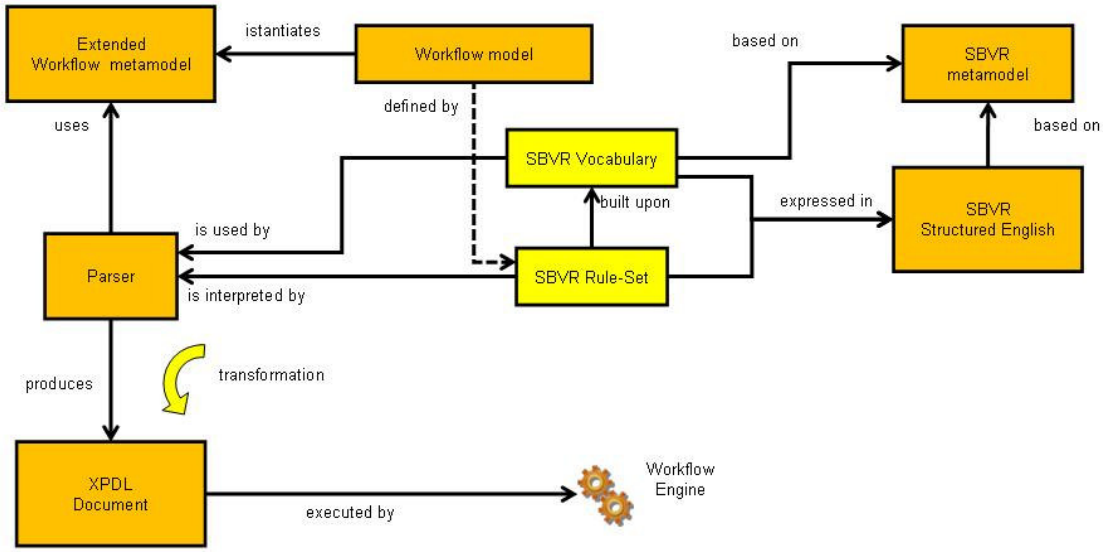


Figure 1 – The adopted approach for automatic workflow generation from SBVR models

The difference between the approach used in D2.2 and the current approach regards above all the completeness of the used workflow metamodel. The previous workflow metamodel is indeed composed of the very essential elements and constructs of BPMN/XPDL, allowing only simple processes to be represented.

The aim of this work is to extend the workflow metamodel and to define the appropriate transformation rules between its elements and SBVR SE patterns, so that a modeler can describe more complex processes and workflow using more constructs. The new workflow concepts included in the Extended Workflow

metamodel were selected among the most used between process modeling specialists and BPMN users, as explained in the paragraph 2.5.
In the following paragraph, instead, there is a description of the motivations concerning the nature and the interpretation of the Rule Set and how it is possible to describe a workflow model by means of rules, trying to overcome the lack of SBVR in supporting temporal aspects.

2.4 Supporting workflow temporal requirements

One of the main difficulties that have been faced during this work is the lack of SBVR [3] in supporting temporal logics [12]. The SBVR metamodel does not provide a temporal logic support to express temporal constraints on the truth value of a proposition. As a consequence, it is impossible to totally leverage the SBVR metamodel, using understandable, simple and natural statements, to represent typical workflow models that are heavily based on the sequence of activities and on other temporal concepts.

It is difficult and not properly correct to give a temporal interpretation to some SBVR constructs/statements which are based on the first-order logic expressions. We consider, for example, the *if <antecedent> then <consequent>* expression. The SBVR logical formulation of the if-then rules is given as “Implication”. Implication is defined in the SBVR Specification as the “binary logical operation that operates on an antecedent and a consequent and that formulates that the meaning of the consequent is true if the meaning of the antecedent is true”. This means that if the <antecedent> condition is true at the same time also the <consequent> condition is set at true as we can see in Figure 2.

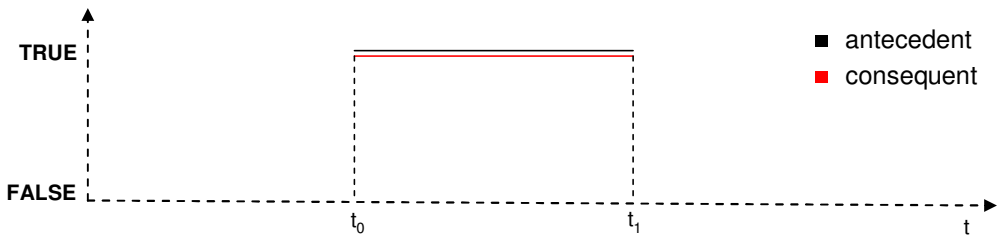


Figure 2 – Truth conditions in an if-then expression.

In the previous figure, it is clear that the antecedent and consequent conditions are simultaneously verified in the arbitrary time range [t_o, t_1].

In our work, it is important to define a temporal sequence between antecedent and consequent because they could be related to workflow concepts, such as activities or events.

Example: If <antecedent> is the activity user *inserts* card, and <consequent> is the activity system *displays* options-screen in pure declarative logical interpretation, it means that the two activities are contemporarily executed. The desired scenario, needed to satisfy the temporal requirements, is depicted in Figure 3

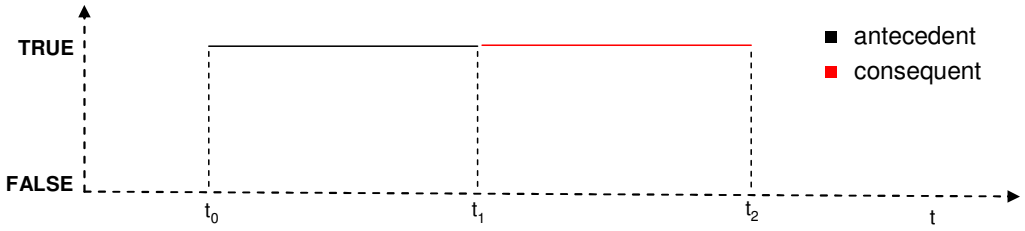


Figure 3 – Desired truth conditions scenario

In the previous figure, we can see that a temporal dimension has been introduced: only after the positive verification of the truth condition of the antecedent in the time range [t_o, t_1], then the truth condition of the consequent will be set to the true value in the time range [t_1, t_2].

In order to realize the previous scenario, we introduce new keywords (e.g. *after*, *while*) in SBVR SE, allowing the formulation of new statements that involve temporal meanings. In the paragraph 2.8, further details about the use of these new keywords will be provided.

The necessity to introduce new constructs, for example *after <antecedent> then <consequent>*, implies the adoption of a procedural-extended approach. In particular, in our work we follow a declarative approach for the definition of the Business Vocabulary and a procedural-extended one for the building of the rules. In this way, the knowledge contained in the declarative Business Vocabulary, which belong to the business domain, can be reused. Moreover, this knowledge is not connected with the definition of a business process model.

Finally, the introduction of the pattern *after <antecedent> if <condition> then <consequent>*, which is better described in the Paragraph 2.8, implies some considerations about the guidance of workflow. At the workflow design time, we could use these rules in order to model the violation of an operative rule, in which the violation of the condition (and therefore of the rule) forces the workflow to follow an alterative path.
Moreover, the operative rules could be considered at the run time. In this case, during the execution of a process it is important to intercept the violations of the operative rules. These rules impose some constraints on the possible change in the state of the business data model and they are not necessarily included in the process model.

2.5 BPMN construct usage

As demonstrated by several studies about usage of BPMN patterns, in particular using the analysis performed by Michael zur Muehlen [8], we found out that:

- Only five elements (normal flow, task, end event, start event, and pool) were used in more than 50% of the analyzed models. Such elements, plus the data-based XOR gateway, form what it is called the **common core of BPMN** (marked in yellow in Figure 4).
- Six additional elements were found in at least 25% of the models - gateways (parallel and unmarked XOR), lanes, text annotations, message flow, and start

messages. This set of elements is called **extended core of BPMN** (marked in green in Figure 4).

- The remaining 17 elements were used in a very low percentage of the analysed models.

The results of this study, in terms of Frequency distribution of BPMN construct usage are shown in Figure 4.

