Project Number: **215219**

Project Acronym: **SOA4All**

Project Title: **Service Oriented Architectures for All**

Instrument: **Integrated Project**

Thematic Priority: **Information and Communication Technologies**

D6.3.3 Evaluation and Final Design of the Lightweight Context-Aware Process Modelling Language

Activity N:	Activity 2 – Core R&D activities	
Work Package:	WP6 – Service Construction	
Due Date:		M30
Submission Date:		31/08/2010
Start Date of Project:		01/03/2008
Duration of Project:		36 Months
Organisation Responsible of Deliverable:		SAP
Revision:		1.0
Author(s):	Patrick Un	SAP
	Pavel Genevski	SAP
	Yosu Gorroñogoitia	ATOS
	Mateusz Radzimski	ATOS
	Gianluca Ripa	CEFRIEL
	Adrian Mos	INRIA
	Barry Norton	UKARL
	Freddy Lecue	UNIMAN
Reviewer(s)	Sven Abels	TIE
	Daniel Winkler	UIBK

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**X**

PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
01	28/05/2010	TOC drafted and a drafted structure of the document is created with description of content	Patrick Un, Pavel Genevski (SAP)
02	01/06/2010	TOC content and structure updated	Patrick Un, Pavel Genevski (SAP)
03	01/07/2010	Structural update	Patrick Un (SAP)
04	30/07/2010	Integrated content	Patrick Un, Pavel Genevski (SAP)
05	01/08/2010	Merged contributions	Yosu Gorroñogoitia, Mateusz Radzimski (ATOS), Freddy Lecue (UNIMAN), Adrian Mos (INRIA)
06	09/08/2010	Further writing	Patrick Un (SAP)
07	17/08/2010	Peer review version	Patrick Un, Pavel Genevski (SAP)
08	24/08/2010	Update on dataflow and conditions	Barry Norton (UKARL)
09	25/08/2010	Integration of reviewers' feedback	Patrick Un, Pavel Genevski (SAP)
10	26/08/2010	Given to final proofreading	Patrick Un (SAP)

Table of Contents

List of Figures

List of Tables

Glossary of Acronyms

Acronym	Definition
API	Application Programming Interface
BPEL	Business Process Execution Language
BPMN	Business Process Modelling Notation
BWW	Bunge-Wand-Weber Framework
D	Deliverable
DTC	Design Time Composer of the Service Construction Platform
EAI	Enterprise Application Integration
EC	European Commission
EPC	Event Process Chain
JAXB	Java Architecture for XML Binding
JOS	Java Object Serialization
LPML	Lightweight Context-aware Process Modelling Language
MSM	Minimal Service Model
PE	Process Editor of SOA4All Studio
POSM	Procedure Oriented Service Model
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Query Language for RDF 1.0
URI	Universal Resource Identifier
URL	Universal Resource Locator
WP	Work Package
WSML	Web Service Modelling Language
WSMO	Web Service Modelling Ontology
WSMO-Lite	Web Service Modelling Ontology-Lite
XML	Extensible Markup Language
XSD	XML Schema Language
XStream	XStream Object-XML Serialization API

Executive Summary

The Lightweight Process Modelling Language (LPML) is devised to provide a lightweight methodology and a set of process modelling vocabularies to aid non-experienced users to get on with their tasks quickly. It is the SOA4All language for process modelling and is used in the entire project Users can create graphical process models using the Process Editor (T2.6). These models are enhanced by the WP6 components such as the Design Time Composer (DTC) and Optimizer (T6.4). Finally the enhanced process models are transformed into executable processes and executed by the Execution Engine (T6.5).

This deliverable provides the final design of the LPML which is based on the last version of the language as described in the deliverable D6.3.2. The fundamental design principles focus on two major aspects. First, we preserve the language elements such as process activities, gateways, control-flow constructs, etc. due to backward compatibility concern. Second, we strive to enhance interoperability by adding further API components such as loops and loop-handling functionality, a better serialization and de-serialization mechanism. Semantic modeling and execution of dataflow has been enhanced using decidable conditions which have been implemented in a language-agnostic way to optimize interoperability. These aspects reflect a more elaborate design of the language and are viewed as response to the continuous WP6 process integration task and service construction needs.

1. Introduction

1.1 Purpose and Scope

LPML has become an indispensable part of the SOA4All Service Construction facility, the core to lightweight process abstraction, process representation, annotation, instantiation, persistence and facilitating medium for process optimization and executions. The SOA4All Studio and its respective platform components of different work packages use LMPL extensively. Non-experienced end users are able to compose services via connecting the activities and defining their control-flows and dataflow using visual LPML conceptualization elements of the SOA4All Process Editor within the SOA4All Studio, thereby creating their business processes using the canonical LPML vocabulary on the language level.

Since LPML is a common effort on the language level to assist the SOA4All Service Construction Platform services that exchange process models using the provisions of the language, it is the correct medium to provide a process vocabulary to link the modellers and the SOA4All tools as well as the Service Construction Platform services together. It is subjected to continuous scrutiny, changes and an evolutionary improvement processes in alignment with the further development of the different studio and platform components. This deliverable will provide an overview and description of the final design and specification of LPML programmatic facilities from the language perspective. We will describe the components of the current API that are aligned with the updated requirements of the Process Editor (D2.6.3), Design Time Composer (D6.4.2, D6.4.3), Optimizer (D6.4.3) and Execution Engine (D6.5.3). We describe the additional features of the current API such as looping in activity, LPML process serialization and de-serialization that makes persistence of processes more flexible; and semantic modeling and execution of dataflow and conditions using SPARQL queries.

Technically the LPML API has been evolved and geared toward optimally aligning the needs of process modelling in a flexible manner, i.e. creation of correct and complete process models that are valid and lightweight from the non-experienced end users' perspective. This means that conventional heavier weight process modelling language such as Business Process Modelling Notation (BPMN) or Event Process Chain (EPC), though could be more expressive and provide more process modelling premises, also incur the drawback of these points: they include a superset of modelling constructs that are either seldom used by non-technical users because of their arcane or incomprehensible usage or due to over-empowerment, i.e., they confer to the modellers a full range of expressive power using these premises that there is no guarantee of the correctness of the process model. With conscious judgments of these drawbacks, the LMPL API is further developed and consolidated to suit the actual need of the SOA4All platform. It tries to retain the minimal expressive set of premises that allow real-world modelling without compromising correctness. This deliverable gives the technical evaluation and discussions about these aspects.

1.2 Structure of the Document

This document is organized in four main parts. Section 1 is an introductory exposition of the deliverable. Section 2 elaborates on the final design of the LPML language API as well as a series of additional features that make the language more suitable for the project. Section 3 evaluates the technicalities from an expressiveness and correctness perspective of the current language design. The final section concludes this document.

1.3 Technical Deliverable Introduction

The main technical background of LPML comprises of characteristics of different aspects of the language that is created for the sole purpose of lightweight process modelling. This principles guide through the technical design of the language and can be enumerated mainly in the following aspects:

- Graphical abstraction of process model and coherent encapsulation of model details
- Semantic annotations of process elements and contextualization
- Process patterns and templates
- Dataflow aspects and data connectors

These following sections of the deliverable elaborate more on these aspects with regard to current final design of LPML.

2. Final Design of Lightweight Context-Aware Process Modelling

The aim of LPML (Schnabel, 2008) is to simplify the work of a process designer by hiding tedious modelling and programming chores from users, performing automatic compositions and allowing for the late binding to concrete semantically annotated services that are bound at runtime. LPML is devised taking into account both the LPML usability in the tool that will be provided to the user and the underlying design process. In this section, we will give an insight into the current design of LPML and how the design principles of lightweight modelling are reflected in enhancing user modelling and designing processes. However, these enhancements are not visualized in the process model view. Activities have to be instantiated by services. Further the tools help to include conditions for gateways and those flows connected to a gateway based on the semantic descriptions given by the user. In the following we will cover various aspects of LPML to illustrate the design principles such as semantic annotations, process patterns and templates, aided semantic service discovery, dataflow mapping, process composition and optimization.
As an evolutionary effort for the service construction platform, the final design of LPML has been enhanced with a concentration on seamless integration with the other service construction components of SOA4All for scalability and stability; additionally other newer features such as loops and dataflow elements address further expressiveness issues of the process language. Scalability requirements of modelling dataflow are mainly addressed by dynamic generative approach of dataflow query using standard open language such as SPARQL which is used to dynamically extract dataflow mapping information and instances at runtime in interaction with backend components. This ensures better integration with other SOA4All runtime components.

