



envision
environmental services infrastructure with ontologies

Deliverable D3.2:

MaaS Composition Portal Version 1 – Initial development and user guide

Date:	23 rd of December 2010
Author(s):	Roy Grønmo, Dumitru Roman, Luís Costa (SINTEF), Ioan Toma (UIBK)
Dissemination level:	PU (Public)
WP:	3
Version:	1.0
Keywords:	Service Composition, MaaS Composition Portal, BPMN, Oryx, BPEL
Description:	MaaS composition portal – ready for quality review



ICT for Environmental Services and
Climate Change Adaption

Small or Medium-scale Focused Research Project
ENVISION (Environmental Services Infrastructure with Ontologies)
Project No: 249120
Project Runtime: 01/2010 – 12/2012

Document metadata

Quality assurors and contributors

Quality assuor(s)	Aphrodite Tsalgaidou (NKUA), Nils Rune Bodsberg (SINTEF MET)
Contributor(s)	All partners

Version history

Version	Date	Description
0.1	3 rd of November 2010	Outline
0.2	28 th of November 2010	Added content to the modeling section
0.3	6 th of December 2010	Added Executive summary, Introduction and Transformation sections
0.6	8 th of December 2010	Added all sections. Ready for internal review
0.7	21 st of December 2010	Comments from two internal reviews are used to improve the document. Version ready for quality review
1.0	23 rd of December 2010	Comments from the technical coordinator are addressed. Final version.

Contents

Executive Summary	3
1. Introduction	5
2. Composition Portal	6
3. Modeling Styles	7
3.1. BPMN Modeling	7
3.2. Declarative Modeling	9
4. Transformations from BPMN	11
4.1. BPMN to BPEL Transformation	11
4.2. BPMN to WSDL Transformation	14
5. Installation Guide	16
5.1. Installing and setting up PostgreSQL	16
5.2. Installing Apache Tomcat	17
5.3. Installing the MaaS Composition Portal	17
5.4. Usage Manual and Screenshots	17
6. Summary and Features for Next Release	20
Bibliography	21
A. Appendices	24
A.1. Declarative Modeling Notation	24
A.2. XSLT Transformation: DI XML to BPEL	26
A.3. XSLT Transformation: DI XML to WSDL	32

Executive Summary

This deliverable reports on the first release of the ENVISION composition portal. We follow the recommendations of D3.1 that the ENVISION service composition language should be based on (a subset of) the Business Process Modelling Notation (BPMN) and that the ENVISION composition portal should adopt and extend the open source Oryx Web-based BPMN editor. The first release of the composition portal provides:

1. implementations of Oryx extensions to support OGC-specific properties of BPMN tasks;
2. transformation of BPMN compositions to executable BPEL processes and associated WSDL files; and
3. deployment of an Oryx instance (together with the above mentioned extensions) on a server to be used within ENVISION.

This deliverable provides details on the aforementioned items and puts them in the context of the ENVISION overall architecture. In addition, this deliverable outlines the features planned for the next release of the composition portal.

The first release of the composition portal presented in this deliverable is functional and already available for internal use for the ENVISION consortium. So far the composition portal has been used to model compositions (and generate corresponding BPEL processes) related to the use cases.

1. Introduction

Deliverable *D3.1 MaaS Composition Portal Architecture specification* [5] identified a set of requirements for the ENVISION Model-as-a-Service (MaaS) composition portal and presented a review of current composition languages and tools. It concluded that a service composition language based on (a subset of) the Business Process Modelling Notation (BPMN) [1] and the adoption and extension of the open source Oryx Web-based BPMN editor [13] would be the most proper approach for the ENVISION composition portal. With this deliverable, we report on the status of the first release of the ENVISION composition portal which takes into account the recommendations provided by D3.1.

This first release of the composition portal is characterized by the following:

1. implementations of Oryx extensions to support OGC-specific [8] properties of BPMN tasks;
2. transformation of BPMN compositions to executable BPEL processes [6] and associated WSDL files [14]; and
3. deployment of an Oryx instance (together with the above mentioned extensions) on a server to be used within ENVISION.

In this deliverable we provide further details on the deployment and extensions of Oryx and the transformations to BPEL and WSDL. Furthermore, we outline the features of the next release. The current release is functional and has been used for modeling compositions (and to generate corresponding BPEL processes and associated WSDL files) related to the use cases.

The remainder of this deliverable is structured as follows. Section 2 provides an overview of the current implementation of the MaaS composition portal and its interaction with other ENVISION components. Section 3 provides an insight on the advantages and disadvantages of the different modeling styles for compositions, and exemplifies these styles with the compositions currently defined by the ENVISION use cases. Section 4 describes our automated transformations from BPMN compositions to BPEL and associated WSDL files. Section 5 describes how to install Oryx and gives a brief usage manual for the modeling of BPMN in Oryx. Section 6 summarizes the current status of the composition portal implementation and outlines the features of the next release.

2. Composition Portal

Figure 2.1 shows the composition portal and its main interactions with other components.

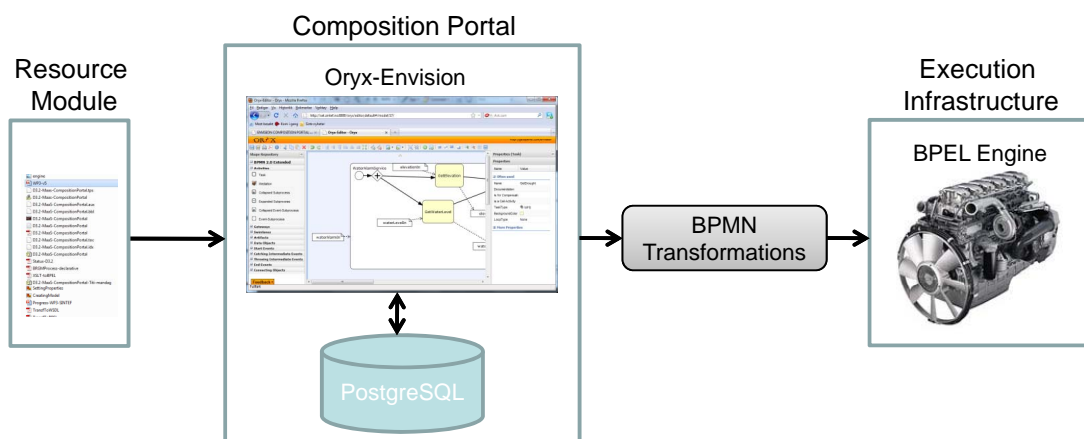


Figure 2.1.: The interaction between the composition portal and other components

The composition portal consists of an Oryx-based modeling environment and PostgreSQL database [11]. The Oryx editor has been customized and extended so that we can generate complete BPEL and WSDL documents from the BPMN compositions. For this, we needed several new properties as Oryx extensions. These concern namespace, partner link properties, port type, and WSDL file locations. More details about these properties are given in Section 4.1.1.

The resource module, which is developed in ENVISION work package 2, contains all the available resources as OGC-based and traditional Web services. Using the resource module, the modeler will select a service at a time and import it into the composition portal. This will result in a new task in the model. This task will get all the required properties assigned from the resource component so that we can generate complete BPEL and WSDL documents from the composition model.

When a BPMN composition model has been finalized it will be exported as BPEL and WSDL files (according to the transformations presented in Section 4) which are sent to the BPEL engine for deployment and execution. We note that the Execution Infrastructure is managed by ENVISION work package 6.

It is important to note that the composition portal, the resource module, as well as the BPEL engine will eventually be embedded inside various portlets in the ENVISION portal.¹

¹Further details on how the various components are wrapped as portlets in the ENVISION portal is provided in deliverable D2.2 Environmental Semantic Web Portal Version 1 – Initial development and user guide.

3. Modeling Styles

We aim for a modeling style which is user-friendly without too many technical details. Other modeling approaches suffer from a design which has too much focus on being similar to BPEL instead of providing an understandable model which is our main goal. At the same time we need enough details so that we can fully generate BPEL. Since our modeling style is less tied to BPEL than in other approaches, the transformation code from BPMN to BPEL becomes a bit more complicated.

Section 3.1 describes the way we use BPMN modeling in ENVISION. Section 3.2 describes declarative modeling as a possible extension to the traditional procedural style of modeling in BPMN, which will potentially allow users to specify compositions in a highly declarative manner.

3.1. BPMN Modeling

BPMN is typically used to model both control and data flow within the same integrated model. Existing approaches have shown that BPEL code can be generated from such a model [10, 4]. Existing approaches tend to be very tightly coupled with BPEL, which reduces the readability of the BPMN models. The receiving and replying of data objects are traditionally modeled with explicit tasks such as in BPEL. We propose instead to derive this automatically from the input and output of the outermost subProcess.