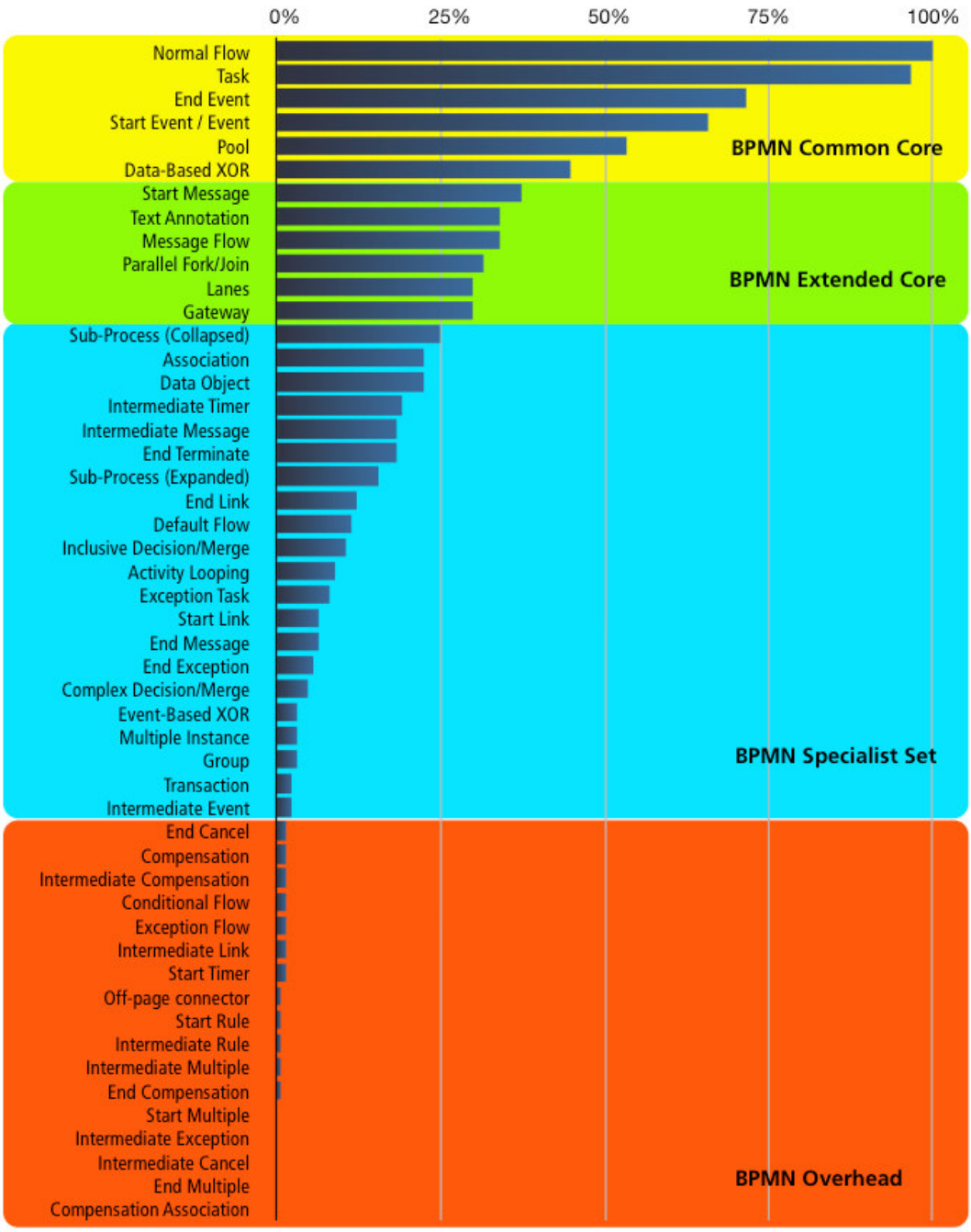


Figure 4 – Frequency distribution of BPMN construct usage [8]

Based on these studies, the most frequently used constructs have been introduced in the extended workflow metamodel as explained in the following section.

2.6 Extended workflow metamodel

A workflow model describes a set of activities, procedural steps, involved organizations or people, required input and output data needed for performing a business process.

This section shows how we extended the previously defined workflow metamodel. This enhanced metamodel includes new elements such as Pool, Lane, Intermediate Event, Message Flow.

Moreover it was necessary to modify some of the elements that were contained in the previous version of the metamodel in order to adapt it to the new configuration. In particular Activity, Gateway and Event elements have been modified.

The elements added or modified with respect to the previous version are shown in Figure 5, while the BPMN representation of such elements is reported in the Section 2.3. In particular Figure 5 provides an UML Class Diagram representation of the new metamodel. From this picture is possible to understand how the process we want to describe is composed by several elements. For every Process it is possible to define:

- one or more Activities, which compose the process, each comprising a logical, self-contained unit of work;
- one or more Participants, i.e. the actors that performs the activities;
- Transition elements, which connect Activities performed by the same Participant;
- Data Fields, which represent the variables of a process and that are typically used to maintain decision data (used in conditions) or reference data values (parameters), which are passed between activities.

It is possible to group several activities which have the same Participants under a Pool element. Lane elements are also used to group activities that have the same Performer. However, Lane elements are always components of Pool and a Pool can be split in one or more Lanes.

A Message Flow element was added to manage the messages exchange between Activities performed by different Participants or grouped in different Pools. Note that Messages Flow and Transitions have different meanings although both are used to connect Activities.

In the end you can see the variety of elements that term Activity can express: it may be specialized Task, Route or Event sub-elements.

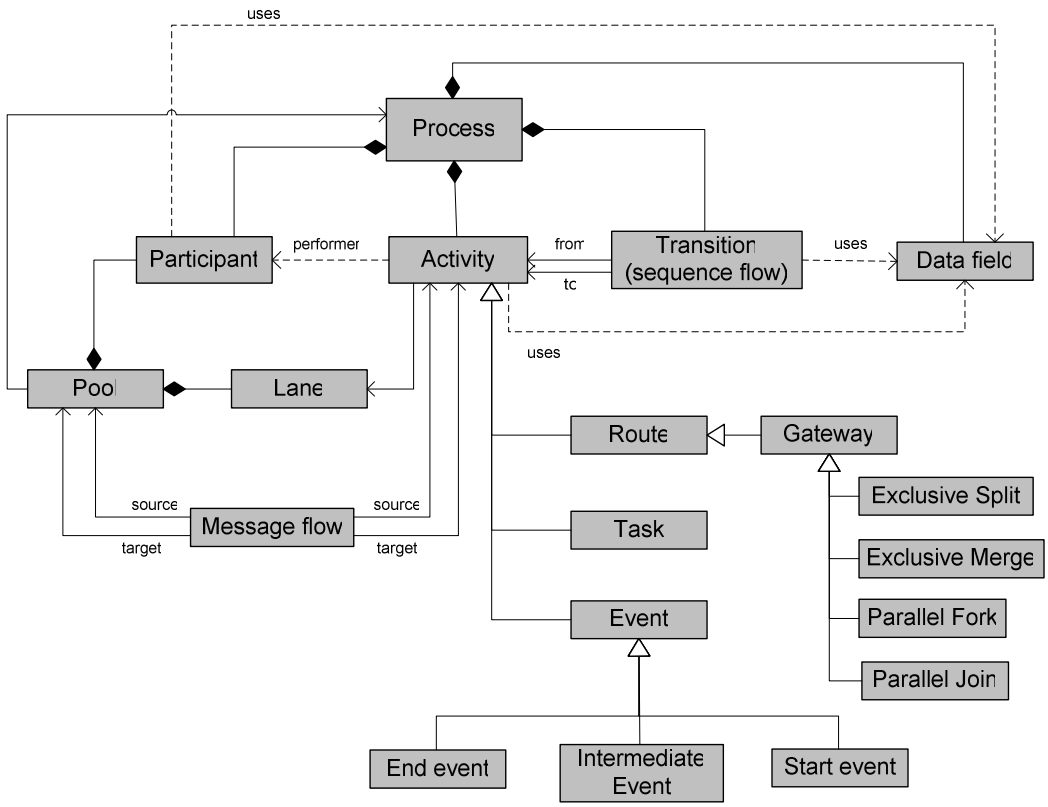


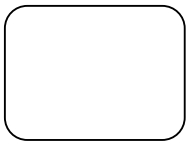
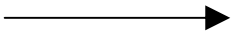
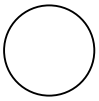
Figure 5 – Extended workflow metamodel

Note that the choice of the elements that have to be considered essential is made using the statistical result [5], exposed in the previous section.

As previously introduced, we also aim to obtain an equivalent representation of the workflow model that can be executed by a workflow engine. For this purpose, up to now, we used the XPDL 1.0 standard, but the problem with such XPDL version is that it does not support some of the proposed changes (i.e. the constructs for the intermediate events). In order to obtain a serialization file-format that is fully compliant to all the introduced constructs, we decided to adopt the XPDL 2.1 specification [5] for the executable file format.

2.7 Description of BPMN and XPDL elements/primitives

In this section the elements of the extended workflow metamodel, introduced in the previous section (Figure 5), will be described using the official specifications of the workflow metamodel defined by WfMC (Workflow Management Coalition).
In order to graphically represent the elements included in the extended workflow metamodel we used the BPMN 1.1 notation [4].
Table 1 and Table 2 describe the features of such elements also providing the correspondent BPMN 1.1 representation. In particular, Table 1 depicts the set of core element used in the first phase of this research study (D2.2 [1]). Table 2, instead, shows the set of elements used to extend the metamodel.

Element	Description	Notation
Activity	An activity is a generic term that designates some kind of work that a company performs. An activity may be a task, an event or a route activity.	
Task (Atomic)	A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of detail.	
Sequence Flow	A Sequence Flow is used to show the order that activities will be performed in a Process.	
Start Event	As the name implies, the Start Event indicates where a particular process will start.	
End Event	As the name implies, the End Event indicates where a particular process will end.	

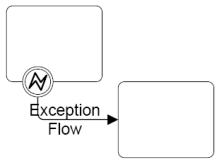
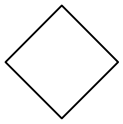
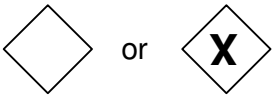
Exception Flow	Exception Flow occurs outside the Normal Flow of the Process and is based upon an Intermediate Event that occurs during the performance of the Process	
Gateway	A Gateway is used to control the divergence and convergence of Sequence Flows. Thus, it will determine branching, forking, merging, and joining of paths. Internal Markers will indicate the type of behaviour control.	
Data Field	Data Field defines the data created and used during process execution. The data is made available to activities or applications executed during the process and may be used to pass persistent information or intermediate results between activities and/or for evaluation in conditional expressions. XPDL includes definition of various basic and complex data types, (including date, string, etc.). Activities, invoked applications and/or transition conditions may refer to process relevant data field. There is no BPMN representation for this concept.	

Table 1 – Core element set of extended workflow metamodel

Element	Description	Notation
Message Flow	A Message Flow is used to show the flow of messages between two participants that are prepared to send and receive them. In BPMN, two separate Pools in a Diagram will represent the two participants (e.g., business entities or business roles).	
Pool	A Pool represents a Participant in a Process. It also acts as a "swimlane" and a graphical container for partitioning a set of activities from other Pools, usually in the context of B2B situations.	
Lane	A Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally. Lanes are used to organize and categorize activities.	
Gateway Control Types	Icons within the diamond shape will indicate the type of flow control behaviour. The types of control include: • Exclusive decision and merging. Data-Based can be shown with or without the "X" marker. • Parallel forking and joining Each type of control affects both the incoming and outgoing flow	Exclusive Data-Based  Parallel 

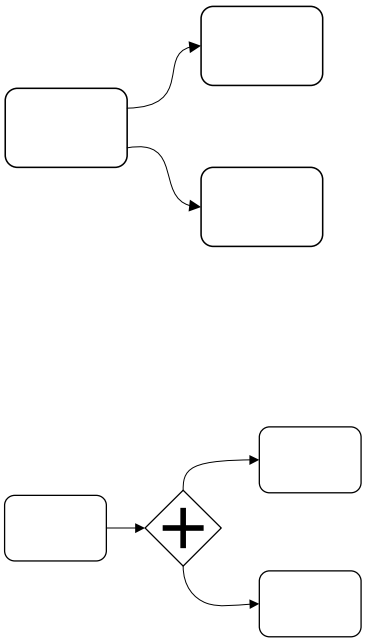
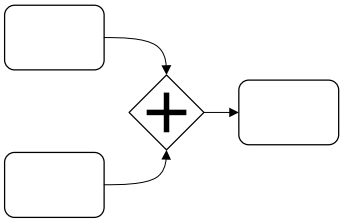
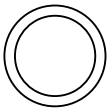
Fork	BPMN uses the term “fork” to refer to the dividing of a path into two or more parallel paths (also known as an AND-Split). It is a place in the Process where activities can be performed concurrently, rather than sequentially. There are two options: • Multiple Outgoing Sequence Flow can be used (see figure top-right). This represents “uncontrolled” flow and is the preferred method for most situations. • A Parallel Gateway can be used (see figure bottom-right).	
Join	BPMN uses the term “join” to refer to the combining of two or more parallel paths into one path (also known as an ANDJoin or synchronization). A Parallel Gateway is used to show the joining of multiple Flow.	
Intermediate Event	Intermediate Events occur between a Start Event and an End Event. They will affect the flow of the process, but will not start or (directly) terminate the process. Intermediate Events have “Triggers” that define the cause for the event. There are multiple ways that these events can be triggered.	