2.1 LPML Metamodel and Language Elements

In the following section, technical aspects of LPML will be explicitly described in its design that addresses a series of technical requirements described in the introduction. Regarding the abstraction of process modelling approach, conceptualization of the underlying principles of LPML to abstractly represent lightweight processes is mainly comprises of two abstraction layers. The scheme comprising the graphical abstraction layer and the canonical LPML metamodel layer is shown in Figure 1. The graphical abstraction layer represents the user interface that is used by end users to create a process and assigning activities to concrete services. It is realized by the implementation of the SOA4All Process Editor (D2.6.2). The LPML metamodel layer is the canonical representation of the business process that is semi-automatically created by the end users assisted by the abstract graphical process layer studio tools, mainly the SOA4All Process Editor. The user can create and edit processes through their interaction with the user interface and thus indirectly instantiating, manipulating the canonical representation of the LPML metamodel. In order to complete the model and make it executable, the user can specify and provide the necessary services details to the model by binding them to activities of the process.

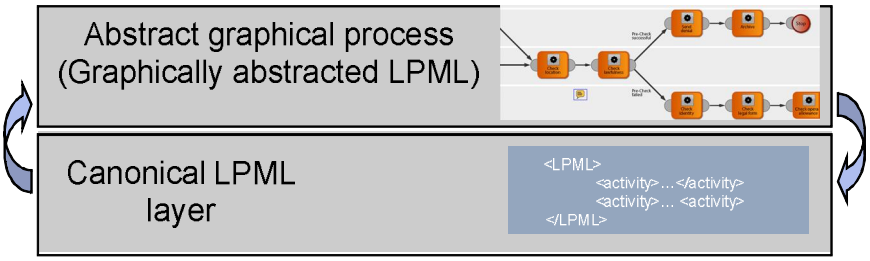


Figure 1: Graphical abstaction and canonical representation of LPML

LPML allows and encourages users to use semantic annotations to enhance a process model by providing semantics or business meaning. LPML caters in this regard by providing the necessary constructs in the model to allow user to specify and attach information of knowledge representation models such as domain ontology and ontology concept instances via references to the process elements. Ontology concept instances are metadata that can be added to the process elements such that these annotations are useful for a variety of purposes. For instance, annotated semantic services can be referenced and attached by instantiating service elements and attaching them to the process activities in the process model. By using SOA4All service discovery components, suitable semantic services can be found and bound to process activities. This way, LPML provides the necessary premises to facilitate service discovery. Moreover by making combination of process activities possible, LPML ensures that composition of services on the canonical language level is guaranteed. By using semantic annotations and constraints on the respective process elements, different conditions and constraints can be checked in this composition process. Within SOA4All, such annotations are drawn from the Web Service Modelling Ontology-Lite (WSMO-Lite) ontology. On the other hand, business users can use annotations to further describe the scope of applicable circumstances, for instance, also in form of semantic annotations and constraints to specify under which context or with what limitations a certain process and its elements can be interpreted and instantiated.

In resemblance to software patterns, process patterns allow effective summarization and communication of process knowledge. Some patterns such as the workflow process patterns have semantics that are proven to be complete, correct and conflict-free. They represent high degree of abstraction of knowledge fragments that are reusable. As shown in (Schnabel, 2008), we have adopted a series of patterns in the design of LPML. We further introduce workflow templates that represent coarse-grained combinable workflow patterns that are functionally and syntactically sound. Workflow templates contain start and end activities and are non-executable fragments of a process with unbound activities and certain unset information e.g. flow conditions. They can be regarded as abstract process because they require an instantiation with concrete missing information and bound activities. Workflow patterns do not possess start and end activities and are also non-executable. They cover the common reusable and recurrent portions of a process which can be reused in many different processes across different scenarios and domains. Further integration has led to the mixed use of workflow patterns and templates represented by LPML canonical representation and details can be found in (Gorronogoitia, 2010).

The idea of dataflow connectors which are introduced in the language is instrumental in order to overlay a layer of dataflow manipulation operations on top of the control-flow based process model. With dataflow oriented language constructs, LPML allows users, via the

graphical layer, to create mash-up-based service compositions and connect the dataflow with a list of operators as described in the previous language specification (Schnabel 2008). These operators allow data operations to be performed on the data passing from one activity to another. A mandatory mapping of both semantic and syntactic nature can be specified and represented in the canonical language model. Further development in the area of dataflow has shown that by using executable SPARQL query expressions, to semantically manipulate messages to effect pre-annotated dataflow schemes, mapping can thus be further simplified and operation effectiveness can be enhanced. We use this approach to simplify data operations by externalizing the tasks towards reusable semantic machinery in order to compute the necessary syntactic data transformation operations.

2.1.1 Application Programming Interface

The concept of abstractions in LPML is described in the previous section. This section provides detailed descriptions of the LPML API. It is used to programmatically create an in-memory process model that conforms to the abstract metamodel of LPML. Such a model has the advantages of facilitating interaction between the components, users as well as between the users and the process editor. In addition it can be serialized either in XML or directly as serializable Java object that can be stored or exchanged easily. Additional features that address iterative processing issues are integrated into the final design of LPML and will be described in detail in section 2.3. We described the mechanism of dataflow mapping support in section 2.5. Finally, better integration of other interoperable standard modelling premises is supported by allowing generating the LPML model directly from BPMN. This can either be imported into the process editor or processed directly by service construction platform components. This is described in detail in section 2.6.

2.1.2 API Elements

In the development of the design, it is apparent that in order to gain compatibility of the metamodel of the previous version, the LPML API programming model is going to retain certain API elements that are general to both versions. These elements are indispensable in creating a valid process model. Moreover, in the development of LPML, we integrate some additional features into the language that enhances usability and scalability. This has led to changes in the API by adding and removing of some elements since the last version. The following section describes the current status.

The principles of good software engineering practices, such as reducing coupleness and enhancing cohesive design, have been followed already in the last version of LPML and it guides the final design here as well. It consists of mainly Java interfaces and implementation classes that are described in the following.

Identifiable

Figure 2 shows an *Identifiable* interface which is used to factor out the canonical representation of the uniquely identifiable characteristics of LPML elements that extend it. Regarding compatibility with the previous version of LPML, in this way it can be guaranteed that all process elements will have an identifier when it is extended by an additional interface or implemented by subclasses where its attributes are inherited. Its characteristics are described in Table 1.

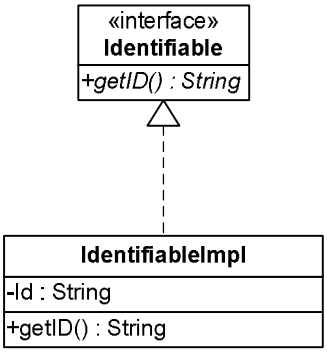


Figure 2: Identifiable Interface

Identifiable	
Attributes	ID
Extending interface	N/A
Serialization	Serializable or via stream
Concrete implementations	*IdentifiableImpl, ProcessElement*

Table 1: Identifiable Interface Description

Annotatable

Attaching additional semantics beside textual description helps to enhance the process model by providing explicit and formal description about process and process elements. Semantic annotations in form of concepts and instances pertinent to domain ontologies can be added by attaching references to them in the dedicated attribute fields of process elements. An *Annotatable* interface as shown in Figure 3 represents a concept to denote that a process element is capable of having semantic annotations. Consequently it offers methods to retrieve a set of *SemanticAnnotation* instances that are contained within it. Its main purpose is to offer to the API an interface which allows implementing classes to specify their ability to attach references of semantic annotations. It is meant to be extended by additional interfaces or implemented by process element classes and is described in Table 2.

Figure 3: Annotatable Interface

Annotatable	
Attributes	Set of SemanticAnnotation
Extending interface	N/A
Serialization	Serializable or via stream
Concrete implementations	N/A (marker interface)

Table 2: Annotatable Interface Description

Process

An LPML process is defined with a *ProcessElement* that serves as a container for other elements and defines the process structure. It represents an LPML process within the metamodel and can be programmatically manipulated through the API. The *Process* interface as shown in Figure 4 that is defined according to this notion and defines the methods for retrieving a set of constituent process components containing within the *Process* container. Since control-flow is distinguished from other process elements such as activities or gateways, there is a method to retrieve a set of these flow elements contained within the process. A process has a unique *ID* which is assigned when it is created or it optionally can be provided by the user. The process can be annotated with metadata which references semantically all process elements such as relevant activities or concrete services. The main purpose of using a *Process* is to structure its constituent process elements in an ordered way with the process node as an entry point of the process graph; moreover it allows proper serialization by providing the method to persist the entire structure of process elements contained. This interface is explained in Table 3.