In Figure 3.1 we show a composition of BRGM’s service called `WaterAlarmService`. Each service, including the composition itself, takes exactly one input and one output parameter. The current version does not use the input variable, but this will be improved in a later version. We start two services in parallel, `GetElevation` and `GetWaterLevel`. The first service is an SOS service and the second one is a WFS service. The output from these two services are then used by the WPS service `GetDrought`, which will start after both the first two services have terminated.

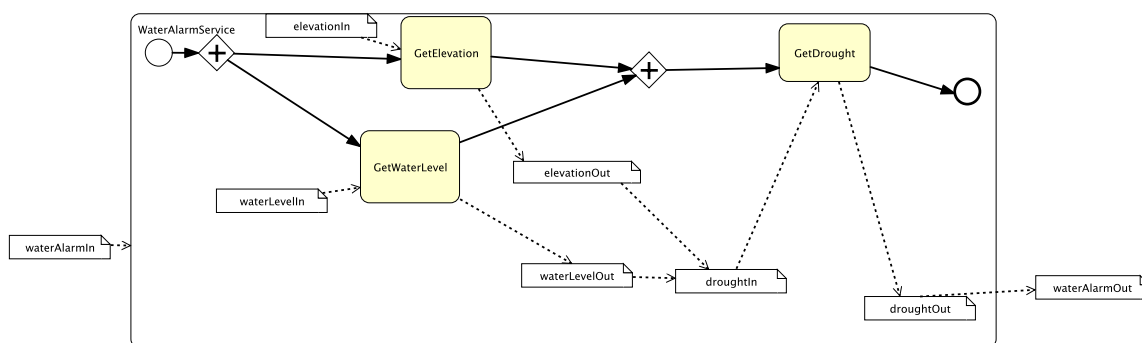


Figure 3.1.: BPMN model of a service called `WaterAlarmService`

The necessary data transformations are not currently captured in the model and need to be specified by manual additions to the generated BPEL in this first version. When our use case partners specified the composition scenarios by free drawings, they only modeled the data flow and the tasks. This gave us the idea that we could only model the data flow and automatically deduce the control flow implied by the data flow. When data is produced by some task and then later consumed by another task, we can deduce that the latter task must start after the completion of the first task. However, the plain data flow-based modeling cannot in general differ between conditional branching and parallel behavior. Without an explicit branch node it is also unclear where to specify the branching condition. For these reasons, we have chosen to model the control flow explicitly.

We require that a BPMN composition has one outermost subProcess to represent the whole composition. Inside the subProcess there shall be exactly one start and one end node. The control flow must be continuous without any dangling tasks, i.e. tasks without incoming or outgoing control flow.

The outermost subProcess represents the composite service and it shall have no incoming or outgoing control flow. We can also have nested subProcesses to structure the model into more manageable units. A task in an ENVISION model shall have exactly one incoming and exactly one outgoing control flow. This means that we require the use of the so-called *gateways* to model splits and joins (and, or, xor). BPEL does not support unstructured cycles [7], which means that our BPMN model must fulfill the so-called single-entry single-exit (SESE) property. The SESE property is fulfilled if each cycle has a single entry node and a single exit node.

In order to support the modeling of OGC-based services, we extend Oryx with task types for tasks and subProcess. The task type values allowed so far are WMS, WFS, WPS, SOS, and WCS. The task type of a task indicates how to call the existing service associated with the task, while the task type of a subProcess indicates what kind of service to produce for the new composite service. For the tasks we need to register values for several properties, so that we can fully generate BPEL from them.

For a task and a subProcess we need to register values for all the following properties: name, namespace, wsdl, task type, partner link type, and port type. All, but the name property are extensions to Oryx. Each service defines a namespace that groups all elements and ensures uniqueness of names and identifiers. The wsdl property specifies the location of the WSDL file of the service. A partner link type refers to an instance of a service and a port type refers to a group of operations for a service.

For a data object we need to register values for the name and type properties. The input data object of the outermost subProcess has no incoming data flow, while the output data object of the outermost subProcess has no outgoing data flow. For all other data objects, there is normally a data flow from a data object to another.

We implicitly assume that we need a data transformation (sometimes quite simple) to convert the source data object to the target data object in a data flow. A data object may have multiple outgoing and multiple incoming data flows. Multiple incoming data flows to a data object means that all the source data objects are needed in a data transformation to produce the target data object.

3.2. Declarative Modeling

End users have common sense notions of process and time (e.g. one task happens before another, one process happens during another) and do not think in computational models (e.g. state machines, token flow, or other "operational" semantics). It is therefore desirable that processes are specified in a rather declarative way.¹ Although BPMN is a notation that targets users with no or little IT-skills, some of its constructs that are core for specification of processes imply and require a procedural way of thinking.

For example, the process specified in Figure 3.1 explicitly requires a join gateway for specifying the fact that both `GetElevation` and `GetWaterLevel` need to complete before `GetDrought` can start its execution, that is, the process explicitly needs to specify the state before the execution of `GetDrought`. This can be seen as an over-specification of the process, since for expressing the relationships between tasks such an explicit specification of the state is not needed. Instead, the end user can focus on the direct relationships between tasks, resulting in a more declarative and intuitive specification of processes. The example in Figure 3.1 can be captured in a more declarative way through a "before" relationship between `GetElevation` and `GetDrought` on one hand, and a "before" relationship between `GetWaterLevel` and `GetDrought` on the other hand, without having to introduce explicit joint gateways.

Moreover, in case of a change in the process, e.g. `GetWaterLevel` should be executed after `GetElevation`, the initial process would need to be redesigned, which is a time consuming and cumbersome task, especially in case of a large process. By combining a procedural specification of processes with a more declarative one, the change in a process can be managed more easily. By keeping the original design of the process, the change can be captured by posing constraints on the original process in terms of new relationships between tasks (e.g. `GetWaterLevel` should take place after `GetElevation`).

For the end user it should be sufficient and more intuitive to specify the process as a set of relationships between tasks, rather than requiring the explicit specification of states in a process. Figure 3.2 depicts how such relationships can be graphically represented.² The arrows labeled with "before" imply a before-relationship between the tasks `GetElevation` and `GetDrought`, and between `GetWaterLevel` and `GetDrought`. These two relationships are placed in a box labeled with "AND", meaning that the process is a conjunction of the two before-relationships.

Note that such a declarative notation is not aimed to replace the BPMN notation for specifying processes, but rather to extend it by offering the end user the possibility to specify processes in a more declarative way. Currently, the notation for declarative modeling consists of graphical elements for specifying elementary constraints: (1) primitive constraints (such as a task must happen in a process, must happen at least or exactly n times, must not happen); (2) serial constraints (such as before a task happens another one must happen, after a task happen another one must happen, a task blocks another task); and (3) immediate serial constraints (such as right before a task another task must happen and no other task must happen in between, after a task happens another one must happen immediately). Furthermore, the graphical notation allows for conjunctive and disjunctive composition of constraints, resulting in a powerful language for declarative modeling of constraints. Appendix A.1 provides further details on the constraints language and its corresponding graphical notation that have been developed so far. The current composition portal does not implement the graphical

¹See for example [2, 3] for discussions on declarative vs. procedural process modeling styles.

²Note that declarative modeling in this case applies to control-flows. Data-flows are in general declarative.

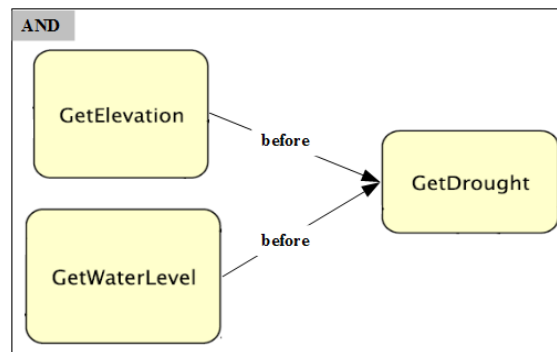


Figure 3.2.: Declarative specification of the control-flow of the process from Figure 3.1

notation for declarative modeling, however, the next release is planned to incorporate declarative modeling support in combination with the modeling style presented in the previous section. This will require a smooth alignment and integration of (a subset of) the BPMN notation with the declarative modeling notation. The current plan is consider the BPMN subset that consists of core elements for sequencing, non-determinism, parallelism, and looping, and extend it with the set of constraints that have been developed for the declarative modeling. This extension can be carried out through the extensibility mechanism provided by BPMN. The formal semantics of this extension is expected to be given in a formal logic, in a similar way as in [12].

4. Transformations from BPMN

Figure 4.1 shows the artefacts involved in the process from modeling the composition to the generation of WSDL and BPEL as the Web service artefacts. A customised and extended version of Oryx serves as the modeling environment in which the user specifies service compositions.