Table 2 – Extended element set of workflow metamodel

2.7.1 Intermediate events

This section provides further explanations about the use of events, with the specific reference to the intermediate ones. An event is something that "happens" during the course/execution of a business process and affects the flow of the process. In particular in BPMN there are various types of Intermediate Events. These are categorized according to the cause (trigger) of the event itself. In this work we use three kinds of trigger: Message, Timer and Error.
Table 3 displays the used triggers and the graphical markers that allow distinguishing one from another.

Element	Description	Notation
Message	A message arrives from a participant and triggers the Event. If used to "catch" the message, the Event marker is unfilled (i.e., white envelope). If used to "throw" the message, the Event marker is filled (i.e. black envelope) If used for exception handling it will change the Normal Flow into an Exception Flow.	Catch  Throw 
Timer	A specific time-date or a specific cycle (e.g., every Monday at 9am) can be set that will trigger the Event. If used within the main flow it acts as a delay mechanism. If used for exception handling, it will change the Normal Flow into an Exception Flow.	
Error	This type of Event can only be attached to the boundary of an activity, thus it reacts to (catches) a named error, or to any error if a name is not specified.	

Table 3 – Intermediate Event types

Moreover, an Intermediate Event can be placed within the normal flow (Figure 6) in order to “catch” the event trigger. In this way, only the Message Event or Timer Event can be used as trigger for the event.

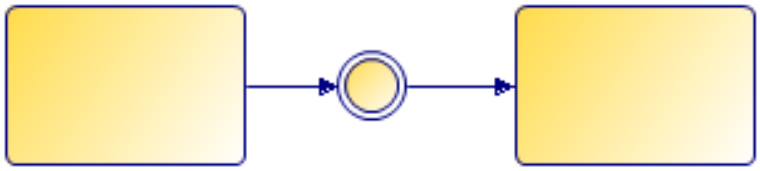


Figure 6 – Intermediate Event within a normal flow

In the following the concept of “token” will be introduced. A “token” will take the usual meaning of an object which represents the exclusive privilege to perform some operation. This privilege is passed from an element to another after completion of the operation.

In other words, when a token arrives at an intermediate event then it waits until a trigger signal occurs (e.g. a message arrives, or time exceeds) and, only after that, the token is passed towards the next element on the outgoing sequence flow.

In the case of Message Intermediate Events that are used to “throw” a message, the trigger of the event immediately occurs (e.g., the message is be sent) and the token follows the outgoing sequence flow.

Alternatively, an Intermediate Event can be used as exception handling, by placing it on the boundary of a task (Figure 7), changing the normal flow into exception flow.

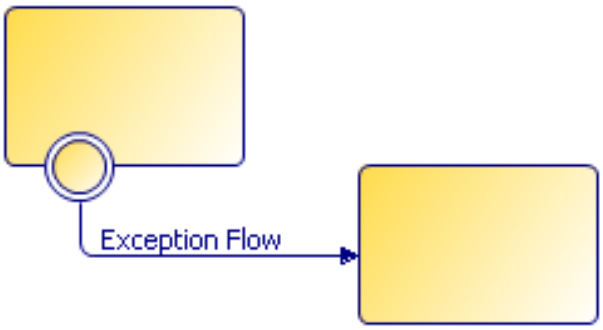


Figure 7 – Intermediate Event used as exception handling

An intermediate event that is attached to the boundary of an activity can only “catch” the event trigger. The triggers of the event, which is used as “exception catch”, can be of type Message, Timer or Error.

2.8 From SBVR to XPDL

In this section the process of transformation from the SBVR [3] representation of a workflow to an equivalent BPMN/XPDL representation is explained. Such transformation process requires the definition of a set of rules necessary to realize the mapping operations from SBVR Structured English syntactic constructs to workflow concepts. In other words, such rules allow obtaining the BPMN graphical notation and the XPDL serialisations of a workflow model starting from a vocabulary and a rule set defined in SBVR Structured English.
Such transformation rules are explained in the following two subsections, listed according to whether they operate on the vocabulary or on the rule set.

2.8.1 Vocabulary

Pool
A Pool represents a Participant in a Process and in the BPMN graphical description can contain flow objects (activities) and sequence flows (transition) between them. A Process can contain more than one Pool.
Interactions between Pools is only allowed through Message Flow (see Section 2.3.2 for more details)

Starting from a vocabulary, it is possible to identify all the participants involved in a process and, consequently, all the pools we must consider in the process description. In particular a participant/pool corresponds to the *<placeholder 2>* of each binary fact type that has the pattern *“<placeholder 1> is role for <placeholder 2>”* where the verb phrase *<is role for>* has to be defined as SBVR keyword.

If the vocabulary does not include the above mentioned fact type then a participant/pool will be inserted in the process definition for each different subject of the binary fact type included in the vocabulary. Moreover a lane with the same name of the participant/pool will be created.

For example, from the following SBVR statement:

director is role for bank

we can derive that Bank is a Pool for the Process and that Director is a Lane for that Pool.

The equivalent BPMN graphical representation is showed in Figure 8:

Figure 8 – Example of pool

The statement is associated to the following XPDL code:

```
<Package Id="PackageID" Name="PackageName">
        <Participants>
                <Participant Id="ParticipantID"  Name="ParticipantName" ></Participant>
        </Participants>
        <Pools>
                <Pool Id=" PoolID" Name=" Bank" Participant="ParticipantID"  Process=" ProcessName">
                        <Lanes>
                                <Lane Id="LaneID"Name="Director">
                                        <Performers>
                                                <Performer> </Performer>
                                        </Performers>
                                </Lane>
                                .....

                        </Lanes>
                        <NodeGraphicsInfos>
                                <NodeGraphicsInfo LaneId=XX></NodeGraphicsInfo>
                        </NodeGraphicsInfos>
                </Pool>
        </Pools>

...
...
</Package>
```

Lane

A pool can be divided into one or more lanes. In BPMN a lane is a graphical representation for internal roles (e.g., Manager), systems (e.g., an enterprise application), an internal department (e.g., shipping, finance). A lane is often used to indicate a performer for a set of activities.

In other words, the possibility to assign a "role name" to a lane allows the activities that are contained into the lane to inherit this name as performer. The XPDL tag Performer, child of Lane, specifies the role required to perform any of the activities in the lane. It is possible, however, to specify a different performer for each single activity, using a Performer tag child of Activity.

Starting from the SBVR vocabulary, a lane can be identified in two ways.

If the vocabulary contains the binary fact type that has the pattern *"<placeholder 1> <is role for> <placeholder 2>"* then a lane will be included in the process with the name specified in the *<placeholder 1>*. Moreover, the created lane must belong to the pool whose name is indicated by the *<placeholder 2>.*

If the vocabulary does not include the above mentioned fact type then each *"<placeholder 1>" of the* binary fact type that has the pattern *"<placeholder 1> <verbPhrase> <placeholder 2>"* will be inserted as lane in the process. It is also necessary to create a pool for each different lane, from which it inherits the name.

For example from statements,

director is role for bank

employee is role for bank

we derive that Bank is a Pool for the Process and that Director and Employee are both lanes for that Pool.

This situation is represented in BPMN notation as showed in Figure 9.

Figure 9 – Example for Lane

This situation is represented by corresponding XPDL code reported below:

```
<Pools>
      <Pool Id="PoolID " Name="Bank" Process="ProcessName">
          <Lanes>
                    <Lane Id="_LaneID1" Name="Director">
                       .<Performers>
                              <Performer>PerformerName </Performer>
                        </Performers>
                    </Lane>

                    <Lane Id="LaneID2 " Name="Employee">
                       <Performers>
                              <Performer>PerformerName </Performer>
                        </Performers>
                    </Lane>
                              <!-- more lane -->
           </Lanes>

      </Pool>
</Pools>
```

Task

A task is an atomic activity that is included within a process. An activity must have a performer.

A task is derived from each binary fact type that has the pattern "*<placeholder 1> <verbPhrase> <placeholder 2>"* where:

- *<placeholder 1>* represents the performer
- *<verbPhrase> <placeholder 2>* represents the activity to be performed (i.e. the task)
- *<verbPhrase> is different from "is role for"*

From the sentence:

system *displays* welcome-screen

It is possible to identify the System as the Performer of the task, thus deriving the following BPMN diagram

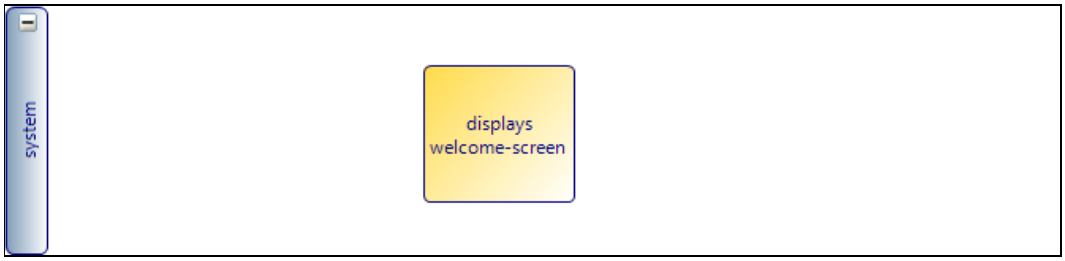


Figure 10 – Example for task

and XPDL code:

```
<WorkflowProcesses>
  <WorkflowProcess Id="WorkflowID" Name="ProcessPackage-Process" >

    <Activities>
      <Activity Id="ActivityID " Name="displays welcome-screen">
        <Implementation>
            <No/>
        </Implementation>
        <NodeGraphicsInfos>
          <NodeGraphicsInfo BorderColor="0,0,128"  Height="63.0" LaneId="LaneID"  Width="96.0">
            <Coordinates XCoordinate="309.0" YCoordinate="139.0"/>
          </NodeGraphicsInfo>
        </NodeGraphicsInfos>
      </Activity>

    <!-- more activities -->

    </Activities>
  </WorkflowProcess>
</WorkflowProcess
```

2.8.2 Rule Set

Sequence Flow

A sequence flow shows the order of the activities in a process. Usually, sequence flows are used to model uncontrolled flows such as those that are not affected by conditions or do not pass through a gateway.