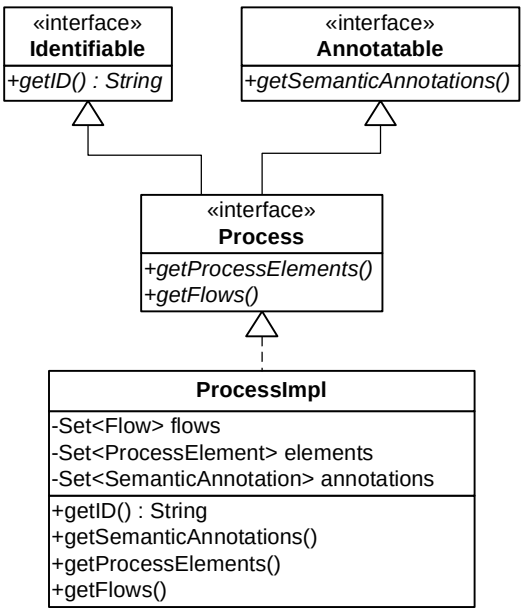


Figure 4: Process Interface and ProcessImpl

Process	
Elements contained	*Flow, ProcessElement, Semantic Annotation*
Attributes	*ID*, *SemanticAnnotation*
Extending interface	*Identifiable, Annotatable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*ProcessImpl*

Table 3: Process Interface Description

Since process definitions in a broad sense can be application-specific, the design of LPML process should incorporate both abstract process container concepts as well as bear representation of unique identifier in its canonical representation to distinguish each instance of an LPML process created. We believe this minimal definition is necessary in order to secure validity of the language element and refrain from bringing in irrelevant application-specific attributes. Composite process elements are realized in simple containment relationship with Process that helps to simplify the memory model of a process instance.

Nameable

The *Nameable* interface shown in Figure 5 represents a concept to denote human readable names for process element. It can be implemented by concrete classes representing any process elements that are displayed in the process editor so that user can give meaningful names to these elements. Consequently it offers method to retrieve and set the name for the element. The characteristics of a *Nameable* interface are shown in Table 4.

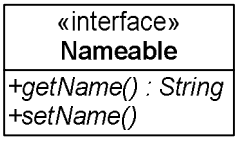


Figure 5: Nameable Interface

Nameable	
Attributes	Name
Extending interface	N/A
Serialization	Serializable or via stream
Concrete implementations	*ProcessElementImpl, BindingImpl, ActivityImpl, ParameterImpl, FlowImpl,*

Table 4: Nameable Interface Description

Positionable

The *Positionable* interface shown in Figure 6 is used to denote process elements that are mappable to certain Cartesian coordinates for graphical representation and positioning in the SOA4All Process Editor. As such it belongs to the abstract graphical layer of LPML and is mainly used to bear positional metadata persisted in the canonical LPML representation. This metadata helps to structurally display a process identically each time when it is saved and reloaded on the SOA4All Process Editor Dashboard. A *Positionable* object can be positioned in a 3-dimensional space within the process editor by keeping track of three coordinate attributes for each instance of process element that implements this interface. The coordinate system and unit of measurement is not specified here for the sake of generic implementation. The obvious choice is the Cartesian coordinate system. While the x and y-coordinates span an element in a two-dimensional space, the third one, i.e. the z-coordinate can be used to specify which process element is overlaid on top of another one. The characteristics of the *Positionable* interface are explained in Table 5.

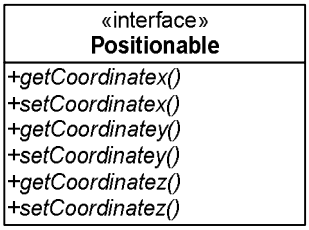


Figure 6: Positionable Interface

Positionable	
Attributes	Float coordinates
Extending interface	N/A
Serialization	Serializable or via stream
Concrete implementations	*Activity, ParallelGateway*

Table 5: Positionable Interface Description

ProcessElement

The *ProcessElement* interface shown in Figure 7 represents a generic interface that all other LPML process elements (process nodes) should inherit. It provides the basic support for identification via globally unique process element identifier, adding human readable naming for these process elements and the capability of attaching references to semantic annotations pertinent to these process elements. Furthermore *Gateway* is conceptually classified as a type of *ProcessElement* in accordance with the previous language version. In order to ensure global uniqueness, for instance, in some collaboration scenarios where process elements of different process instance might be included and used in mixed manner,

the identifiers help to ensure which specific instance of a process element is addressed.

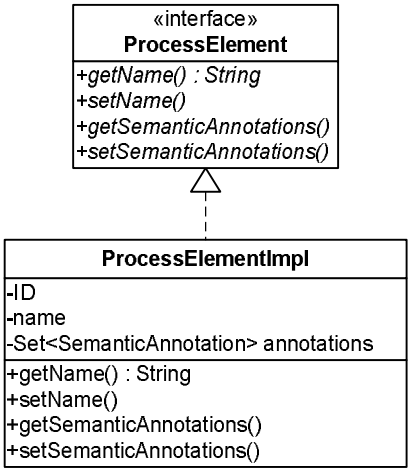


Figure 7: ProcessElement Interface and ProcessElementImpl

An instance of *ProcessElement* is implemented with the inherited identifier, which is a generated UUID. Consequently, the process element identifier can be used across different *ProcessElements* contained within different instances of processes as well to identify the process element globally. The *ProcessElementImpl* is the primary concrete class, which implements this interface and provides methods to retrieve a corresponding human readable name of this *ProcessElement* and a set of *SemanticAnnotations* present. The characteristics of *ProcessElement* are described in Table 6.

ProcessElement	
Elements contained	Set of SemanticAnnotation
Attributes	ID, Name
Extending interface	*Identifiable, Namable, Annotatable, Serializable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*ProcessElementImpl, ActivityImpl, FlowImpl, GatewayImpl, ParallelGatewayImpl*

Table 6: ProcessElement Interface Description

Activity

The *Activity* interface shown in Figure 8 is defined conceptually to represent the notion of a process activity. The *Activity* interface provides the flexibility in the final version to bind one or more compatible pairs of services and their operations. It represents a unit of work in an

LPML business process. It extends the *ProcessElement* in order to inherit the globally unique process element identifier and the capability to attach *SemanticAnnotations* for an instance of *Activity.* Furthermore, Activity extends the *Positionable* interface to inherit the coordinate information used to properly display placement of *Activity* instances in an LPML process model.

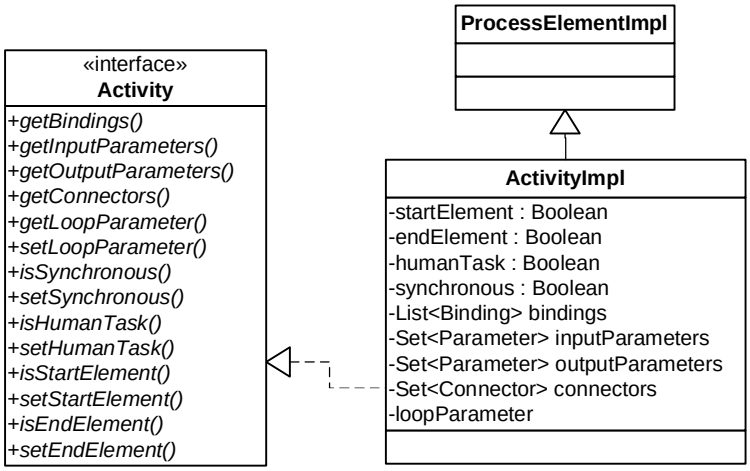


Figure 8: Activity Interface and ActivityImpl

An *Activity* has a list of *Bindings* used to reference and bind semantic services based on their semantic annotations. We devise this design with the advantage that in case of failover of services, the backend components with service adaptation logics can simply fallback to the next available service in the service list during runtime. Consequently, the order within the service list, which is defined when they are bound during design-time, reflects the invocation preference of the services during runtime. This characteristic implies the invocation semantics that the higher ordered services should be attempted first during runtime over the remaining ones, where evocable services can dynamically replace unavailable ones.

Each *Activity* defines a set of input *Parameters*, which are required to instantiate the control-flow. A process modeller is knowledgeable about the required set of *Flow* objects to realize the control-flow; which together with the defined *Activity* object instances realize the specific execution order of a process model. Regarding design-time specification of dataflow, there is a partial order relationship among all the *Parameters* of the *Activities*. During runtime the service backend components traverse the process graph to realize the execution order based on the *Flow* set before executing these *Activity* instances. The set of input *Parameters* also contain the dataflow mapping information which is kept in the attributes within the dataflow *Connectors*. There is a new supported feature for looping activities in this version of LPML by introducing a new looping *Parameter* within the *Activity* to denote that this type of *Activity* is executed iteratively during runtime until all the elements within the loop parameter are traversed. Analogous to the set of input *Parameters*, an instance of *Activity* provides methods to set and retrieve a set of output *Parameters* for the current *Activity*. By specifying the proper order and mappings of the input and output pairs of Parameters that are connected to an *Activity*, there is an additional advantage of enabling lightweight service composition. The backend Design Time Composer (DTC) component identifies the proper set of parameter pairs in combination with the service-operations pairs in order to retrieve the most appropriate set of semantic service operations for resolving execution. This task to bind appropriate services to activities can also be done manually in the SOA4All Process Editor.

Furthermore, this information can also be used to automatically render human tasks to the proper set of human task clients.

In the dataflow regard, each *Activity* instance has a set of *Connectors* that define the mappings for dataflow information. These *Connectors* specify which type of dataflow operations can be performed on the data which passes a specific *Flow*. There can be zero or more *Connectors* specified for each input or output *Parameter* of an *Activity*. The details of process level mechanism of handling concrete operations for dataflow is described in Section 2.5. *Activity* provides method retrieve a set of associated with an *Activity*.