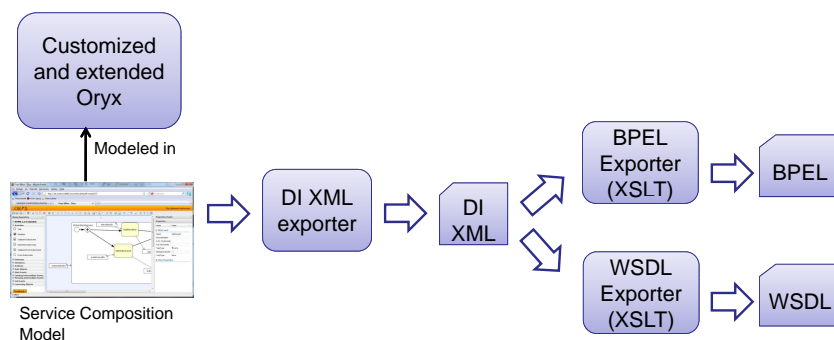


Figure 4.1.: From BPMN Modeling to Transformation of Web Service Artefacts

The standard Oryx implementation has an exporter that generates DI XML [9], which is the standard XML format for BPMN models. For each extension to the modeling environment of Oryx, we need to extend the DI XML exporter with the additional elements and properties. We have written XSLT code [15] that takes DI XML as input and produces BPEL and WSDL.

Unfortunately, the DI XML exporter has a few bugs which we have reported back to the Oryx open source community. Data flow associations between data objects are not exported and all objects inside a subProcess are duplicated. In our first versions of the transformation, we do not depend upon the data flow between data objects, since data transformations are written manually. The duplicated objects are removed manually as part of the generation process in our first version. In the next version we intend to either fix the DI XML exporter or to introduce an intermediate XSLT that fixes the problem.

4.1. BPMN to BPEL Transformation

The receive and reply activities of BPEL are not modeled as BPMN tasks, which is the most common way of modeling BPEL models in BPMN. Instead the receive and reply activities are implicitly captured by the input and output data objects of the outermost subProcess which represents the composed service.

The transformation identifies the start node within the outermost subProcess and follows the outgoing control flow. We generate appropriate BPEL code of the target node and continue the transformation by following its outgoing control flow to reach further nodes. In the mapping from BPMNs graph structure to the block structure of BPEL, we need to ensure

that the end of the same block is not generated multiple times. This is for instance the case when we generate BPEL flow from the BPMN parallel gateway. Our transformation handles this by passing a parameter to indicate if the outgoing control flow from a gateway is the 'last' one. The transformation ignores further processing of incoming control flow into a merge/join node except for the 'last' incoming control flow.

In the first version, we do not provide modeling and associated transformation support to handle data transformations. Instead we insert slots in the BPEL file to manually insert data transformation statements by BPELs assign/copy. Such a statement may also call XSLT code. The manual slots are inserted into BPEL right before a data object is consumed by a service.

4.1.1. The mapping from BPMN to BPEL

Figure 4.2 gives an overview of the mapping to BPEL, where each BPMN construct is mapped to one or more BPEL elements. For instance, a BPMN task construct is mapped to three corresponding BPEL elements: `invoke`, `import` and `partnerLink`.

BPMN	BPEL
subProcess	process receive reply import partnerLink
data object	variable
task	invoke import partnerLink
parallel gateway	flow

Figure 4.2.: Overview of the mapping from BPMN to BPEL

Below we give the details of the mapping to BPEL by listing the used properties and how they are mapped to BPEL. All the listed properties, except the name properties, were added as extensions to Oryx. These properties also constitute all the extensions we have added to our first release of the ENVISION-based Oryx.

4.1.1.1. BPMN element: *subProcess*

Mapped Properties: name, namespace, task type, port type, partner link type, wsdl

Mapping to BPEL:

- *process*. operation = (task type='wps' → execute, task type='wfs' → getFeature, task type='sos' → getObservation, task type='wcs' → getCoverage)
- *receive*. variable = the incoming data object of the subProcess, partnerLink = partner link type, portType = port type
- *reply*. variable = the outgoing data object of the subProcess, partnerLink = partner link type, portType = port type
- *import*. namespace = namespace, location = wsdl
- *partnerLink*. partnerLinkType = partner link type, myRole= name+'Role'

4.1.1.2. BPMN element: *Data Object*

Mapped Properties: name, type

Mapping to BPEL: *variable*. name = name, messageType = type

4.1.1.3. BPMN element: *Task*

Mapped Properties: name, namespace, task type, port type, partner link type, wsdl

Mapping to BPEL:

- *invoke*. partnerLink= name, portType = port type, operation = (task type='wps' → execute, task type='wfs' → getFeature, task type='sos' → getObservation, task type='wcs' → getCoverage), inputVariable = name of the BPEL variable mapped from the incoming data object, outputVariable = name of the BPEL variable mapped from the outgoing data object
- *import*. namespace = namespace, location = wsdl
- *partnerLink*. partnerLinkType = partner link type, partnerRole = name+'Role'

4.1.1.4. BPMN element: *Parallel Gateway*

Mapped Properties: < none >

Mapping to BPEL: *flow*

Figure 4.3 gives an overview of the mapping to BPEL for the simple use case example.

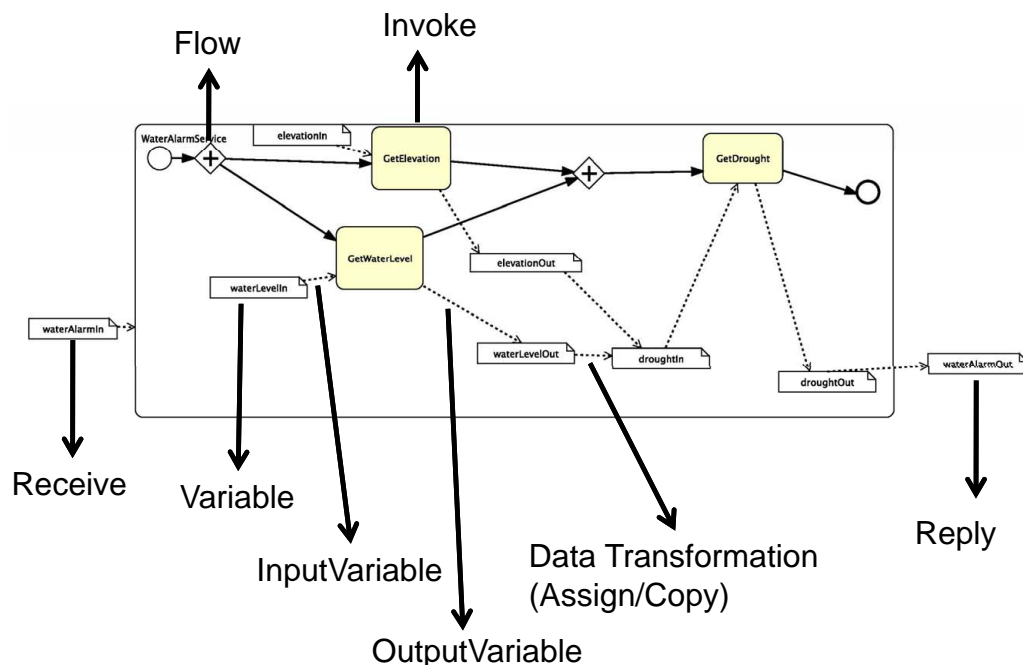
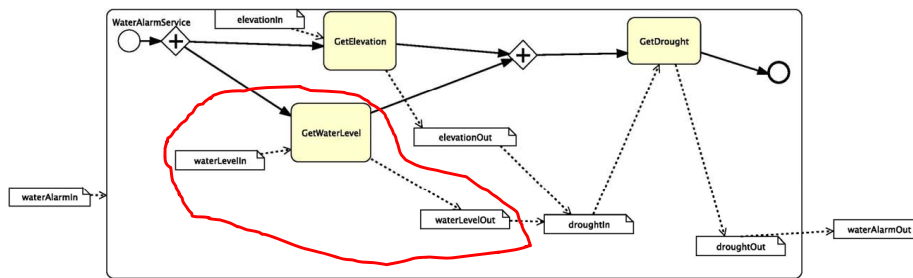


Figure 4.3.: Transformation from BPMN to BPEL

In Figure 4.4 we provide details about how XSLT (Figure 4.4(c)) works on DI XML files (Figure 4.4(b)) to produce BPEL (Figure 4.4(d)) for an extract of the simple use case