Starting from the SBVR Rule Set it is possible to identify a sequence flow by selecting the rules that have the pattern "a*fter <antecedent> then <consequent>*". Note that both *<antecedent>* and *<consequent>* are binary fact types and both *after* and *then* are keywords.

For example, the following SBVR statement:

after the sales *processes* the order then the distribution *packs* the goods

represents the Sequence Flow between the activities "processes order" and "packs goods" as shown in Figure 11. The activity named "packs goods" is the next activity to perform after the execution by the sales department of the activity named "processes order". We also know that the performer of the activity "pack goods" is the distribution department.

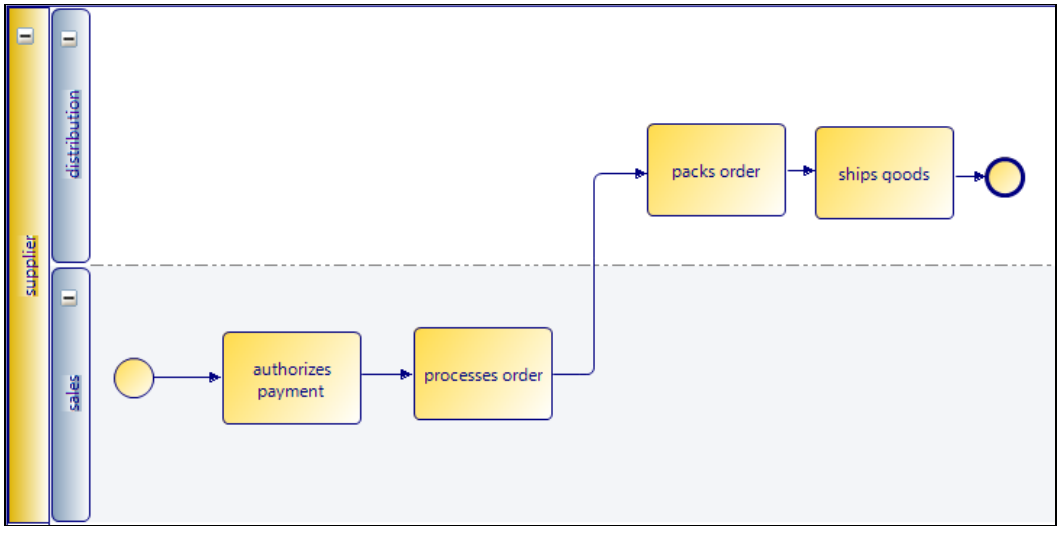


Figure 11 – Example of Sequence Flow

From the above SBVR statement the equivalent XPDL code is reported below:

```
<WorkflowProcesses>
   <WorkflowProcess Id="WorkFlowID " Name="ProcessPackage-Process">
     <Transitions>
            ….

       <Transition Id="TransitionID" Name="TransitionName" From="StartingActivityID" To="EndingActivityID">

   <!--  in this case startingActivityID is the ID of "processes order"  and EndingActivityID is the ID of “packs goods
”-->

       </Transition>
            ….
     </Transitions>
  </WorkflowProcess>
</WorkflowProcesses>
```

Parallel gateways

Parallel gateways are used for two purposes:

- to create parallel flows realizing the parallel fork workflow pattern.
- to synchronize parallel flows realizing the parallel join workflow pattern.

Parallel Fork (AND-Split) Gateways

Parallel fork allows splitting a flow into two or more parallel branches each of which is executed concurrently.

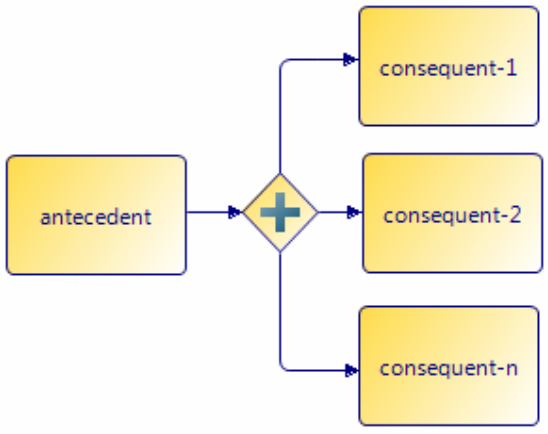


Figure 12 – Parallel Fork Gateways

Starting from an SBVR rule set, there are two ways to recognize a parallel fork:

- Selection of the rules that have the pattern:
  after <antecedent> then <consequent-1> and <consequent-2> ... and <consequent-n>. Note that *<antecedent>* and all *<consequent>s* are binary fact type.
- Selection of the set of the rules that have the pattern *after <antecedent> then <consequent>*. Such rules must have the *<antecedent>* based on the same binary fact type.

For example if we have the rule:

after the system *checks* the availability then the system *updates* the account and the system *delivers* the money

or, alternatively, the set of rules:

after the system *checks* the availability then the system *updates* the account

after the system *checks* the availability then the system *delivers* the money

we deduce that two different activities have to be performed concurrently, rather than sequentially: first of all the system checks the availability of money in the user's account. After that the system concurrently updates user's account and delivers money. This situation corresponds to the BPMN representation showed in Figure 13.

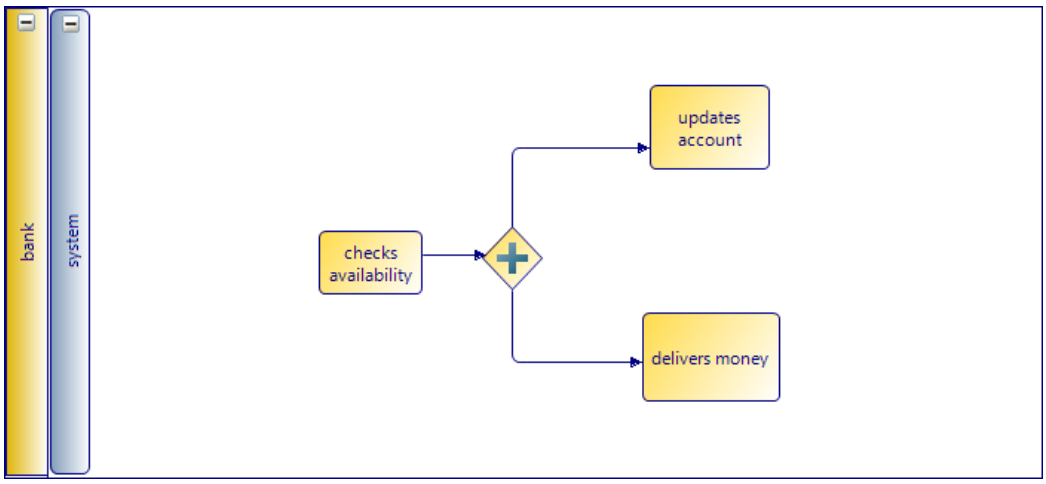


Figure 13 – Example of Parallel Fork

From this SBVR statement is generated the equivalent XPDL code reported below:

```
<WorkflowProcesses>
        <WorkflowProcess Id=”WorkFlowID " Name="ProcessName">
                ..
                <Activities>
                        <Activity Id="ActivityID_1" Name="Check Availability" />
                        <Activity Id="ActivityID_2" Name="Update Account" />
                        <Activity Id="ActivityID_3" Name="Deliver Money" />

                        <Activity Id="GatewayID" Name=”ActivityName”>
                                <Route GatewayType="Parallel"/>
                                <TransitionRestrictions>
                                        <TransitionRestriction>
                                                <Split Type=”Parallel”>
                                                    <TransitionRefs>
                                                        <TransitionRef Id=”second_transitionID”/>
                                                                <TransitionRef Id=”third_transitionID”/>
                                                    </TransitionRefs>
                                                </Split>
                                        </TransitionRestriction>
                                </TransitionRestrictions>
                        </Activity>

                ..
                </Activities>
                <Transitions>
                        <Transition Id="first_transitionID" From="ActivityID_1" T0="GatewayID"/>
                        <Transition Id="second_transitionID" From="GatewayID" T0="ActivityID_2"/>
                        <Transition Id="third_transitionID" From="GatewayID" T0="ActivityID_3"/>
                </Transitions>
        </WorkflowProcess>
</WorkflowProcess>
```

Parallel Join (AND-Join) synchronization

A parallel join allows the convergence of two or more incoming sequence flows into a single outgoing sequence flow. Regarding the execution/control aspects, the process has to wait the arrival of all signals before it can continue.