One of the challenges is to support iterative operations within an *Activity* and it is addressed in the final version of LPML. Support of iteration is realized by providing support of *Parameters* of type Java Collection. During runtime each value of the *Parameter* of the *Collection* type is executed when a *Collection* parameter is set as the loop parameter and the iteration will be executed until the *Collection* is exhausted. Otherwise the value is treated as a common input parameter. The rationale of using this iteration semantics lies in the fact that the majority of looping is performed mainly in *Activities* of a business process. Each such *Activity* can be either viewed as iterative task or a dedicated service which executes exactly the number of iterations according to its inputs. Due to this practical design, we believe to achieve a homogeneous mechanism of iterative processing that is oriented to *Parameters* of an *Activity*. The advantage is to align iterative loop handling with other non-iterative activity execution without further introducing other more complex or unnecessary artifacts. The *Activity* interface provides methods to set and retrieve the looping *Parameter*.

There is a series of Boolean attributes that specify whether an *Activity* is synchronously executed, whether it is a human task similar to the previous version of the language. The start and end event of an LPML process can be specified by setting the Boolean flags of an *Activity* instance accordingly. The characteristics of the *Activity* interface are described in Table 7.

Activity	
Elements contained	Set of *Binding*, set of input/output *Parameter*, set of *Connector*, loop Parameter
Attributes	ID, Name, Start-element, End-element, human task, synchronous (Boolean).
Extending interface	*ProcessElement, Positionable, Serializable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*ActivityImpl*

Table 7: Activity Interface Description

Connector

The *Connector* interface shown in Figure 9 represents the concept of an associative element that defines the dataflow mappings between one or more outputs of preceding activities and a single input of a successive activity. In order to specify the data operations that are

possible on the outflow of the parameter before the flow reaches the input of the successive activity, the *Connector* interface provides methods to set and retrieve the SPARQL query-based semantic dataflow mappings associated with this *Connector* instance. This is stored internally as query string within the *Connector*. Thus data transformation can be applied conveniently to the dataflow in the flow connected by the *Connector*. While this mechanism elegantly separates the notational specification from the concrete dataflow operations that are actually performed, we believe that using indirection to retrieve dynamic SPARQL dataflow manipulation query provides greater scalability to the dataflow handling as a whole. It is necessary to specify a set of input and output *Parameters* when creating an instance of a *Connector* and it provides methods accordingly for doing so. A *Connector* instance inherits the globally unique process element identifier through its relation to *ProcessElement*. The characteristics of the *Connector* interface are shown in Table 8.

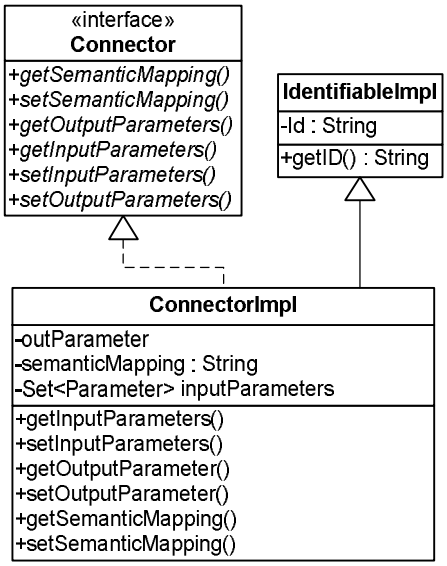


Figure 9: Connector Interface and ConnectorImpl

Connector	
Elements contained	String of dataflow operation mapping, set of input Parameter
Attributes	ID, Name, output Parameter
Extending interface	*Identifiable, Serializable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*ConnectorImpl*

Table 8: Connector Interface Description

Binding

This *Binding* interface shown in Figure 10 represents an executable entity that is used to combine executable process elements i.e. *Activity* with a group of services in the form of service-operation pairs. This rationale of this design lies in the lightweight view of services being unit of works that are directly mappable to *Activities* or human tasks without regarding other technical details of these services except their corresponding operations. A tuple of a service-operation pair consists of the service instance in the form of an URL reference and one operation that is defined in the interface of this specific service. Since multiple operations are allowed within each service, in practice there is usually one or more of such tuples that must be stored within each instance of *Binding*.

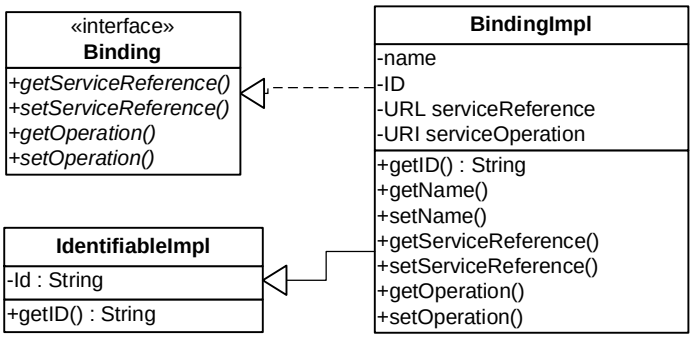


Figure 10: Binding Interface and BindingImpl

This design approach elegantly simplifies the information that is needed to be kept within each *Binding*. Each tuple represents therefore an executable operation within such execution context. In the previous version of the language, since service operations are contained within a service instance which must be bound via a conversation instance to realize a *Binding*, it has the drawback that operation lookup require navigating two levels of indirection to reach the available operations of a service; this has been simplified in the final design substantially by storing service-operation tuples directly within a *Binding* instance. Binding is based on the concept of the MSM model (Lambert, 2010) as integrated in the SOA4All Procedure-Oriented Service Model (POSM)[1], in which semantic services are identified by URLs and their operations are identified by URIs. If a certain process has been deployed to the execution engine previously, it is possible to discover its service URL by looking up the deployment URL that is referenced in a standard Linked Data-style `rdfs:seeAlso` annotation to the MSM/POSM model. A Binding instance provides methods to set and retrieve the inherent service URL as well as those to set and retrieve the associated operation URI. The characteristics of the *Binding* interface are shown in Table 9.

Binding	
Elements contained	Service reference URL, operation URI
Attributes	ID, Name, deployment URL, operation URI

[1] http://www.wsmo.org/ns/posm

Extending interface	*Identifiable, Nameable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*BindingImpl*

Table 9: Binding Interface Description

Parameter

The *Parameter* interface shown in Figure 11 represents the notion of I/O-parameters of activities and services, either for input or output parameters. In general this notion can be extended to cover also human tasks which are special types of activities requiring human interventions. It is possible to attach a set of *SemanticAnnotation* to the *Parameter* to further refine semantic descriptions about the process element. Other than the previous version of LPML, for pragmatic reason *Parameters* are now associated with an *Activity* through an instance of *Connector* rather than going through an extra indirection of *Conversation*. It is described in Table 10.

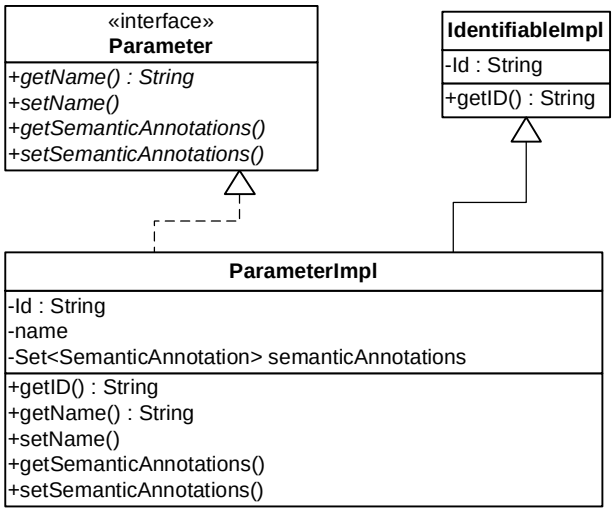


Figure 11: Parameter Interface and ParameterImpl

Parameter	
Elements contained	Set of *SemanticAnnotation*
Attributes	ID, Name
Extending interface	*Identifiable, Annotatable, Nameable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*ParameterImpl*

Table 10: Parameter Interface Description

SemanticAnnotation

Similar to the previous definition of a *SemanticAnnotation*, this interface shown in Figure 12 is used to represent arbitrary form of semantic annotations. Pertinent to the current project are the annotations defined by the WSMO-Lite, MicroWSMO ontology, expressions defined in WSML (axioms) and SPARQL (queries), and links to further RDFS vocabularies and WSML ontologies. There are two main ways to define *SemanticAnnotation*: the first is to specify a reference URI to point to a piece of *SemanticAnnotation* that can be retrieved by dereferencing the URI to retrieve the corresponding concept instances for instance from the Semantic Space; the second way is to embed and store such annotations as String in an instance of *SemanticAnnotation*. For the definition of valid types of *SemanticAnnotation*, a method is provided to set the *AnnotationType* for the instance. The characteristics of *SemanticAnnotation* are shown in Table 11.