(a) Model Extract

```

<wfsTask wsdl="..depth.wsdl"
partnerLinkType="sosLinkType"
namespace="http://www.example.com/sos"
portType="wfsPortType"
name="GetWaterLevel" id="oryx_62...">
  <dataInputAssociation id="oryx_34...">
    <sourceRef>oryx_36.</sourceRef>
    <targetRef>oryx_629.</targetRef>
  </dataInputAssociation>
  <dataOutputAssociation id="oryx_C9...">
    <sourceRef>oryx_629.</sourceRef>
    <targetRef>oryx_60.</targetRef>
  </dataOutputAssociation>
</wfsTask>

<xsl:template name="makeInvoke">
  <xsl:param name="isLast"/>
  <xsl:param name="task"/>
  <xsl:param name="operation"/>
  <xsl:for-each select="$task">
    <invoke name="{@name}" partnerLink="{@name}"
portType="{@portType}" operation="{@operation}"
xmlns:tns="{@namespace}">
      <xsl:call-template name="inOutVariables">
        <xsl:with-param name="task" select="."/>
      </xsl:call-template>
    </invoke>
    <xsl:call-template name="nextNodes">
      <xsl:with-param name="startId" select="@id"/>
      <xsl:with-param name="isLast" select="$isLast"/>
    </xsl:call-template>
  </xsl:for-each>
</xsl:template>

```

(b) DI XML Extract

(c) XSLT Extract

```

<invoke portType="wfsPortType" name="GetWaterLevel" outputVariable="waterLevelOut"
partnerLink="GetWaterLevel" operation="getFeature" inputVariable="waterLevelIn"
xmlns:tns="http://www.example.com/sos"/>

```

(d) BPEL Extract

Figure 4.4.: Transformation from BPMN to BPEL by XSLT

example. The model extract (Figure 4.4(a)) includes the `GetWaterLevel` task and its input and output data objects. In DI XML the task is represented as a `wfsTask` element with two inner elements for the input and output data objects.

There are XSLT templates to match the different task types which then calls the `makeInvoke` template shown in Figure 4.4(c). The `makeInvoke` template is called with three parameters. The `isLast` parameter is the previously mentioned way to control that merge/join nodes are only handled once. The `operation` parameter provides the name of the operation based on the task type, e.g. `getFeature` for a task of type WFS. The `task` parameter holds the node of the task that we will map to an `invoke` statement in BPEL.

At the end of the `makeInvoke` template we continue the transformation by following the outgoing control flow, which is achieved by a call to the template `nextNodes`. Figure 4.4(d) shows the resulting BPEL `invoke` statement which corresponds to the `GetWaterLevel` BPMN task.

4.2. BPMN to WSDL Transformation

A WSDL file defines the set of operations that a service offers and the binding to a protocol to invoke the operations. Partner links can also be defined here and should be synchronized with those used in the BPEL definition of the corresponding implementation of an operation.

A composition model represents a new service which can be exposed by a WSDL file.

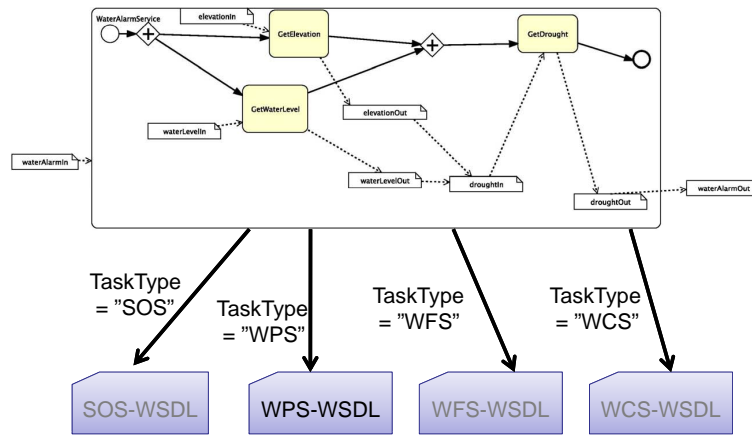


Figure 4.5.: Transformations from BPMN to WSDL

With the task type property we define what kind of OGC service the composite service represents. Every OGC service of the same type has a standardized, fixed set of operations. Hence, our transformation checks the task type of the outermost subProcess and generates a WSDL file accordingly.

Figure 4.5 illustrates the different task types we intend to support. At the current stage we have only implemented a transformation to a WPS-based WSDL, which is also without the binding section. Our plan is to extend the transformation with bindings, where we will make a default protocol choice that can be configured by the user.

5. Installation Guide

The MaaS Composition Portal is an extension of the Oryx framework. This framework supports graphical process modeling in several modeling languages including the commonly used BPMN. Oryx is a process platform - stakeholders can access process models via the web through a web browser which eliminates the need for additional local software installation. It is based on web technology which facilitates its integration in collaboration platforms like wikis and portals. Oryx is extensible - new functions can be added via a plug-in mechanism. It is based on open internet standards and the user authentication is done through OpenID.

To install the MaaS Composition Portal besides the distribution war files, it is necessary to have the Apache Tomcat application server and the PostgreSQL database. The following sections explain the required setup for Apache Tomcat and PostgreSQL and how to deploy the MaaS Composition Portal in Tomcat.

5.1. Installing and setting up PostgreSQL

In case it is not already there, PostgreSQL should be installed in the server where the MaaS Composition Portal is intended to be installed. The installation files are available in the PostgreSQL website (<http://www.postgresql.org>). In our development and testing installation we are using version 8.3.11. To execute the installation command, administrator privileges are needed. It is also essential to install PostgreSQL with PL/python support. For a detailed description of the installation procedure for your specific operative system please check <http://www.postgresql.org/docs/8.3/interactive/admin.html>

Then, the database should be configured such that it does not require passwords for local tcp connections. Traditionally these configurations are stored in a file called 'pg_hba.conf'. The configurations should be modified as follows: # Database administrative login by UNIX sockets

```
local all postgres ident sameuser
# TYPE DATABASE USER CIDR-ADDRESS METHOD
# "local" is for Unix domain socket connections only
local all trust
# IPv4 local connections:
host all 127.0.0.1/32 trust
# IPv6 local connections:
host all ::1/128 md5
```

After these changes the database should be started again.

The next step is to create a user called poem without password using the following command which should be executed by the user postgres:

```
createuser -superuser -echo poem
```

Afterwards, PostgreSQL's binary directory should be added to the PATH environment variable and the file 'db_schema.sql' should be copied to the server.

Then (still logged as the postgres user), the following commands should be executed:

```
createuser -U postgres -echo -pwprompt -encrypted poem
createdb -U postgres -echo -encoding utf8 -owner poem
psql -U postgres -dbname poem -file db_schema.sql
```

Now, the PostgreSQL database configuration process is completed with the following results: (i) there is a user called poem which can connect to the database without passwords and (ii) the database server trusts all the tcp local connections. This completes the PostgreSQL configuration requirements for executing the MaaS Composition Portal.

5.2. Installing Apache Tomcat

In case it is not already there, Apache Tomcat should also be installed in the server where the MaaS Composition Portal is intended to be installed. The installation files are available in the Apache Tomcat website (<http://tomcat.apache.org>). In our development and testing installation we are using version 5.0.28. There is no specific set up in Apache Tomcat needed in order to install the MaaS Composition Portal.

5.3. Installing the MaaS Composition Portal

The only thing needed to install the MaaS Composition Portal is copying its war files (backend.war and oryx.war) to Tomcat's deploying directory (which is by default the directory 'webapps').

5.4. Usage Manual and Screenshots

The MaaS Composition Portal can be accessed through the address:

```
http://set.sintef.no:8080/backend/poem/repository
```

Firstly, the user should authenticate himself by filling in the field in the right upper corner:

An OpenID user is required for logging in. This can be obtained at:

```
https://getopenid.com/
```

After logging in (Figure 5.1), one gets an overview of the already existing models (Figure 5.2)



Figure 5.1.: User login for the MaaS Composition Portal

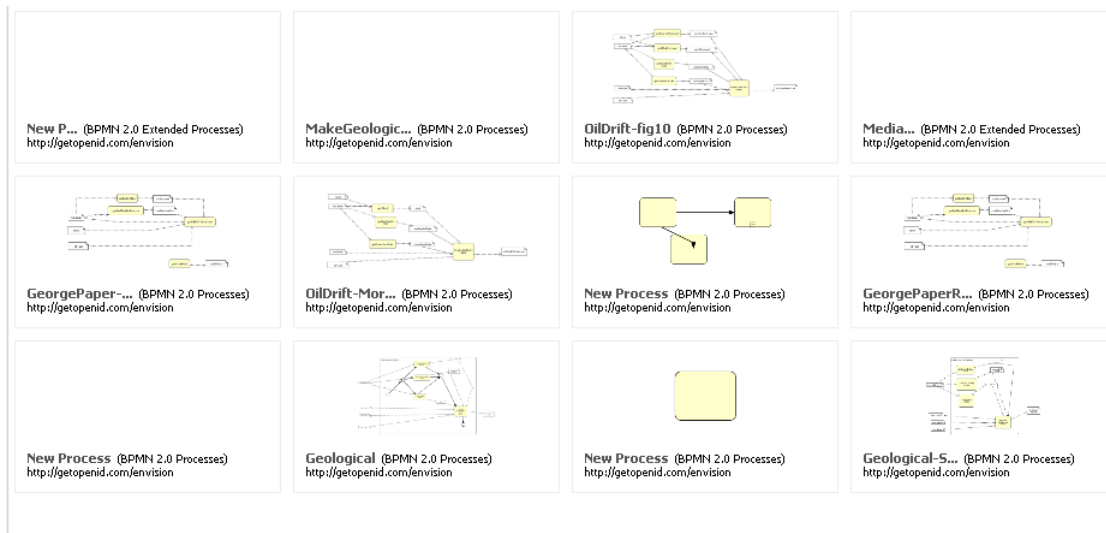


Figure 5.2.: Existing models

Our Oryx extensions in the MaaS Composition Portal are available in the 'BPMN 2.0 Extended Processes' models.