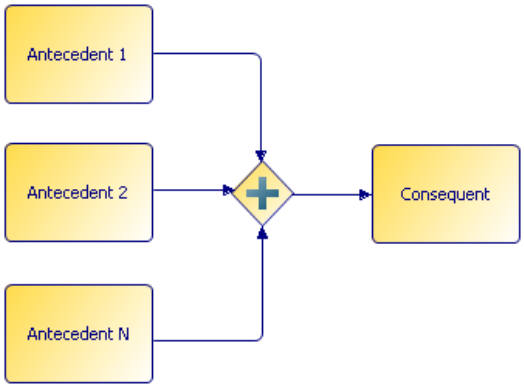


Figure 14 – Parallel Join

Each rule that has the pattern *After <antecedent 1> and <antecedent 2> and ..... <antecedent N> then <consequent>* with N arbitrary, is mapped to a parallel join. Note that all antecedents and the consequent are binary fact type.
For example, from the rule:

after the system *updates* the account and the system *delivers* the money then the system *returns* the card

we understand that only after the system performs both activities, i.e. updates the account and delivers the money, it returns the card.
This situation corresponds to the BPMN representation showed in Figure 15:

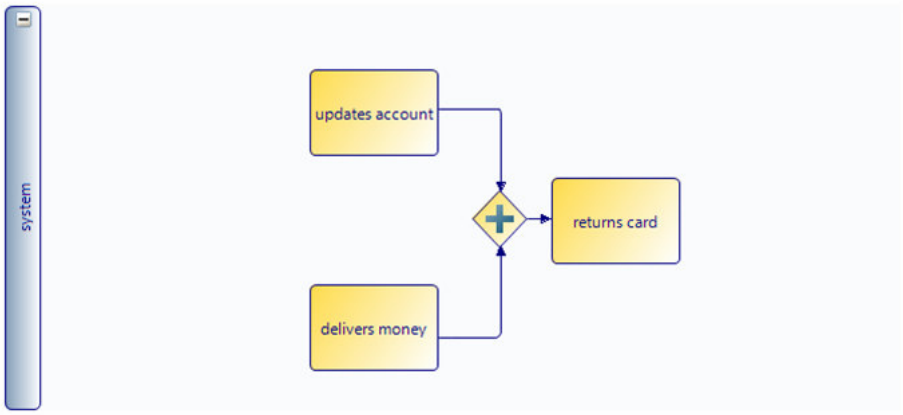


Figure 15 – Example of Parallel Join

From this SBVR statement is generated the equivalent XPDL code reported below:

```
<WorkflowProcesses>
  <WorkflowProcess Id="_WorkFlowID" Name="ProcessName">

    <Activities>                    ...
      <Activity Id="ActivityID" Name="ActivityName">
        <Route GatewayType="Parallel"/>
      </Activity>

    </Activities>
  </WorkflowProcess>
 </WorkflowProcess>
```

Exclusive Merge Gateways

An exclusive gateway can also be used as a merge for alternative Sequence Flow. This element allows the convergence of two or more branches into a single subsequent branch. Because only one between alternative paths can occur during the process execution, the activation of this incoming branch causes the passage of control to the subsequent branch.

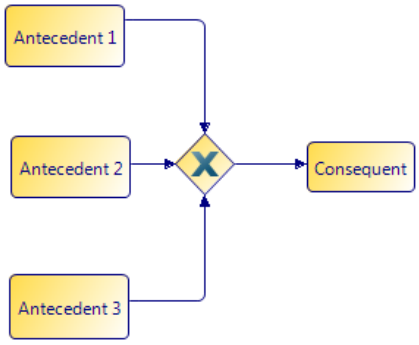


Figure 16 – Exclusive Merge

Each rule that has the pattern *after <antecedent 1> or <antecedent 2> or …… <antecedent N> then <consequent>* with N arbitrary, is mapped to an exclusive merge. Note that both antecedents and consequent are binary fact types.
From example from the rule:

after the system *delivers* the money or the system *accepts* the money then the system *returns* the card

we understand that after the activities of delivering or accepting money the return card activity will be performed. There is not any mechanism of synchronization: a performed activity immediately will pass the token to the subsequent. This situation is represented in BPMN in Figure 17

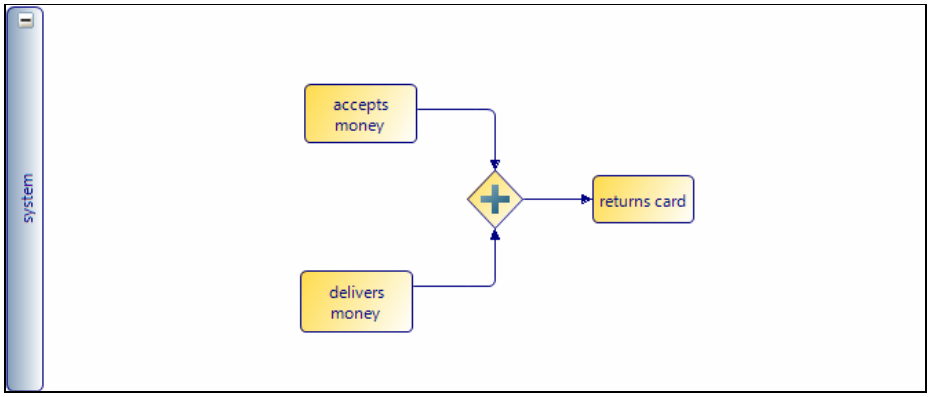


Figure 17 – Example of Exclusive Merge Gateway

From this SBVR statement is generated the equivalent XPDL code reported below:

```
<WorkflowProcesses>
  <WorkflowProcess Id="WorkFlowID" Name="ProcessName">

     <Activities>                    ...
        <Activity Id="ActivityID"  Name=”ActivityName”>
           <Route GatewayType="Exclusive" MarkerVisible="TRUE" />

     </Activities>
  </WorkflowProcess>
</WorkflowProcesses>
```

Exclusive Split Gateways

Exclusive gateways represent a point within the process where the sequence flow can take two or more alternative paths. The type of exclusive gateway most commonly used is data-based exclusive gateway also called Data-Based XOR or XOR-split. Each outgoing sequence flow has a condition expression whose boolean value is used to determine if the associated sequence flow has to be taken/chosen. Only one of the outgoing sequence flows may be taken during the execution of the process.

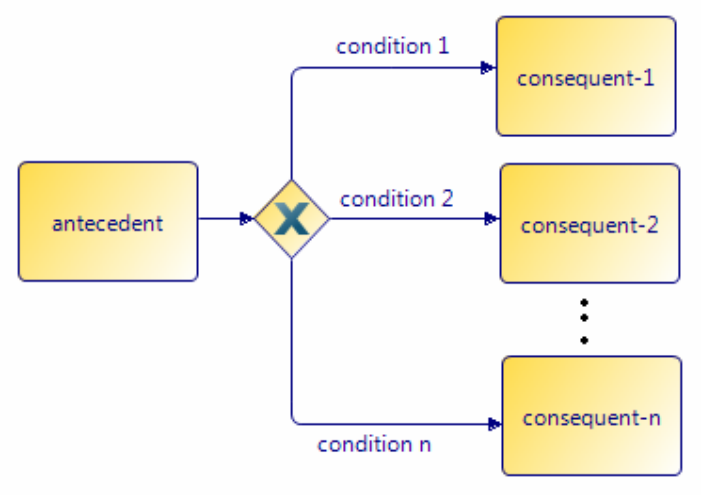


Figure 18 – Exclusive Gateway

Each rule that has the pattern *after <antecedent> if <condition i> then <consequent i>*, with i generic index from 1 to arbitrary n, is mapped to data-based exclusive gateway. Both *<antecedent>* and *<consequent i>* are binary fact types while *<condition i>* can be either a binary or a unary fact type.

For example from the rules:

after the system *checks* the card-validity if the card *is valid* then the system *displays* the pin-screen

and

after the system *checks* the card-validity if the card *is not valid* then the system *displays* the not-valid-card-screen

we understand that there is a decision point after the performer executes the activity called “checks card-validity”. If the card is valid then the “displays pin-screen” activity will be executed otherwise the “display not-valid-card screen” activity will be performed.

The above explained situation has an equivalent BPMN representation as showed in Figure 19.

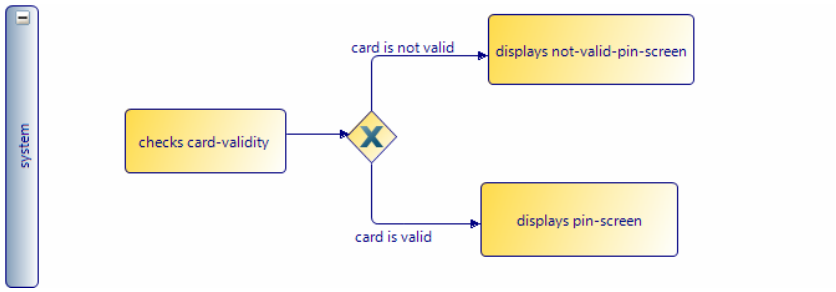


Figure 19 – Example of exclusive gateway

The following is the equivalent generated XPDL code:

```
<WorkflowProcesses>
        <WorkflowProcess Id=”WorkFlowID " Name="ProcessName">
                ..
                <Activities>
                        <Activity Id="ActivityID_1" Name="Check Card Validity"/>
                        <Activity Id="ActivityID_2" Name="Display Not valid card screen"/>
                         <Activity Id="ActivityID_3" Name="Display Pin Screen"/>

                        <Activity Id="GatewayID" Name=”ActivityName”>
                                <Route GatewayType="EXCLUSIVE"/>
                                <TransitionRestrictions>
                                        <TransitionRestriction>
                                                <Split Type="EXCLUSIVE">
                                                    <TransitionRefs>
                                                            <TransitionRef Id=”second_transitionID”/>

                                                            <TransitionRef Id=”third_transitionID”/>
                                                    </TransitionRefs>
                                                </Split>
                                        </TransitionRestriction>
                                </TransitionRestrictions>
                        </Activity>

                ..
                </Activities>
                <Transitions>
                        <Transition Id="first_transitionID" From="ActivityID_1" T0="GatewayID"/>
                        <Transition Id="second_transitionID" From="GatewayID" T0="ActivityID_2"/>
                                <Condition Type="CONDITION">card is valid</Condition>
                        <Transition Id="third_transitionID" From="GatewayID" T0="ActivityID_3"/>
                                <Condition Type="CONDITION">card is not valid</Condition>
                </Transitions>
        </WorkflowProcess>
</WorkflowProcess>
```

Message Flows

A Message Flow allows the communication, as exchange of messages, between two participants that are represented in BPMN notation by two separate Pools as shown in Figure 20. A message flow cannot connect two objects within the same pool.