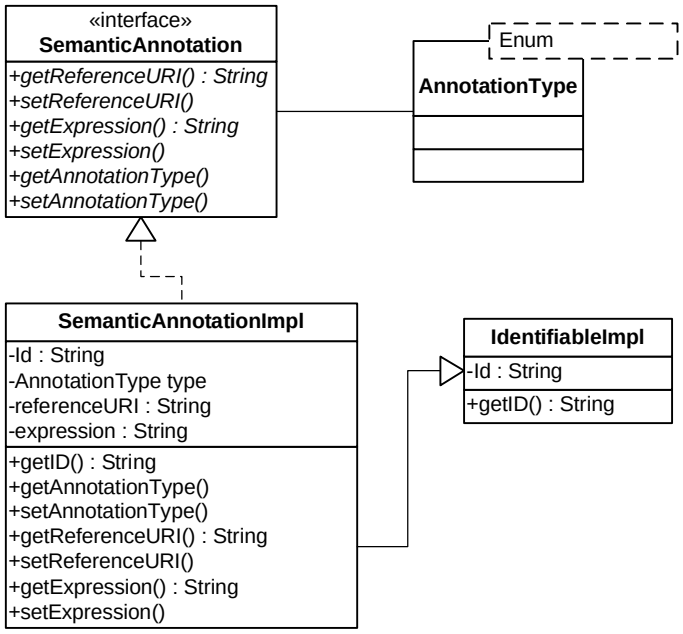


Figure 12: SemanticAnnotation Interface and SemanticAnnotationImpl

SemanticAnnotation	
Elements contained	*AnnotationType*
Attributes	ID, Name, type of *SemanticAnnotation*, reference URI to external *SemanticAnnotation*, expression of internal *SemanticAnnotation*
Extending interface	*Identifiable*

Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*SemanticAnnotationImpl*

Table 11: SemanticAnnotation Interface Description

AnnotationType

The *AnnotationType* is used to set the appropriate values for the displayable property of a *SemanticAnnotation*. Since *SemanticAnnotations* are widely used for a variety of purposes in SOA4All, for instance, to annotation services, define the concepts and properties in domain ontology that are used by these services, to denote metadata to specify functional classification of services, to denote non-functional properties of services, to specify conditions, constraints and effects axioms for states and state transitions of a stateful semantic service model such as WSMO/WSMO-Lite. Consequently it is necessary to differentiate the types of *SemanticAnnotations* and define them accordingly when an instance of *SemanticAnnotation* is created. These types and usage are described in Table 12.

Types	*Descriptions*
FUNCTIONAL_CLASSIFICATION	Specifies a functional classification according to the taxonomic scheme of MSM
NON_FUNCTIONAL_PROPERTY	Specifies any non-functional parameters such as efficiency, user preferences or performance related indicators
PRECONDITION	Specifies, as a semantic annotation (i.e., in WSML), a logical expression that must be evaluated true before an element is considered valid
EFFECT	Specifies any logical expression or semantic annotations in form of predicates that must be evaluated and after a process element is executed and result produced
META_DATA	Specifies the specific type of semantic annotations that are used to describe and are related to I/O parameters of Activity
REQUIREMENT	Specifies the specific type of semantic annotations that are used on the Process instance and Activity instance level by backend components, denoting global (process level) and local (activity level) compulsory requirements.
CONSTRAINT/PREFERENCE	Specifies the type of semantic annotations that are related to global (process level) and local (activity level)

	optional preferences used for binding filtering.
SELECTION_CRITERIA	Specifies the type of semantic annotations that are evaluated as preferences when service selection is performed
REPLACEMENT_CONDITION	Similar to the above type and is retained for backward compatibility
CONTEXTUAL_INFORMATION	Specifies contextual semantic annotations
REFERENCE	Specifies the process and activity level preferences of an LPML process

Table 12: AnnotationType Description

Flow

The *Flow* interface shown in Figure 13 represents the abstract notion of a transition from one business process element to another. It mainly differentiates between inbound and outbound flow with the corresponding source and destination of the current *Flow*. The source and destination are basically process elements from which the current *Flow* originates from or to which it directly converges. This configuration helps to define implicitly the partial ordering relationship between process elements connected by any *Flow*. In order to form a complete process, each *Flow* instance must have a condition associated with it, yielding a Boolean value by evaluation of the condition. *Flow* condition is stored internally as String which specifies an arbitrary predicate that is either evaluated to true or false at runtime. While the former value dictates that the *Flow* can be traversed, the latter outcome prohibits the corresponding *Flow* to be traversed at runtime while the process instance is executed. Consequently, it is mandatory to define a default *Flow* in the set of flows going out of a process element to accommodate the special case if all previous conditions associated with the *Flow* evaluate to false. In that case, the default *Flow* is followed.

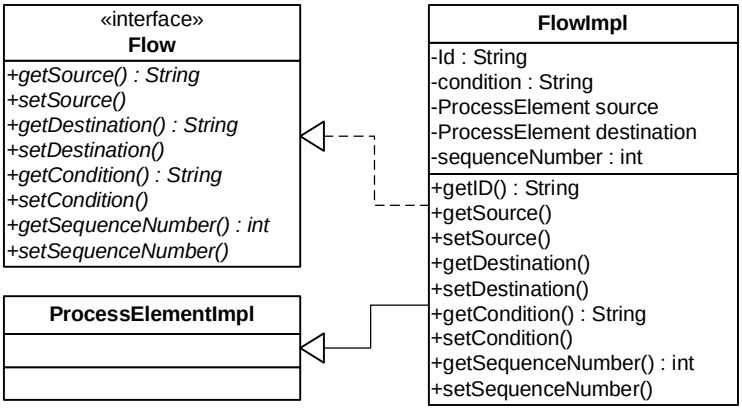


Figure 13: Flow Interface and FlowImpl

Flow conditions are abstract notions representing arbitrary Boolean expression, evaluable predicates and logical formulae. At the abstract level they should be expressed independent to a specific *Flow* implementation. One implementation approach is to delegate the

evaluation of concrete conditions at runtime to some external logical instance, e.g. we could incorporate an executable SPARQL expression which contains an SPARQL ASK query instead of simple predicates to handle conditions or we rely on XPath expressions to achieve the same effect. Using the SPARQL approach could potentially provide the advantage of using both standard machinery from the semantic technology community, a familiar language for those working with lightweight semantic languages, including the growing community using Linked Data, and support for more complex dataflow oriented condition specification. In the final design of the LPML, the notion of an inherent sequence number provides a mechanism to order different instances of *Flow* that is based on a natural order of them within the LPML process model. This mainly helps for more efficient traversal of the process graph and allows mapping different *Flow* instances in a certain control-flow context. The characteristics of the *Flow* interface are shown in Table 13.

Flow	
Elements contained	Source and destination process elements, condition predicate
Attributes	ID, Name, Source, Destination, Condition expression, sequence number
Extending interface	*Identifiable, Annotatable, Nameable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*FlowImpl*

Table 13: Flow Interface Description

Gateway

The *Gateway* interface shown on the left in Figure 14 retains its semantics as in the previous version and represents the notion of a split of *Flow* that is implicitly handled by *Flow* condition evaluation in an LPML process model. In addition a *Gateway* represents also the concept of merging after *Flow* has been split. The *Gateway* allows process modeller to split a *Flow* into two or more outbound *Flows* which must be merged before each LPML process is terminated.

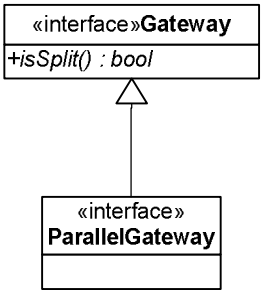


Figure 14: Gateway and ParallelGateway Interfaces

This is done by using the provided method to set the Boolean attribute to specify the type of

the *Gateway*. This version of the language supports the parallel semantics of split and merges *Gateway* to address the requirements of real-world business usecase scenarios to support parallel *Flow* and *Activity* instantiation without blocking. The concrete semantics of a *Gateway* should be specified in the classes that implement it. The characteristics of the *Gateway* interface are shown in Table 14.

Gateway	
Attributes	Split flag (Boolean)
Extending interface	*ProcessElement, Positionable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	N/A

Table 14: Gateway Interface Description

ParallelGateway

The *ParallelGateway* interface shown in Figure 15 represents the semantics of a parallel split or merges *Gateway* in the control-flow. It is mainly used to give the execution engine a hint that some set of independent process elements of the LPML process can be executed in parallel in order to improve the overall process execution efficiency. Support of parallelism can enhance efficiency of execution of independent activities in a process such as human tasks or asynchronous web services which are often inherently slow in response; consequently other activities that do not depend on response of them can be executed without waiting. We device this design with regard to a given set of N services, parallel execution can asymptotically yield polynomial performance improvement because the overall execution time is equal to the time needed to execute the slowest service or human task within the set.