Figure 5.3 illustrates how to create these type of models:

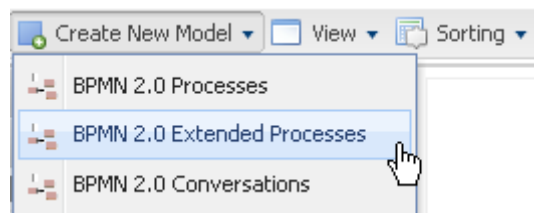
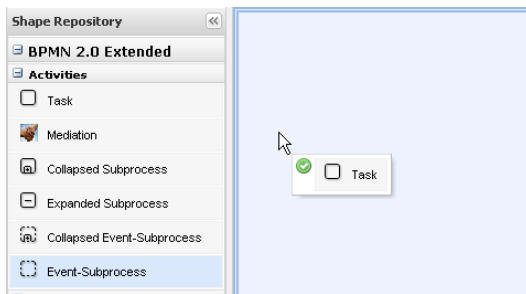


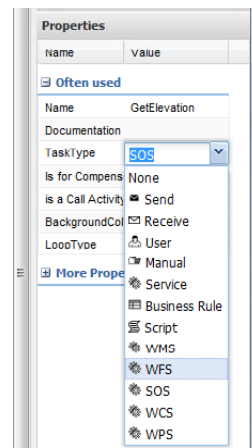
Figure 5.3.: Creating a new model

Adding elements to the model consists in selecting them from the Shape Repository and dragging them to the model. In Figure 5.4a, a task is being added to a model.

The elements in a model have properties which can be set when they are selected. Figure 5.4b shows some of the properties of a Task. Some of these properties, e.g. namespace, port type, are extensions to Oryx. We have extended subProcesses and tasks with some OGC service types (WFS, WCS, WPS, WMS, and SOS) for which we provide special support in ENVISION. The OGC service type is registered in the property called taskType.



(a) Adding a task to a model



(b) Properties of a Task

Figure 5.4.: Adding a task and assigning its properties

6. Summary and Features for Next Release

This deliverable provided an overview of the first release of the composition portal. The main achievements in this first release consist of (1) implementations of Oryx extensions to support OGC-specific and BPEL-specific properties, (2) transformation of BPMN compositions to executable BPEL processes and associated WSDL files, and (3) deployment of the extended Oryx system on a server to be used within ENVISION. This first release is functional and already available for internal use for the ENVISION consortium at <http://set.sintef.no:8080/backend/poem/repository>. So far the composition portal has been used to model compositions (and generate corresponding BPEL processes) related to the use cases.

The next release of the composition portal will be provided at M24. There are several directions in which the current release will be extended. One direction is the inclusion of data transformation specification in a composition. Tasks in a composition might have mismatching inputs/outputs, e.g. the output of one task is the input of another task, however the structure and semantics of the input and output can be different. Composition becomes thus an even more complex task when data heterogeneity is faced. In order for the composition to function correctly, a mapping between the mismatching data needs to take place during the design of the composition. Such a solution should be able to mediate the interchanged data part of the communication process and to keep the communication process transparent from the data representation point of view.

In ENVISION, the data passed between the services have a semantic representation. An ontological approach is thus proposed for data mediation, where communicating parties express data in terms of ontologies and interchanges messages are based on instances. The translation of instances from a source ontology to instances of a target ontology is done based on a set of mappings that are created at the design time. This phase includes the creation of mappings between the source and target ontology.

The design time mediation includes a graphical interface that allows the mediation expert to define the mappings, as shown in Figure 6.1. The mediation tool includes as well a set of algorithms that support the semi-automatic creation of mappings. Once the mappings are created they can be saved and formally represented using an abstract mapping language. The mappings are then saved in a persistent storage. The design time data mediation tool that will be used in the next release is based on the set of tools developed by UIBK in previous projects (DIP¹, SEEMP² and SemanticGov³).

Another direction for the next release of the composition portal is the inclusion of a more declarative modeling style for compositions as described in Section 3.2. This will imply a thorough analysis of the potential combination of procedural and declarative modeling styles and its realization as an extension to BPMN. This will then have to be reflected in

¹<http://dip.semanticweb.org/>

²<http://www.seemp.org/>

³<http://www.semantic-gov.org/>

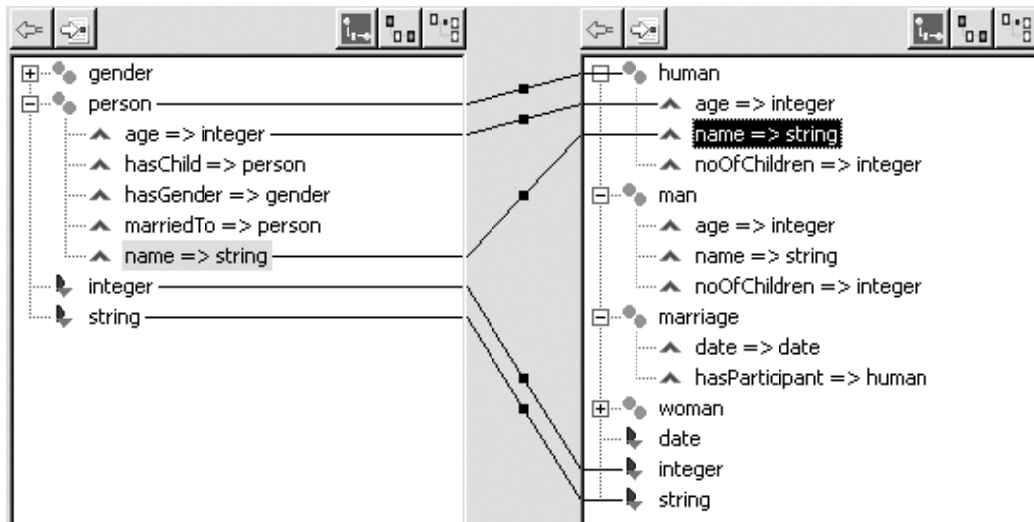


Figure 6.1.: Creating mappings by using the Design Time Data Mediation Tool

further extensions of the current composition portal, as well as an enhanced transformation of compositions to executable BPEL processes.

In addition, there are other more technical improvements that are planned for the next release, such as better visualization of task types (SOS, WPS, WFS, etc.), validation of control and data flows in compositions, and the integration of the composition portal with the resource module.

Bibliography

- [1] BPMI.org. Business Process Modeling Notation (BPMN) Version 1.0. <http://www.bpmn.org>, May 2004.
- [2] Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo A. Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. In *BMMDS/EMMSAD*, pages 353–366, 2009.
- [3] Dirk Fahland, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative vs. Imperative Process Modeling Languages: The Issue of Maintainability. In Bela Mutschler, Roel Wieringa, and Jan Recker, editors, *1st International Workshop on Empirical Research in Business Process Management (ER-BPM'09)*, pages 65–76, Ulm, Germany, September 2009. (LNBIP to appear).
- [4] Tracy Gardner. UML Modelling of Automated Business Processes with a Mapping to BPEL4WS. In *Proceedings of the First European Workshop on Object Orientation and Web Services at ECOOP*, 2003.
- [5] Roy Grønmo, Dumitru Roman, Alejandro Llaves, and Patrick Maué. Deliverable D3.1: MaaS Composition Portal Architecture Specification. <http://www.envision-project.eu/wp-content/uploads/2010/01/D3.1-FINAL.pdf>.
- [6] Diane Jordan and John Evdemon. Web Services Business Process Execution Language Version 2.0. Committee Specification. OASIS WS-BPEL TC., 2007.
- [7] Jana Koehler, Rainer Hauser, Shane Sendall, and Michael Wahler. Declarative techniques for model-driven business process integration. *IBM Systems Journal*, 44(1), 2005.
- [8] OGC. Open Geospatial Consortium. <http://www.opengis.org>.
- [9] OMG. Business Process Modeling Notation (BPMN) Version Beta 1 for version 2.0 - OMG Document Number: dtc/2009-08-14. <http://www.omg.org/spec/BPMN/2.0/>, August 2009.
- [10] Chun Ouyang, Marlon Dumas, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Jan Mendling. From business process models to process-oriented software systems. *ACM Trans. Softw. Eng. Methodol.*, 19(1), 2009.
- [11] PostgreSQL Global Development Group. PostgreSQL – The world’s most advanced open source database. <http://www.postgresql.org/>, 2010.
- [12] Dumitru Roman and Michael Kifer. Semantic web service choreography: Contracting and enactment. In *International Semantic Web Conference*, pages 550–566, 2008.