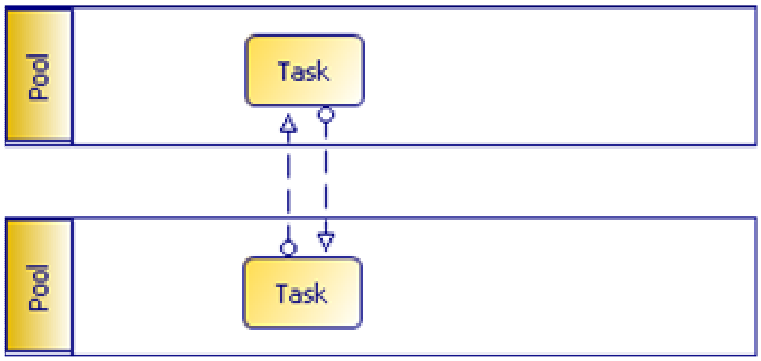


Figure 20 – Message Flow BPMN representation

Starting from a rule set it is possible to recognize the message flow elements by selecting the rules that have the patter *after <antecedent> then <consequent>*, where *<antecedent>* and *<consequent>* are based on a binary fact type. A further check is needed to distinguish the message flow from sequence flow since they are based on the same pattern. In order to recognize a message flow, the subjects of the two fact types, which are contained in the rule, must be:

- referred to two different participants/pools
- referred to performers/lanes belonging to different participants/pools

For example, from the rule:

after the user *inserts* the card then the system *checks* the card-validity

we extract that there is a message flow between the two tasks “inserts card”, performed by the user, and “checks card-validity”, performed by the system. We observe that there are two different performers for the activities. This situation is showed in an equivalent BPMN notation in Figure 21.

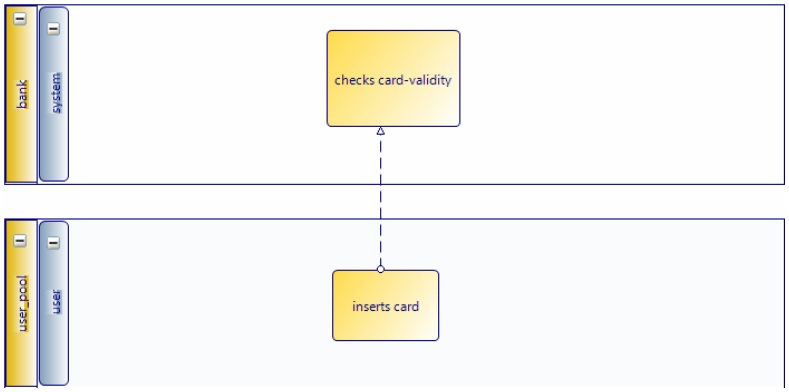


Figure 21 – Example of Message Flow

The following XPDL code is generated from the above SBVR statement:

```
....
<MessageFlows>
   <MessageFlowId="MessageFlowID"          Name="MessageFlowName"          Source="StartActivityID"
Target="EndActivityID ">

   </MessageFlow>
 </MessageFlows>
....
```

Timer Intermediate Events

An intermediate event, which is triggered by a timer, can be placed in a normal sequence flow, as shown in Figure 22, in order to introduce a delay after the execution of an activity.

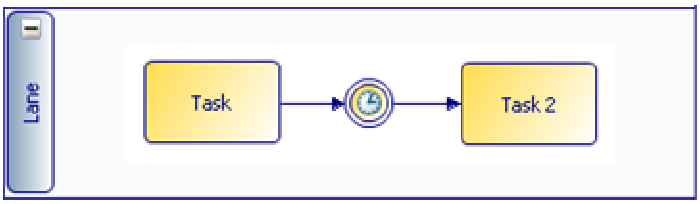


Figure 22 – Timer Intermediate Event within a normal flow

Each rule that has the pattern *after <antecedent> wait until <event> then <consequent>* identifies an intermediate event. Note that both *<antecedent>* and

<consequent> are based on a binary fact type. The structure of *<event>* defines the nature of the event type. The intermediate event will be a Timer Intermediate Event if the *<event>* is based on a binary fact type that has the following structure:

time-elapsed is <time unit>

where:

- *time-elapsed* is a key word;
- *<time unit>* can be expressed by any quantifier of time (for example second, minute or hour) that has been previously defined in the vocabulary.

From the following rule:

after the system *displays* the error-screen wait until time-elapsed *is* 30 sec then the system *displays* the welcome-screen

we deduce that a delay of 30 seconds will be introduced in the performance of the second activity after the first one is completed. This situation is showed in Figure 23 in an equivalent BPMN representation.

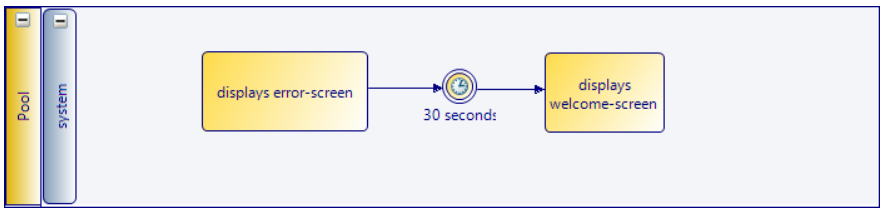


Figure 23 – Example of Timer Intermediate Event within a normal flow

From the above SBVR statement is generated the next equivalent XPDL code:

```
….
<Activity Id=" ActivityID" name=”ActivityName”>
        <Event>
          <IntermediateEvent Trigger="Timer">
             <TriggerTimer>
                      <TimeDate ExpressionType=”FormatType”>30 sec</TimeDate>
             </TriggerTimer>
          </IntermediateEvent>
        </Event>
</Activity>
….
```

An intermediate event can also be used for exceptions handling as we can see in fig 22.

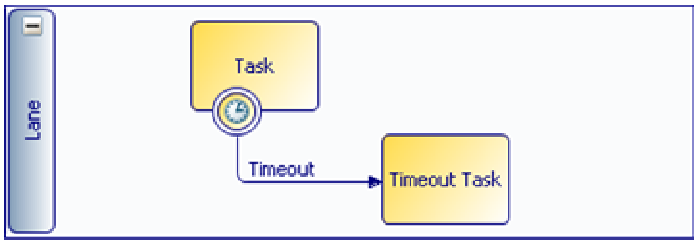


Figure 24 – Timer Intermediate Event within an exception flow

Starting from a rule set, it is possible to identify all intermediate events used in an exception flow by selecting the rules that have the pattern:
while <antecedent> if <event> then <consequent>
Note that both *<antecedent>* and *<consequent>* are based on a binary fact type.
The structure of *<event>* defines the nature of the event type. The intermediate event will be a Timer Intermediate Event only if *<event>* is based on a binary fact type that has the following structure:
time-elapsed is <time unit>
where:

- *time-elapsed* is a key word;
- *<time unit>* can be expressed by any quantifier of time (for example second, minute or hour) that has been previously defined in the vocabulary.

For example from the following rule:

while the administrator *moderates* the email-discussion if the time-elapsed *is* 7 days then the administrator *reviews* the status-of-discussion

we derive that after a timer intermediate event occurs, in this example after seven days from the first moderation event in an email discussion performed by the administrator, then an alternative flow will be followed in the performance of the process. So, seven days after the “moderates email-discussion” activity, the “reviews status-of-discussion” activity will be performed instead of the “evaluates discussion-progress” activity. This situation is showed in Figure 25 in an equivalent BPMN representation.

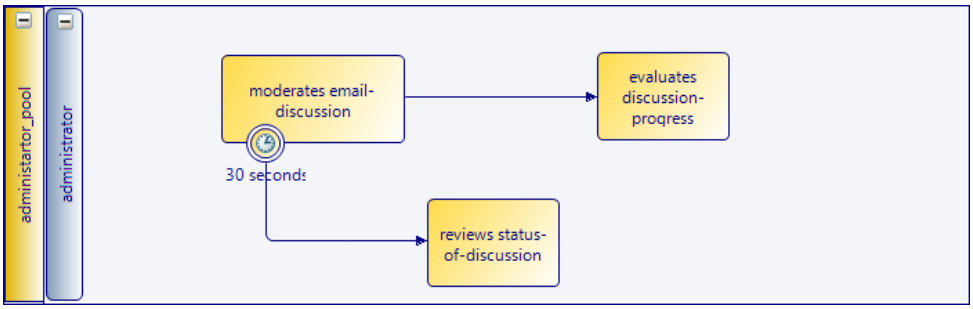


Figure 25 – Example of Timer Intermediate Event within an exception flow

From the above SBVR statement is generated the next equivalent XPDL code:

```
<Activity Id=" ActivityID" name=”ActivityName”>
        <Event>
          <IntermediateEvent Target="ActivityID" Trigger="Timer">
                    <!—in Target attribute we specify the ID of the activity on which the intermediate event is attached-->
            <TriggerTimer>
                    <TimeDate ExpressionType=”FormatType”>7 Days</TimeDate>
            </TriggerTimer>

          </IntermediateEvent>
        </Event>
 </Activity>
```

Message Intermediate Events

An intermediate event triggered by sending or receiving a message can be placed in a normal sequence flow, as showed in Figure 26 and Figure 27, in order to introduce a delay after the execution of an activity.

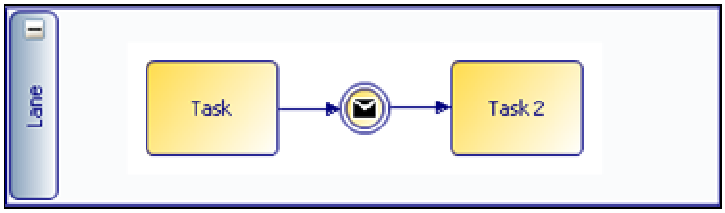


Figure 26 – Sending Message Intermediate Event within a normal flow

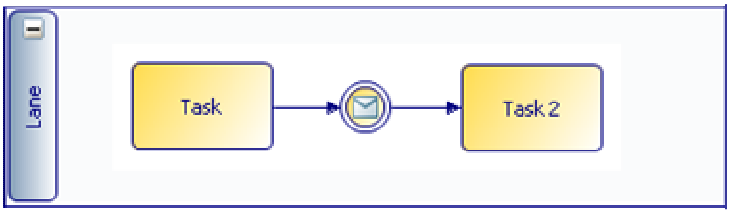


Figure 27 – Receiving Message Intermediate Event within a normal flow

The starting point is the assumption that a rule that has the pattern *after <antecedent> wait until <event> then <consequent>* identifies an Intermediate Event. The structure of the <event> placeholder defines the nature of the event type.
An intermediate event will be a sending Message Intermediate Event only if *<event>* has the structure:
message is sent
where *message* is a generic reference to any possible message-based event supported by XPDL specification.