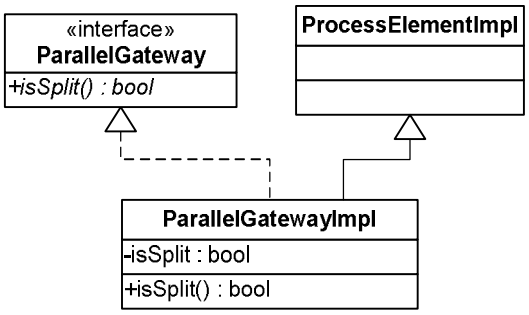


Figure 15: ParallelGateway Interface and ParallelGatewayImpl

When an instance of *ParallelGateway* is reached during process execution, the execution engine can create one parallel execution token, i.e. parallel instances of *Flow* emerging out of this gateway so that the subsequent instances of *Activity* can be instantiated and executed in parallel. For valid process model, a merging instance of *ParallelGateway* must be used to synchronize such parallel executing *Flow* back to one *Flow*. At runtime execution, the

Gateway must wait until all the preceding parallel tokens have arrived before it continues triggering subsequent execution of the following process elements. In practice merging status can be determined by traversing the process graph backwards in order to find all tokens that could potentially arrive at the corresponding merging Gateway. A *ParallelGateway* inherits the fields of the previously described *Gateway* interface and its characteristics are shown in Table 15.

ParallelGateway	
Attributes	ID, Name, Split flag (Boolean)
Extending interface	*Gateway, Serializable*
Serialization	Serializable both in XML format and via Java object serialization mechanism
Concrete implementations	*ParallelGatewayImpl*

Table 15: ParallelGateway Interface Description

2.2 LPML: Process Modelling Lifecycle

In the following sections we describe a design process for LPML modelling and execution. A design process represents a lifecycle of step-by-step breakdown of the necessary measures which the user employs, aided semi-automatically by all WP6 and appropriate GUI based tools, in order to create a valid and executable LPML process. We walk through the necessary steps required to create executable LPML process model by involving both user interaction and support of the automated tools of the service construction platform of WP6. We refer to the LPML API elements whenever necessary to give an updated view of the language usage in a tools-guided process modelling context.

2.2.1 Conceptual Design Process

Some external effort in the process modelling community such as the approach given in (Gil 2006) describes creating executable workflows out of templates comprising three steps in general. The first step defines data and execution independent workflow templates. Subsequently, workflow instances are created and specification of data input needed on activity level are mapped on the dataflow definition for these instance. In the final step executable workflow is instantiated by mapping existing resource dependent data sources and other resources to the dataflow. Our design process to create executable LPML process shows resemblance to the described approach consisting out of five steps as shown in Figure 16.

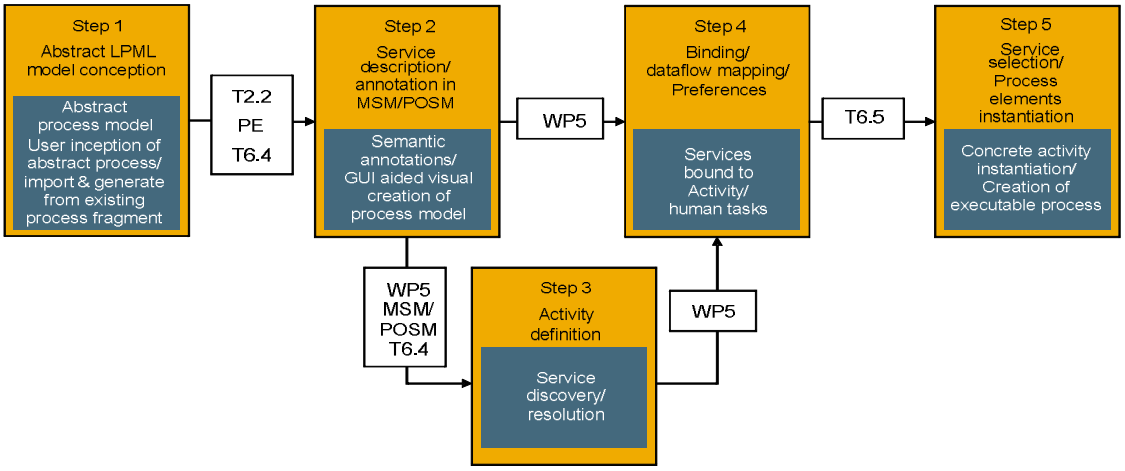


Figure 16: Design Process using LPML and Service Construction Components

The first step the user (a process modeller) conceptually conceives of the abstract process which the modeller intends to create. The modeller can either start from scratch on paper or in a mental process. In order to derive the set of requirements, especially concerning required resources and data, requirement elicitation process can be carried out in advance. The current version of language supports in addition to manual creation of a process fragment also the import of generated LPML process fragments or existing LPML process templates created previously in other modelling projects and stored in a process/template repository (Gorronogoitia 2010). Templates can be generated by using transformation from BPMN as described in Annex A. The main rationale lies in the fact that the user may want to reuse a certain piece of process workflow that exists as template expressed in another standard modelling notation. Supported by the SOA4All Process Editor, in step 2, the modeller interacts with the GUI to draw her conceptual process on the dashboard. A graphical process model is created with the underlying canonical LPML process model representation that is used by the platform services of the service construction components. By using additional tools, e.g. the external service annotation widgets (SWEET/SOWER) in the dashboard, process modellers can provide semantic annotations to the set of candidate web services which the modeller wants to use for fulfilling the tasks within the process context. The modeller either searches for and integrates existing services in hand or uses the service discovery component in the studio to find a set of previously annotated candidate services. The first and second steps represent the design-time tasks. Ideally the backend components will automatically enhance and transform the created process model in the following steps at runtime without further user interaction.

In case further user interaction and information is necessary, the process modeller can walk through step 3 and 4. In step 3 the modeller specifies the concrete *Activity* that is specifically intended for a portion in the process. In the canonical representation of the process, *Activity* instances can be created using the LPML API and further characteristics, e.g. message

processing mode can be specified. On the other hand according to the minimal service model (MSM) (Lambert 2010), existing service annotations may require harmonization or mediation to transform them into conforming format. The model can manually provide further editing on the annotations using the Studio tool and subsequently define binding for the selected services. In step 4 manual creation of *Binding* and *Connector* oriented dataflow mappings assisted by the Studio tools aims to realize the concrete dataflow operations on data between adjacent *Activity* instances as well as providing specification for the corresponding input and output *Parameters* (I/O) for these *Activity* instances.

In step 5, WP6 components are capable of finalizing the LPML process by performing service selection and binding, *Activity* instantiation, optimizations on the process model and a series of other operations. Subsequently the execution component can load the LPML process and execute it. Details of these steps are described in deliverable D6.4.3.

2.2.2 Modelling-centric Approach

The previously described lifecycle clearly indicates that using the current process language elements as described in Section 2.1.2 contributes to a comprehensive design process using our updated LPML approach. These LPML elements belong to the following groups which together constitute the essential LPML vocabulary:

- *Process*
- *ProcessElement*
- *Flow*
- *Gateway*
- *Activity*

Assisted by the Process Editor GUI tool in the SOA4All Studio and the WP6 backend components, the core LPML vocabulary and visual language elements enhance the efficiency of the modeller's task in both automated and user-interaction-guided manner.

We have tested the prototype of these tools with the update final version of the LPML. User experiences have shown that by simplifying the language to an appropriate expressiveness and retaining proper tooling support, our approach has successfully guided test modellers to perform a series of recurring modelling tasks.

2.3 Iterative Activity and Looping

The most common programming languages include iterative blocks or loops that iterate their body block while or until a particular condition is hold. The condition can be evaluated before entering the loop (while) or after each iteration (do-while). More specific sort of loops is specialized to transverse collection of objects (for each) where the loop body is executed for each item within the collection.

In the context of business process modelling, languages such as BPMN do not offer special language primitives to specify iterative blocks or loop over an arbitrary sub-process or arbitrary sub-graph of a composite task. Although they can be modelled using other modelling primitives such as exclusive gateways. However, that makes the model difficult to understand, since it adds artificial structures that deviates the model from its intending meaning. Fortunately, BPMN offers a loop activity, which, although limited to loop over only one activity when the condition holds, can be easily extended to support more complex blocks of activity by using sub-processes.

The SOA4All scenarios have identified the need to support loop in LPML, although the need is restricted only to the case for going over each object within a collection. This is the case when the output of one activity is a collection C<T> of types T and the subsequent block of activities should be traversed over each individual object of type T. Even if this scenario can be modelled with the current LPML primitives, we opt for adding special support to for-each loop in the language in order to avoid aforementioned artificial and cumbersome complex models. This is specially required if we want to keep LPML concise, easy to use and understandable.

Similar to the BPMN specification, we support loop activities, constrained as for-each loops. The LPML loop activity does not include a condition (as BPMN) that determines how far the activity must be iterated, but a reference to one of its inputs that contains the collection whose items must be traversed. In case the block to iterate is more complex than a single activity, the modeller can extract the complex block as a sub-process that is deployed and afterwards bound to a single loop activity, in a similar way BPMN encourages.