- [13] University of Potsdam, Germany. Oryx Editor - Web-based Graphical Business Process Editor. <http://code.google.com/p/oryx-editor/>.
- [14] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. <http://www.w3.org/TR/wsdl20>, August 2004.
- [15] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0. <http://www.w3.org/TR/xslt>, 1999.

A. Appendices

A.1. Declarative Modeling Notation

Declarative modeling style of compositions implies that compositions are specified as a set of relationships (also called constraints) that can exist between tasks. There are three types of elementary constraints — primitive, serial, and immediate serial constraints. Elementary constraints can be combined using conjunction and disjunction. The followings define in more detail the various types of constraints.

If a is a task then the following are **primitive** constraints:

- $existence(a, n)$ — task a must execute at least n times ($n \geq 1$):
- $absence(a)$ — task a must not execute
- $exactly(a, n)$ — task a must execute exactly n times ($n \geq 1$)

If a, b are tasks then the following are **serial** constraints:

- $after(a, b)$ — whenever a executes, b has to be executed after it. Task b does not have to execute immediately after a , and several other instances of a might execute before b does
- $before(a, b)$ — whenever b executes, it must be preceded by an execution of a . Task a does not have to execute immediately before b
- $blocks(a, b)$ — if a executes, b can no longer be executed in the future
- $between(a, b, a)$ — b must execute between any two executions of a , i.e., after an execution of a , any subsequent execution of a is blocked until b is executed
- $not_between(a, b, a)$ — b must not execute between any pair of executions of a . If b executes after a , no future execution of a is possible

If a, b are tasks then the following are **immediate serial** constraints:

- $right_after(a, b)$ — whenever a executes, b has to execute immediately after it
- $right_before(a, b)$ — whenever b executes, a must have been executed immediately before it
- $not_right_after(a, b)$ — whenever a and b execute, b must not execute immediately after a , i.e., between the execution of a and b there must be an execution of a task other than a and b

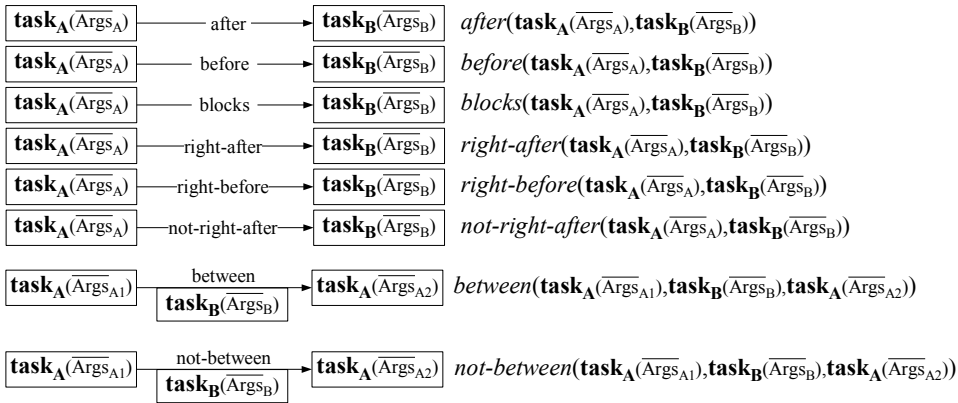
Furthermore, if C_1, C_2 are elementary constraints then so are $C_1 \wedge C_2$ (conjunction), and $C_1 \vee C_2$ (disjunction).

Figure A.1 provides an overview of the current notation for declarative modeling of processes, together with a textual equivalent for the various graphical elements.

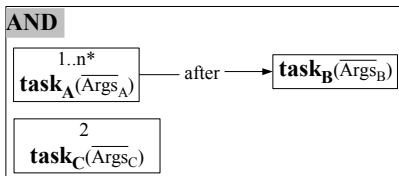
Primitive constraints

$n..*$ task ($\overline{\text{Args}}$)	$existence(\mathbf{task}(\overline{\text{Args}}),n)$
0 task ($\overline{\text{Args}}$)	$absence(\mathbf{task}(\overline{\text{Args}}))$
n task ($\overline{\text{Args}}$)	$exactly(\mathbf{task}(\overline{\text{Args}}),n)$
$0..n$ task ($\overline{\text{Args}}$)	$at-most(\mathbf{task}(\overline{\text{Args}}),n)$

Serial and immediate serial constraints

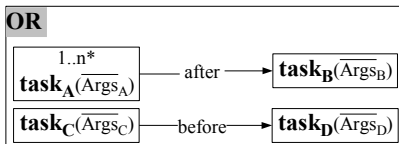


Composite constraints



Conjunctive composition

$$existence(\mathbf{task}_A(\overline{\text{Args}}_A),1) \wedge after(\mathbf{task}_A(\overline{\text{Args}}_A),\mathbf{task}_B(\overline{\text{Args}}_B)) \\ \wedge exactly(\mathbf{task}_C(\overline{\text{Args}}_C),2)$$



Disjunctive composition

$$(existence(\mathbf{task}_A(\overline{\text{Args}}_A),1) \wedge after(\mathbf{task}_A(\overline{\text{Args}}_A),\mathbf{task}_B(\overline{\text{Args}}_B))) \\ \vee before(\mathbf{task}_C(\overline{\text{Args}}_C),\mathbf{task}_D(\overline{\text{Args}}_D))$$

Figure A.1.: Declarative Modeling Notation.

A.2. XSLT Transformation: DI XML to BPEL

```

<?xml version='1.0' ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
  Transform"
  exclude-result-prefixes="bpel bpmn"
  xmlns:bpmn="http://schema.omg.org/spec/BPMN/2.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns0="http://www.opengis.net/sos/1.0"
  xmlns:ns1="http://www.opengis.net/wfs"
  xmlns:ns2="http://www.opengis.net/wps/1.0.0"
  xmlns:ns3="http://www.opengis.net/ows/1.1"
  xmlns:ns4="http://docs.oasis-open.org/wsbpel/2.0/process/
    executable">

<xsl:output indent="yes" method="xml"/>

<xsl:template match="*/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="text()|@*" />

<xsl:template match="bpmn:process/bpmn:subProcess">
  <process name="{@name}"
    targetNamespace="{@namespace}"
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/
      executable"
    xmlns:tns="{@namespace}">
    <xsl:call-template name="imports"/>
    <partnerLinks>
      <xsl:call-template name="partnerLinks">
        <xsl:with-param name="namespace" select="@namespace"/>
      </xsl:call-template>
    </partnerLinks>
    <variables>
      <xsl:call-template name="variables"/>
    </variables>
    <xsl:variable name="operation">
      <xsl:choose>
        <xsl:when test="@taskType = 'wps'">execute</xsl:when>
        <xsl:when test="@taskType = 'wfs'">getFeature</xsl:when>
        <xsl:when test="@taskType = 'sos'">getObservation</xsl:when>
        <xsl:when test="@taskType = 'wcs'">getCoverage</xsl:when>
      </xsl:choose>
    </xsl:variable>
    <sequence>
      <xsl:variable name="inputVarId" select="bpmn:dataInputAssociation/
        bpmn:sourceRef"/>
      <receive name="Receive" createInstance="yes" partnerLink="{
        @partnerLinkType}" xmlns:tns="{@namespace}" portType="tns:{
        @portType}">
      <xsl:attribute name="variable">