An intermediate event will be an incoming Message Intermediate Event only if *<event>* is based on a binary fact type that has the following structure:
message arrives
where *message* is a generic reference to any possible message-based event supported by XPDL specification.

A Message Intermediate Event can also be used for exceptions handling. Only a Receiving Message Event can be placed on the boundary of an activity to redirect the execution flow in an exception flow, as shown in Figure 28.

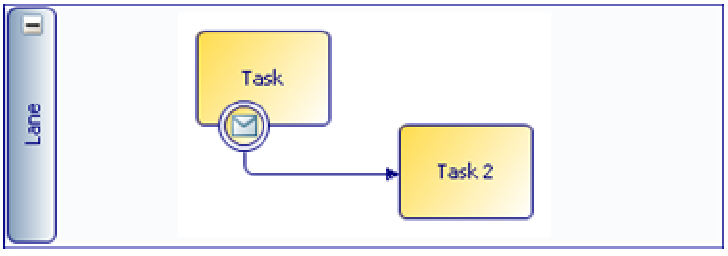


Figure 28 – Example of Message Intermediate Event within an exception flow

Starting from a rule set, it is possible identify all intermediate events used in an exception flow by selecting the rules that have the pattern:
while <antecedent> if <event> then <consequent>
Note that both *<antecedent>* and *<consequent>* are based on a binary fact type.
The structure of *<event>* defines the nature of the event type. The intermediate event will be a Message Intermediate Event only if *<event>* is based on a binary fact type that has the following structure:
message arrives
where *message* is a generic reference to any possible message-based event supported by XPDL specification.

Error Intermediate Events

An Error Intermediate Event can only be placed within an exception flow as shown in Figure 29. This type of event can only be attached to the boundary of an activity, thus it reacts to (catches) a named error, or to any error if a name is not specified. The Error Event only responds to errors generated within the activity on which is attached.

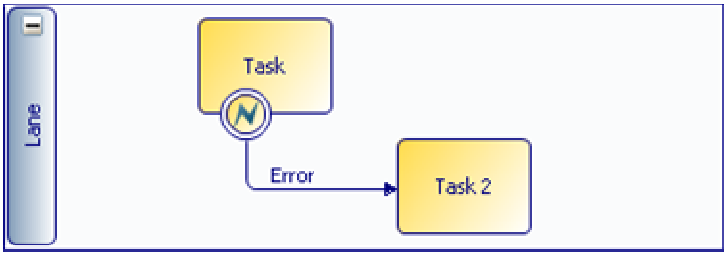


Figure 29 – Error Intermediate Event

Each rule that has the pattern *while <antecedent> if <event> then <consequent>* identifies an intermediate event. Note that both *<antecedent>* and *<consequent>* are based on a binary fact type. The structure of *<event>* defines the nature of the event type. The intermediate event will be an Error Intermediate Event if *<event>* is based on a unary fact type that has the structure:
error occurs

For example from the following rule:

while the user *books* the flight if the error *occurs* then the system *handles* the error

we derive that during the execution of the activity “books flight” performed by a user the occurrence of an error causes the start of the activity performed by the system to handle the error. This situation is showed in Figure 30 in an equivalent BPMN representation.

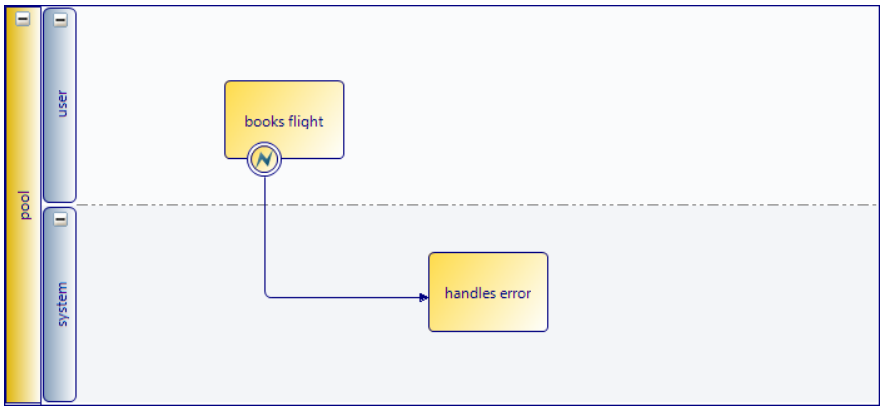


Figure 30 – Example of Error Intermediate Event

From the above SBVR statement is generated the next equivalent XPDL code:

```
<Activity Id="ErrorID">
        <Event>
          <IntermediateEvent Target="ActivityID" Trigger="Error">
                  <!—in Target attribute we specify the ID of the activity on which the intermediate event is attached--
>
             <ResultError ErrorCode=""/>
          </IntermediateEvent>
        </Event>
</Activity>
```

3 Example

This section provides an example that extends the simple scenario used in the deliverable D2.2 [1].

The extended example, showed in Figure 31, describes the workflow to perform a withdrawal or a deposit from an automatic teller machine (ATM). This simple workflow scenario is used to show how the approach introduced above can be used to model workflows that comply with the simple workflow metamodel of section 2.2.

In particular, the process includes four new XPDL/BPMN primitives that are: intermediate timer event, exclusive merge gateway, exclusive gateway and message flow. These new elements are pointed out in the Figure 31 by the red sketched circle. The intermediate timer event is inserted not only in the normal flow but also in the exception flow. The process is cyclic because it starts and ends with the same task that is the visualization of a welcome screen by the ATM.

Each ATM user has an ATM card as well as a personal identification number (PIN). After the insertion of the card, the ATM system checks its validity. If the card is not valid then the display will show an error card message otherwise it will prompt the user to enter the PIN.

After the visualisation of the PIN screen, the user has sixty seconds to insert the code. If the sixty seconds expire the system shows a timeout error. The intermediate timer event, on the boundary of the “displays pin-screen” task, is used to model this behaviour.

After the user provides his private PIN, the system checks for the PIN validity and in the negative case it displays a PIN error message.

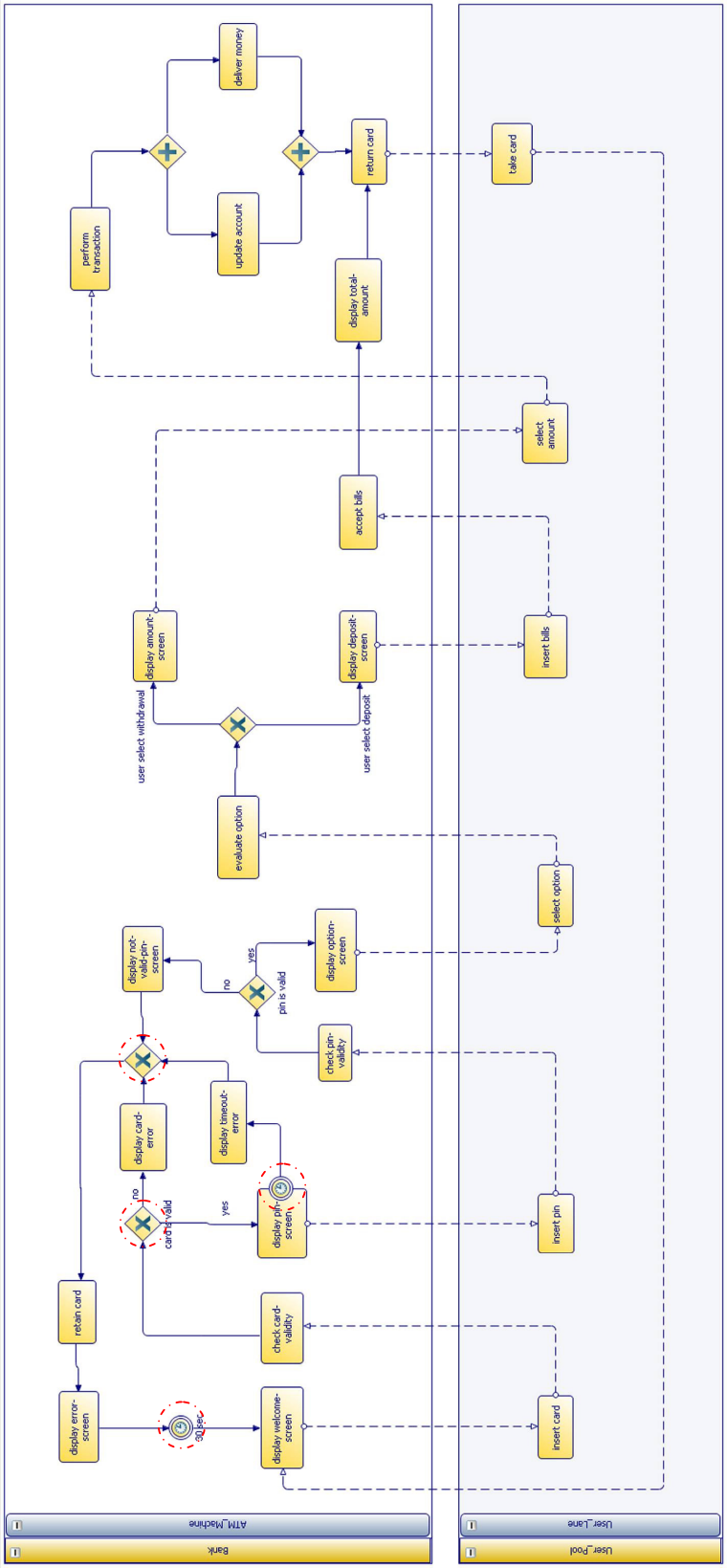


Figure 31 – Example of ATM withdrawal

In a real scenario if an incorrect PIN has been used then the system will give the possibility to try again and only after three subsequent attempts the machine retains the card. In this simplified process only one attempt is allowed. Anyhow this assumption does not compromise/alter the process. If the card is not valid or the timer is exceeded or pin is not valid then the system retains the card. In order to model this behaviour, an exclusive gateway is used as shown in Figure 32.