Figure 17 shows a snapshot of LPML metamodel with loop activities. Note that LPML does not add an additional loop activity but reuses the existing one that is considered as a loop activity as soon as its *loopParameter* property is not null.

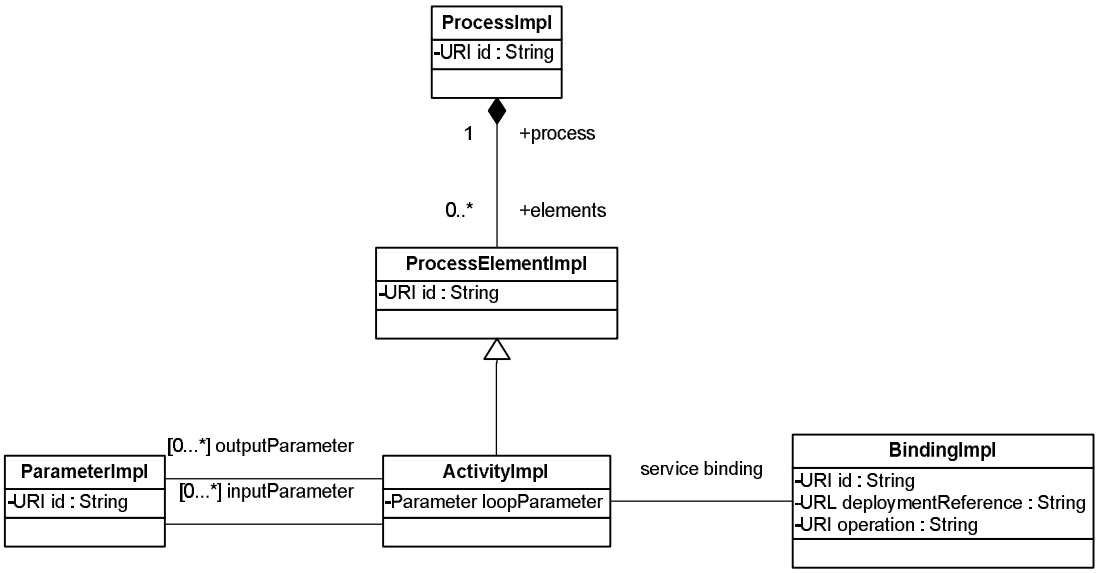


Figure 17: Loop Parameter

2.4 Dynamic Semantic Service Discovery and Binding

The implementation of the LPML[1] tooling support currently uses a SPARQL query based on the use of rdf:List to support collections, as explained in Section 2.5.

In the area of service discovery and ranking the LPML tooling is being updated to take account of the definition of unbound Activities which serve the purpose of process templates, encapsulating users' functional requirements and preferences in a standard model (previously called 'goal') as well as the provision of the DisCloud repository, which is based on the storage and brokerage of these templates against the service descriptions in the iServe service repository, as described in D5.4.2.The use of DisCloud brokerage is seen at

two points in process definition and execution. In the first place, this extends the use of discovery to bind activities at design time, by allowing process designers to take advantage of reused templates rather than define an ad hoc one for a requirement that might have been experienced before – by this designer or another.

Secondly, while in several scenarios it is unlikely that open-ended dynamic discovery and binding would be applied at run-time, an extension is planned to DisCloud to allow templates do be matched, per user, against a pre-approved 'short-list' of services, from among those discovered, which are still dynamically ranked. Since this ranking includes up-to-the-minute monitoring information from the crawler, and in future from the dedicated SOA4All analysis platform, this allows binding of the best candidate at run-time without the perceived dangers of open-ended dynamic discovery. At the same time the process designer is enabled to expand on the short list without re-defining or re-deploying the process.

2.5 Dataflow and Conditions Perspective and Design Issues

Besides purely control flow oriented constructs, the LPML aims at providing some data flow oriented constructs for supporting mashup-like service composition. As such we require a reasonably simple and familiar data processing language. In the previous deliverable D6.3.2 we presented a holistic view of data flow connectors and operators that should be supported by any mashup-based description language. Such operators were required to model data manipulation through the LPML. In particular, operators such as Merge, Split, Count, Filter, Reduction, Sort, Loop, Sub-Description and Aggregation were detailed.

Rather than implementing each and every operator, the LPML team decided to be more general, and then considering more operators by allowing the LPML language to host SPARQL queries on data. In this direction, every previous single operator is mapped to a specific pre-defined SPARQL query. The mapping depends on the services involved in the composition and the data that their required operations produce and consume, therefore needs to be manipulated.

We depend heavily on the definition of message partonomy, introduced in iServe's extension of the MSM[2] and adapted, based on standard vocabularies and predicates, for POSM. The basis of this extension is the original MSM concept `Message`, which was modeled as follows:

```
msm:hasInputMessage  rdf:type  rdf:Property ;
    rdfs:domain  msm:Operation ;
    rdfs:range  msm:Message .
msm:hasOutputMessage  rdf:type  rdf:Property ;
    rdfs:domain  msm:Operation ;
    rdfs:range  msm:Message .
msm:hasInputFault  rdf:type  rdf:Property ;
    rdfs:domain  msm:Operation ;
    rdfs:range  msm:Message .
msm:hasOutputFault  rdf:type  rdf:Property ;
    rdfs:domain  msm:Operation ;
```

[2] http://iserve.kmi.open.ac.uk/wiki/index.php/IServe_vocabulary

```
    rdfs:range  msm:Message .
msm:Message  rdf:type  rdfs:Class .
```

This has so-far been extended as follows[3]:

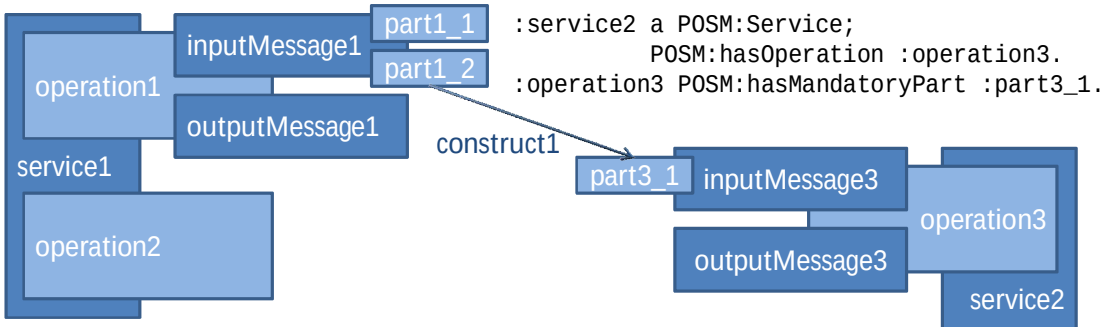
```
:Message rdfs:subClassOf :MessagePart .

:hasPart rdfs:subPropertyOf <http://www.w3.org/2001/sw/BestPractices/OEP/
                             SimplePartWhole/part.owl#hasPart_directly> ;
         rdfs:domain :MessagePart ;
         rdfs:range :MessagePart .

:hasOptionalPart rdfs:subPropertyOf :hasPart .
:hasMandatoryPart rdfs:subPropertyOf :hasPart .

:hasName rdfs:subPropertyOf rdfs:label ;
         rdfs:domain :MessagePart .
```

This is particularly useful in the case, for instance, that a number of inputs are provided to a service. This was explicitly (but non-semantically) modelled in WSDL 1.1, where messages could possess parts then separately typed in XML Schema. This is not present in: WSDL 2.0, where SOAP bodies are typed as a whole in XML Schema; in XML over REST, where the same may be true but without a direct unifying model; or in JSON over REST, where JSON Schema is likely not to be used at all and does not distinguish separate inputs anyway (JSON is simply a tagged recursive array-based data structure).

In all of these cases we can use the partonomy model to both provide further documentation of the expected interpretation of the message and, via appropriate lifting, to enable the automatic generation of the appropriate SPARQL query to effect dataflow from abstract mappings between message parts (for which a graphical user interface has been provided within the SOA4All Studio).

Let us first consider a simple example. Consider the message partonomies shown on the left and right on the following figure, and the SPARQL query shown underneath.

[3] Note that `Message` may also be renamed `MessageContent` in order to disambiguate.




```
:service2 a POSM:Service;
          POSM:hasOperation :operation3.
:operation3 POSM:hasMandatoryPart :part3_1.
```

```
:service1 a POSM:Service;
          POSM:hasOperation :operation1, :operation2.
:operation1 POSM:hasOptionalPart :part1_1;
            POSM:hasMandatoryPart :part1_2.
```

```
:construct1 rdf:value
"CONSTRUCT {o2:part3_1 rdf:value ?part1}
 WHERE {o1:part1_2 rdf:value ?part1}"
   ^^sparql:construct
```

Figure 18: Simple Dataflow Example

Note that since the partonomy is defined at the instance level in service definitions, and at run-time values are simply attached to this implicit structure using the `rdf:value` predicate

(see Lambert, 2010), multiple such mappings can be used to create a complete input message, perhaps derived from the output messages of different services. This scheme requires that each part in the partonomy is given a URI, i.e., blank nodes cannot be used as there is no way to separately attach a value when the part is unidentified. At the implementation level it is also necessary to keep the set of values for a given invocation separate from all other invocations in the process, as there may be other invocations of the same services. In BPEL this is natural as input and output messages are formed in separate mutable variables.