```

```

        <xsl:value-of select="// bpmn:dataObject[@id = $inputVarId ]/
            @name"/>
    </xsl:attribute>
    <xsl:attribute name="operation">
        <xsl:value-of select="$operation"/>
    </xsl:attribute>
</receive>
<xsl:apply-templates select="bpmn:startEvent"/>
<assign>
    <xsl:attribute name="name">
        <xsl:text>assignReplyVariable</xsl:text>
    </xsl:attribute>
    <xsl:comment>Manually insert the data transformation here by copy
        statements , xslt script etc.</xsl:comment>
</assign>
<xsl:variable name="outputVarId" select="bpmn:dataOutputAssociation
    /bpmn:targetRef"/>
<reply name="Reply" partnerLink="{ @partnerLinkType}" xmlns:tns="{
    @namespace}" portType="tns:{ @portType}">
    <xsl:attribute name="variable">
        <xsl:value-of select="// bpmn:dataObject[@id = $outputVarId ]/
            @name"/>
    </xsl:attribute>
    <xsl:attribute name="operation">
        <xsl:value-of select="$operation"/>
    </xsl:attribute>
</reply>
</sequence>
</process>
</xsl:template>

<!-- Import all WSDL files with associated namespace
    This is taken from the tasks and the data objects
    We need to ensure that each wsdl file occurs exactly once
    Each task needs to register a wsdl file for its service
-->

-->
<xsl:template name="imports">
    <xsl:for-each select="// bpmn:serviceTask | // bpmn:sosTask | //
        bpmn:wpsTask | // bpmn:wcsTask | // bpmn:wfsTask | // bpmn:process/
        bpmn:subProcess">
        <import importType="http://schemas.xmlsoap.org/wsdl/" namespace="{
            @namespace}" location="{ @wsdl}"/>
        <!-- If the namespace differs from partnerLinkNS , then make another
            import for that namespace -->
        <!-- <xsl:if test="not( @partnerLinkNS = @namespace) ">
            <import importType="http://schemas.xmlsoap.org/wsdl/" namespace="{
                @namespace}" location="{ @WSDL-NS}"/>
        </xsl:if>
        <import importType="http://schemas.xmlsoap.org/wsdl/" namespace="{
            @namespace}">
            <xsl:attribute name="location">
                <xsl:value-of select="substring( @wsdl"/>
                <xsl:text>Wrapper.wsdl</xsl:text>
            </xsl:attribute>
        </import> -->

```

envision

```
<!-- WORKING: We can remove the Partner Link Namespace property and
associated WSDL file from Oryx Tasks -->
</xsl:for-each>
</xsl:template>

<!-- We get one partnerLink per task and one for the new composite
service represented by the subProcess -->
<!-- Let the name of a partner link be the same as the name of the task
WORKING: Remove partnerlink attribute -->
<xsl:template name="partnerLinks">
  <xsl:param name="namespace"/>
  <xsl:for-each select="// bpmn:serviceTask | // bpmn:sosTask | //
    bpmn:wpsTask | // bpmn:wcsTask | // bpmn:wfsTask | // bpmn:process /
    bpmn:subProcess">
    <partnerLink name="{@name}" partnerLinkType="tns:{@partnerLinkType}"
      xmlns:tns="{ $namespace }">
      <xsl:choose>
        <xsl:when test="name() = 'subProcess' ">
          <xsl:attribute name="myRole">
            <xsl:value-of select="@name"/>
            <xsl:text>Role</xsl:text>
          </xsl:attribute>
        </xsl:when>
        <xsl:otherwise>
          <xsl:attribute name="partnerRole">
            <xsl:value-of select="@name"/>
            <xsl:text>Role</xsl:text>
          </xsl:attribute>
        </xsl:otherwise>
      </xsl:choose>
    </partnerLink>
  </xsl:for-each>
</xsl:template>

<!-- CHECKS IF THE DATA OBJECT IS AN INPUT OR OUTPUT OF A TASK AND USE
THE TASKS NAMESPACE VALUE -->
<xsl:template name="variables">
  <xsl:for-each select="// bpmn:dataObject">
    <xsl:variable name="dataObjId" select="@id"/>
    <xsl:variable name="dataObjName" select="@name"/>
    <xsl:variable name="dataObjType" select="@type"/>
    <!-- Two for-each loops where only one occurrence in one of them will
generate output:
Either the dataObject is an input object or it is an output
object -->
    <!-- Maybe an input variable -->
    <xsl:for-each select="//*/ bpmn:dataInputAssociation[ bpmn:sourceRef =
      $dataObjId ]">
      <variable name="{ $dataObjName }" messageType="tns: { $dataObjType }"
        xmlns:tns=" { ../ @namespace }"/>
    </xsl:for-each>
    <!-- Maybe an output variable -->
    <xsl:for-each select="//*/ bpmn:dataOutputAssociation[ bpmn:targetRef =
      $dataObjId ]">
      <variable name="{ $dataObjName }" messageType="tns: { $dataObjType }"
        xmlns:tns=" { ../ @namespace }"/>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>
```

```

</xsl:for-each>
</xsl:template>
<xsl:template match="bpmn:startEvent">
  <!-- Follow the control flow and check the next node type -->
  <xsl:variable name="startId" select="@id"/>
  <xsl:variable name="outFlow" select="// bpmn:sequenceFlow [ @sourceRef=$
    startId ]"/>
  <xsl:apply-templates select="//*[ @id = $outFlow/@targetRef]"/>
</xsl:template>
<xsl:template match="bpmn:parallelGateway[ @gatewayDirection='diverging ']"
  >
  <xsl:param name="isLast"/>
  <!-- start tag: flow -->
  <xsl:text>&#xa;</xsl:text>
  <!-- <xsl:text disable-output-escaping='yes'>&lt;flow&gt;&lt;sequence&
    gt;</xsl:text> -->
  <xsl:text disable-output-escaping='yes'>&lt;flow&gt;</xsl:text>

  <!-- make one sequence for each outgoing control flow -->
  <!-- Follow the control flow and check the next node type -->
  <xsl:variable name="startId" select="@id"/>
  <xsl:variable name="outFlow" select="// bpmn:sequenceFlow [ @sourceRef=$
    startId ]"/>

  <xsl:for-each select="//*[ @id = $outFlow/@targetRef]">
    <xsl:text disable-output-escaping='yes'>&lt;sequence&gt;</xsl:text>
    <xsl:apply-templates select=".">
      <xsl:with-param name="isLast" select="position() = last()"/>
    </xsl:apply-templates>
    <xsl:if test="not(position() = last())">
      <xsl:text disable-output-escaping='yes'>&lt;sequence&gt;</
        xsl:text>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
<xsl:template match="bpmn:parallelGateway[ @gatewayDirection='converging ']"
  >
  <xsl:param name="isLast"/>

  <!-- Do only proceed if this is the last incoming control flow
    to the converging parallel -->
  <xsl:if test="$isLast = 'true'">
    <!-- end tag: flow -->
    <xsl:text>&#xa;</xsl:text>
  <!-- <xsl:text disable-output-escaping='yes'>&lt;sequence&gt;&lt;
    ;&lt;flow&gt;</xsl:text> -->
  <xsl:text disable-output-escaping='yes'>&lt;sequence&gt;&lt;
    ;&lt;flow&gt;</xsl:text>

  <xsl:call-template name="nextNodes">
    <xsl:with-param name="startId" select="@id"/>
    <xsl:with-param name="isLast" select="$isLast"/>
  </xsl:call-template>
</xsl:if>

```

```

</xsl:template>

<!-- Make invoke element for serviceTask, wpsTask, sosTask etc. -->
<xsl:template name="makeInvoke">
  <xsl:param name="isLast"/>
  <xsl:param name="task"/>
  <xsl:param name="operation"/>

  <xsl:call-template name="assignSkeleton">
    <xsl:with-param name="name" select="@name"/>
  </xsl:call-template>
  <xsl:for-each select="$task">
    <invoke name="{@name}" partnerLink="{@name}" portType="{@portType}"
      operation="{ $operation }" xmlns:tns="{@namespace}">
      <xsl:call-template name="inOutVariables">
        <xsl:with-param name="task" select="."/>
      </xsl:call-template>
    </invoke>
    <xsl:call-template name="nextNodes">
      <xsl:with-param name="startId" select="@id"/>
      <xsl:with-param name="isLast" select="$isLast"/>
    </xsl:call-template>
  </xsl:for-each>
</xsl:template>

<xsl:template match="bpmn:serviceTask">
  <xsl:param name="isLast"/>
  <xsl:call-template name="makeInvoke">
    <xsl:with-param name="isLast" select="$isLast"/>
    <xsl:with-param name="task" select="."/>
    <xsl:with-param name="operation">
      <xsl:value-of select="@operationRef"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template match="bpmn:sosTask">
  <xsl:param name="isLast"/>
  <xsl:call-template name="makeInvoke">
    <xsl:with-param name="isLast" select="$isLast"/>
    <xsl:with-param name="task" select="."/>
    <xsl:with-param name="operation">getObservation</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template match="bpmn:wpsTask">
  <xsl:param name="isLast"/>
  <xsl:call-template name="makeInvoke">
    <xsl:with-param name="isLast" select="$isLast"/>
    <xsl:with-param name="task" select="."/>
    <xsl:with-param name="operation">execute</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template match="bpmn:wcsTask">
  <xsl:param name="isLast"/>
  <xsl:call-template name="makeInvoke">
    <xsl:with-param name="isLast" select="$isLast"/>