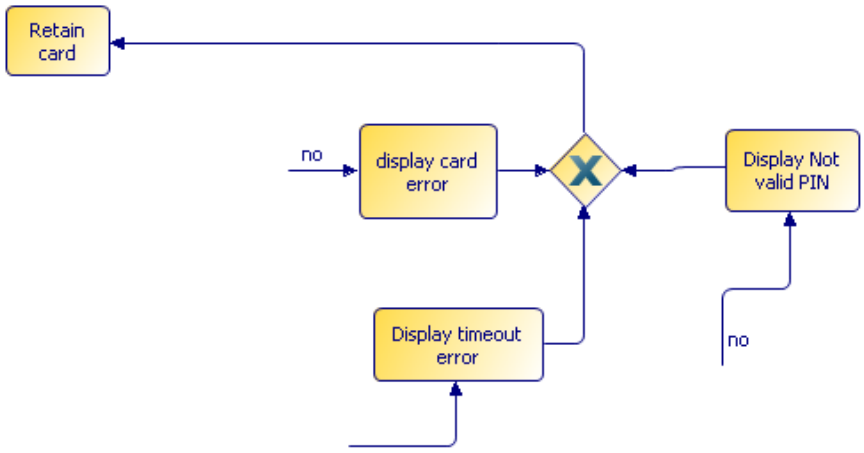


Figure 32 – Detail of the exclusive merge gateway

If the card is retained by the ATM then the system displays an error message in which the user is invited to contact the financial institution, owing the machine, in order to retrieve his card. The process waits thirty seconds in order to give the user the possibility to read the message before it displays the welcome screen. This behaviour is modelled through the insertion of an intermediate timer event within the normal flow of the activities, as shown in Figure 33.

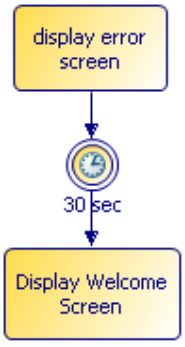


Figure 33 – Detail of the timer intermediate event

Only after a positive check of the card and the PIN number, the ATM is ready to proceed with a transaction, therefore it shows the available options which include: 1) withdraw; 2) deposit.
Assuming the user selects the withdrawal option, the ATM displays a set of monetary amounts and allows the user to choose one of these. After the user selects an amount the system performs the transaction and the flow of activities is split into two parallel activities. The first one regards the update of the user's bank account while the second one concerns the delivery of the money.
Assuming now that the user has selected the deposit option in order to deposit an amount of money into his bank account. In this case, a cash deposit screen is displayed with a message that invites the user to insert the money into the acceptor slot. After the user inserts his bills, the system counts them and displays the total amount of cash.
After the execution of the deposit or withdrawal transaction the system returns the card to the user and the process ends.

According to the transformation rules stated in 2.4, the above workflow example can be represented by an SBVR model. The SBVR Vocabulary of such model is shown in Figure 34.

Terms
bank
atm
user
welcome-screen
error-screen
card
card-validity
card-error
pin-screen
pin
pin-validity
not-valid-pin-screen
timeout-error
option
option-screen
withdrawal
not-valid-pin
deposit
money
amount
deposit-screen

Fact types
atm *is role for* bank
atm *displays* welcome-screen
user *inserts* card
atm *checks* card-validity
card *is valid*
atm *displays* card-error
atm *displays* pin-screen
user *inserts* pin
atm *checks* pin-validity
pin *is valid*
atm *displays* not-valid-pin-screen
atm *displays* option-screen
user *selects* option
user *selects* withdrawal
user *selects* deposit
atm *displays* amount-screen
atm *displays* pin-screen
atm *displays* deposit-screen
atm *accepts* bills
user *inserts* bills
atm *delivers* money

amount-screen
bills
transaction
account
total-amount
time-elapsed
second

atm *performs* transaction
atm *returns* card
user *takes* card
atm *displays* total-amount
atm *evaluates* option
time-elapsed *is* second

Figure 34 – An SBVR vocabulary for modeling the ATM workflow example

Moreover, a rule set, based on the previous vocabulary, is needed in order to completely describe the workflow example. The SBVR Rule Set of such model is shown in Figure 35.

after the atm *displays* the welcome-screen then the user *inserts* the card
after the user *inserts* the card then the atm *checks* the card-validity
after the atm *checks* the card-validity then if the card *is valid* the atm *displays* the pin-screen
after the atm *checks* the card-validity then if the card *is not valid* the atm *displays* the card-error
while the atm *displays* pin-screen if time-elapsed *is* 30 seconds then atm *displays* the timeout-error
after the atm *displays* the timeout-error then the atm *retains* the card
after the atm *displays* the card-error then the atm *retains* the card
after the atm *displays* not-valid-pin-screen then the atm *retains* the card
after the atm *retains* the card then the atm *displays* the welcome-screen
after the atm *displays* the error-screen wait until time-elapsed is 30 seconds then the atm *displays* the welcome-screen
after the atm *displays* the welcome-screen then the user *inserts* the pin
after the user *inserts* the pin then the atm *checks* the pin-validity
after the atm *checks* the pin-validity if the pin *is valid* then the atm *displays* the option-screen
after the atm *checks* the pin-validity if the pin *is not valid* then the atm *displays* the not-valid-pin-screen
after the atm *displays* the option-screen then the user *selects* the option
after the user *selects* the option then the atm *evaluates* the option
after the atm *evaluates* the option if the user *selects* withdrawal then the atm *displays* the amount-screen
after the atm *evaluates* the option if the user *selects* deposit then the atm *displays* the deposit-screen

after the atm *displays* the deposit-screen then the user *inserts* the bills
after the user *inserts* the bills then the atm *accepts* the bills
after the atm *accepts* the bills then the atm *displays* the total-amount
after the atm *displays* the amount-screen then the user *selects* the amount
after the user *select* the amount then the atm *performs* the transaction
after the atm *performs* the transaction then the atm *updates* the account
after the atm *performs* the transaction then the atm *delivers* the money
after the atm *updates* the account and the atm *delivers* the money then the atm *returns* the card
after the atm *returns* the card then the user *takes* the card
after the user *takes* the card then the atm *displays* the welcome-screen

Figure 35 – An SBVR Rule Set for modeling the ATM workflow example

4 Conclusions

In this document we have extended the approach presented in D2.2 for automatic workflow generation from SBVR models. The mapping among workflow primitives/constructs and SBVR Structured English syntactic patterns has been enriched and the ATM example has been extended with the new workflow elements. In particular the new elements are: Pool, Lane, Intermediate Events, Parallel Gateways (join and merge), Exclusive Gateways (join and merge), Message Flows.
Note that they have been chosen with respect to public results on frequency distribution of BPMN construct usage.
As to the next steps, starting from the Phase I (D2.2) SBVR-based workflow-modeling framework that allows business users to automatically generate executable workflows, a prototype for dynamically reconfiguring such workflows will be designed and developed.
Activities related to workflow reconfiguration can be divided into two steps:

- theoretical approach to automatic workflow reconfiguration: reconfiguration will be applied to workflow models that are extended according to the results of D2.3. Note that such extension requires the use of XPDL 2 as the target representation for the workflow.
- development of a prototypical implementation based on an existing workflow engine. This activity was planned according to the roadmap of the workflow engine Nova Bonita, which is the most promising project of open source Java workflow engine. Indeed such roadmap (http://wiki.bonita.objectweb.org/xwiki/bin/view/Main/Roadmap) scheduled a basic support of XPDL 2 to be available by May 2008 and the full support by December 2008. Unfortunately the roadmap is not respected. (http://wiki.bonita.objectweb.org/xwiki/bin/view/Main/Downloads).

When Nova Bonita, or some other workflow engine currently under observation, will provide a full support to the XPDL 2 specification, the development of the prototypical implementation will be possible in its complete version. Until then we will leverage the

available support of Nova Bonita to XPDL 1 in order to provide a simple reconfiguration example.

Bibliography

[1] Raimund Eder, Thomas Kurz, Thomas J. Heistracher, Mario Russo, Antonella Filieri, Prabhakar TV, Hagen Peukert, Alexandros Marinos, Stan Hendryx - D2.2 - *Automatic code structure and workflow generation from natural language models.* OPAALS Project, March 2008.

[2] Raimund Eder, Thomas Kurz, Thomas J. Heistracher, Victor Bayon, Mario Russo, and Antonella Filieri. D2.1 - *Design of Software Generation Prototype*. OPAALS Project, October 2007.

[3] OMG. Semantics of Business Vocabulary and Business Rules Specification (SBVR), January 2008. http://www.omg.org/spec/SBVR/1.0/PDF/. Last accessed on 02/01/2008.

[4] OMG. Business Process Modeling Notation (BPMN 1.1), February 2008. http://www.bpmn.org/.

[5] WfMC. XML Process Definition Language (XPDL 2.1), March 2008. http://www.wfmc.org/xpdl.html.

[6] WfMC. XML Process Definition Language (XPDL 2.0), October 2005. http://www.wfmc.org/xpdl.html.

[7] WfMC. XML Process Definition Language (XPDL 1.0), October 2002. http://www.wfmc.org/xpdl.html.

[8] Michael zur Muehlen, Jan Recker. (Jun 16, 2008). *How Much Language is Enough? Theoretical and Practical Use of the Business Process Modeling Notation*, 20th International Conference on Advanced Information Systems Engineering (CAiSE 2008), Montpellier, France, June 16-20, 2008., Springer LNCS

[9] W.M.P van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Distributed and Parallel Databases, July 2003. http://www.workflowpatterns.com/documentation/documents/wfs-pat-2002.pdf

[10] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns : A Revised View. BPM Center Report BPM-06-22 , BPMcenter.org,2006. [http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf

[11] Amit Raj, T. V. Prabhakar, Stan Hendryx *Transformation of SBVR business design to UML models*. Proceedings of the 1st conference on India software engineering conference. Hyderabad, India 2008, ACM ISBN:978-1-59593-917-3.

[12] Stan Hendryx. *Semantic Interpretation in Terms of the SBVR Logical Formulation of Semantics Vocabulary*. August 2008.

[13] Raimund Eder, Antonella Filieri, Thomas Kurz, Thomas J Heistracher, Miriam Pezzuto, *Model-transformation-based Software Generation utilizing Natural language notations*. DEST 2008. 2nd IEEE International Conference.