```

```

    <xsl:with-param name="task" select="."/>
    <xsl:with-param name="operation">getCoverage</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template match="bpmn:wfsTask">
  <xsl:param name="isLast"/>
  <xsl:call-template name="makeInvoke">
    <xsl:with-param name="isLast" select="$isLast"/>
    <xsl:with-param name="task" select="."/>
    <xsl:with-param name="operation">getFeature</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template name="assignSkeleton">
  <xsl:param name="name"/>
  <assign>
    <xsl:attribute name="name">
      <xsl:text>assignInput_</xsl:text>
      <xsl:value-of select="$name"/>
    </xsl:attribute>
    <xsl:comment>Manually insert the data transformation here by copy
      statements, xslt script etc.</xsl:comment>
  </assign>
</xsl:template>

<!-- generate inputVariable and outputVariable for an invoke task -->
<xsl:template name="inOutVariables">
  <xsl:param name="task"/>
  <xsl:attribute name="inputVariable">
    <xsl:value-of select="// bpmn:dataObject[@id = $task /
      bpmn:dataInputAssociation / bpmn:sourceRef] / @name" />
  </xsl:attribute>
  <xsl:attribute name="outputVariable">
    <xsl:value-of select="// bpmn:dataObject[@id = $task /
      bpmn:dataOutputAssociation / bpmn:targetRef] / @name" />
  </xsl:attribute>
</xsl:template>

<!-- Follow the control flow and check the next node type -->
<xsl:template name="nextNodes">
  <xsl:param name="startId"/>
  <xsl:param name="isLast"/>
  <xsl:variable name="outFlow" select="// bpmn:sequenceFlow[ @sourceRef=$
    startId]" />
  <xsl:apply-templates select="// *[ @id = $outFlow / @targetRef]">
    <xsl:with-param name="isLast" select="$isLast" />
  </xsl:apply-templates>
</xsl:template>

</xsl:stylesheet>

```

A.3. XSLT Transformation: DI XML to WSDL

```

<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  exclude-result-prefixes="bpmn"
  xmlns:bpmn="http://schema.omg.org/spec/BPMN/2.0"
  xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
  xmlns="http://schemas.xmlsoap.org/wsd/"
  xmlns:wsd="http://schemas.xmlsoap.org/wsd/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns0="http://www.opengis.net/gml/3.2"
  xmlns:ns="http://www.opengis.net/wps/1.0.0"
  xmlns:ns2="http://www.example.com/sos">

<xsl:output indent="yes" method="xml"/>

<xsl:template match="*/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="text()|@*" />

<!-- Generates different WSDL file
      according to the task type: SOS, WPS, WFS, etc. -->

<xsl:template match="bpmn:process/bpmn:subProcess">
  <definitions name="{@name}" xmlns:tns="{@namespace}"
    targetNamespace="{@namespace}" >
    <xsl:call-template name="imports"/>
    <xsl:call-template name="types"/>
    <xsl:call-template name="messages"/>
    <xsl:call-template name="portTypes"/>
    <xsl:call-template name="partnerLinkTypes"/>
  </definitions>
</xsl:template>

<xsl:template name="imports">
  <xsl:for-each select="//bpmn:serviceTask | //bpmn:sosTask
    | //bpmn:wpsTask | //bpmn:wcsTask
    | //bpmn:wfsTask">
    <import importType="http://schemas.xmlsoap.org/wsd/"
      namespace="{@namespace}" location="{@wsdl}"/>
    <!-- If the namespace differs from partnerLinkNS,
          then make another import for that namespace -->
    <!-- <xsl:if test="not(@partnerLinkNS = @namespace)">
      <import importType="http://schemas.xmlsoap.org/wsd/"
        namespace="{@namespace}" location="{@WSDL-NS}"/>
    </xsl:if> -->
  </xsl:for-each>
</xsl:template>

<xsl:template name="types">
  <types>

```

```

<xsd:schema targetNamespace="{ @namespace }">
  <xsd:import namespace="http://www.opengis.net/gml/3.2"
    schemaLocation="http://schemas.opengis.net/gml/3.2.1/coverage.
      xsd"/>
  <xsd:import namespace="http://www.opengis.net/wps/1.0.0"
    schemaLocation="http://schemas.opengis.net/wps/1.0.0/
      wpsExecute_request.xsd"/>
  <xsd:import namespace="http://www.opengis.net/wps/1.0.0"
    schemaLocation="http://schemas.opengis.net/wps/1.0.0/
      wpsGetCapabilities_response.xsd"/>
  <xsd:import namespace="http://www.opengis.net/wps/1.0.0"
    schemaLocation="http://schemas.opengis.net/wps/1.0.0/
      wpsGetCapabilities_request.xsd"/>
  <xsd:import namespace="http://www.opengis.net/wps/1.0.0"
    schemaLocation="http://schemas.opengis.net/wps/1.0.0/
      wpsDescribeProcess_request.xsd"/>
  <xsd:import namespace="http://www.opengis.net/wps/1.0.0"
    schemaLocation="http://schemas.opengis.net/wps/1.0.0/
      wpsDescribeProcess_response.xsd"/>
</xsd:schema>
</types>
</xsl:template>

<!-- Different set of messages
  depending on the type of service: SOS, WPS etc. -->
<xsl:template name="messages">
  <xsl:choose>
    <xsl:when test="@taskType = 'wps'">
      <message name="executeRequest">
        <part name="part1" type="ns:ComplexDataType"/>
      </message>
      <message name="executeResponse">
        <part name="part1" element="ns0:GridCoverage"/>
      </message>
      <message name="getCapabilitiesRequest">
        <part name="capabilityReq" element="ns:GetCapabilities"/>
      </message>
      <message name="getCapabilitiesResponse">
        <part name="capabilityRes" element="ns:Capabilities"/>
      </message>
      <message name="describeProcessReq">
        <part name="processReq" element="ns:DescribeProcess"/>
      </message>
      <message>
        <xsl:attribute name="name">
          <xsl:value-of select="@name"/>
          <xsl:text>WSDLOperationResponse</xsl:text>
        </xsl:attribute>
        <part name="part1" element="ns:ProcessDescriptions"/>
      </message>
    </xsl:when>
  </xsl:choose>
</xsl:template>

<!-- Different set of operations
  depending on the type of service: SOS, WPS etc. -->
<xsl:template name="portTypes">
  <portType>

```

```

<xsl:attribute name="name">
  <xsl:value-of select="@name"/>
  <xsl:text>_Interface</xsl:text>
</xsl:attribute>
<xsl:choose>
  <xsl:when test="@taskType = 'wps'">
    <operation name="execute">
      <input name="input1" message="tns:executeRequest"/>
      <output name="output1" message="tns:executeResponse"/>
    </operation>
    <operation name="getCapabilities">
      <input name="input2" message="tns:getCapabilitiesRequest"/>
      <output name="output2" message="tns:getCapabilitiesResponse"/>
    </operation>
    <operation name="describeProcess">
      <input name="input3" message="tns:describeProcessReq"/>
      <output name="output3">
        <xsl:attribute name="message">
          <xsl:text>tns:</xsl:text>
          <xsl:value-of select="@name"/>
          <xsl:text>WSDLOperationResponse</xsl:text>
        </xsl:attribute>
      </output>
    </operation>
  </xsl:when>
</xsl:choose>
</portType>
</xsl:template>

<xsl:template name="partnerLinkTypes">
  <xsl:comment>A partner link type is automatically generated
    when a new port type is added. Partner link types are used by
    BPEL processes.
    In a BPEL process, a partner link represents the interaction
    between the BPEL process and a partner service.
    Each partner link is associated with a partner link type.
    A partner link type characterizes the conversational
    relationship
    between two services. The partner link type can have one or two
    roles.
  </xsl:comment>

  <xsl:for-each select="// bpmn:process/bpmn:subProcess">
    <plnk:partnerLinkType name="{ @partnerLinkType }">
      <plnk:role name="{ @name }Role" portType="ptns: { @portType }"
        xmlns:ptns="{ @namespace }"/>
    </plnk:partnerLinkType>
  </xsl:for-each>

  <xsl:for-each select="// bpmn:serviceTask | // bpmn:sosTask
    | // bpmn:wpsTask | // bpmn:wcsTask
    | // bpmn:wfsTask">
    <plnk:partnerLinkType name="{ @partnerLinkType }">
      <plnk:role name="{ @name }Role" portType="ptns: { @portType }"
        xmlns:ptns="{ @namespace }"/>
    </plnk:partnerLinkType>
  </xsl:for-each>

```

```
</xsl:for-each>  
</xsl:template>  
</xsl:stylesheet>
```