In the case of loops that iterate over collections, discussed above, a generic dataflow is most of the time sufficient to extract the items, provided that this is lifted to form an instance of `rdf:List`. In the same way a generic dataflow can be used to re-assign the tail of the list for the next iteration. These two queries are as follows[4]:

```
:constructHead rdf:value
```

[4] We assume that a usable sparql prefix can be defined as (this is unofficial, but matches the anchors in the standard):

`http://www.w3.org/TR/rdf-sparql-query/#`

```
"PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 CONSTRUCT {?root rdf:value ?value .
            ?value ?pred ?obj}
 WHERE {?root rdf:value ?list .
        ?list rdf:first ?head .
        ?head rdf:value ?value .
        OPTIONAL {?value ?pred ?obj}}"
   ^^sparql:construct

:constructTail rdf:value
" PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 CONSTRUCT {?root rdf:value ?new .
            ?node rdf:first ?each .
            ?node rdf:rest ?rest .
            ?each rdf:value ?value .
            ?value ?pred ?obj}
 WHERE {?root rdf:value ?list .
        ?list rdf:rest ?new .
        ?node rdf:first ?each .
        ?node rdf:rest ?rest .
        ?each rdf:value ?value .
        OPTIONAL {?value ?pred ?obj} .
        FILTER (!sameTerm(?node, ?list))}"
   ^^sparql:construct
```

To illustrate the use the instances (note that the round bracket notation is the list constructor in n3/Turtle):

```
@prefix :<http://www.example.com/list#> .
@prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
:root rdf:value ([rdf:value 1] [rdf:value [:p1 :i1; :p2 :i2]]).
```

The initial instances so defined, and the effect of the two queries can be seen in the following figure. Note that even if `i1` and `i2` are defined in the source graph they will be available only ‘by reference’ to activities within the loop: this can be addressed with a custom query if necessary.

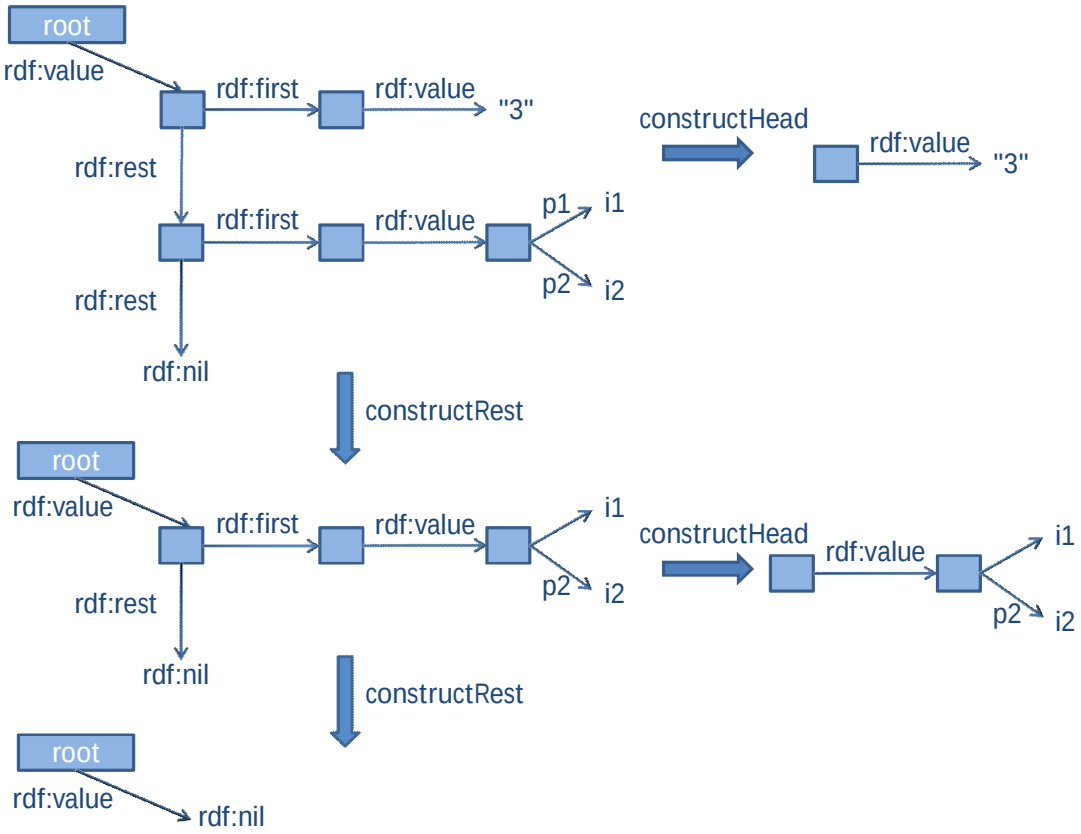


Figure 19: Dataflow for Lists

The example of lists also provides an example for the definition of conditions within processes using SPARQL. In order to define the loop condition we simply need the following ASK query:

```
:askList rdf:value
"PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 ASK {?root rdf:value ?value .
      FILTER (!sameTerm(?value, rdf:nil))}"
   ^^sparql:ask
```

In general conditions within processes may be combined with dataflow to derive an ASK query on values from some part of an output message since the condition handling of the LPML is formalism agnostic. For example if we consider `part1_2` from the simple dataflow example was given integer values, which would be documented by the triple:

```
:part1_2 sawsdl:modelReference xsd:int.
```

Now, given that some invocation of service1 has given values to its output message, we can use these values to evaluate the following query as a condition (deciding on which branch of the process to execute next, for example):

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX :<http://www.example.com/list#>
```

```
ASK {:part1_3 rdf:value ?value .
     FILTER (?value > 3)}
```

The primary outstanding issues with the partonomy and SPARQL-based approach to dataflow and conditions specified here concern the existing SOA4All approach to lifting and partonomy. In particular, it is not clear how HTTP status codes and header fields in RESTful are to be included in lifting.

This is important since:

- A condition might rely, for instance, on whether a preceding activity, which involved PUTting a resource, was successful or not. This might only be represented (i.e. the HTTP body might be empty in both cases) by having the HTTP status code of the response equal to 200 or, if unsuccessful, to a code in the 4xx range (e.g., 409 Conflict).
- A dataflow, for instance if the source activity involved POSTing a resource, might need to communicate a value from a header field (e.g. `Location` – the URI given to the resource on the server) from the response to some new message.
- It is also possible that a dataflow might be needed to pass some value into a request header field. In particular credentials may need to be passed in a process that deals explicitly with authorization.

Annex B contains our proposed extension of the partonomy model, which would allow the SPARQL-based approach detailed above to be used in all of these cases, given appropriate lifting.

2.6 Generation of LPML from BPMN

We have designed and implemented a prototype for generating LPML files from BPMN models. This work has been realized in the context of the Eclipse modelling environment, leveraging the open-source BPMN editor[5] that is very popular with Eclipse-based designers and architects. This choice is justified by the fact that the Eclipse environment is very popular for BPMN creation and we already have good technical expertise with the BPMN editor. Lastly, this choice reduces development efforts as it reuses existing code.

The technical foundation for this transformation is the same as for the LPML to Business Process Execution Language (BPEL) one[6]: the metamodel based transformation support offered by the Mangrove[7] project (formerly STP-IM). We describe the user interaction support for this transformation in Annex A. From an architectural point of view, the transformation chain comprises the following elements:

- Eclipse plug-in for adding contextual menu support for BPMN to LPML (this plug-in drives the entire transformation chain)
- Eclipse plug-ins for generating the Mangrove Core from BPMN (updated existing plug-ins in order to adapt to specific requirements of the LPML generation)

[5] http://www.eclipse.org/bpmn/

[6] please refer to D6.5.2 and D6.5.3 for more information on this transformation

[7] http://www.eclipse.org/mangrove/

- Generation of LPML from the Mangrove core, using the Eclipse Modelling Framework (EMF)-based metamodel matching with Java code (invoked from the main plug-in after the Mangrove generation)
- Eclipse plug-in for textual display of transformation status and the updating of the Eclipse workspace to show the generated LPML file[s]

We have taken the choice to generate one LPML process (and therefore one LPML file) per BPMN pool. This choice is a simplifying choice but we believe it is suitable for the purposes of this proof-of-concept prototype. In addition, we generate a rather basic LPML that is structurally identical to the BPMN pool it corresponds to, but which does not contain any executable content (such as service names or semantic information). Such data is not present in basic BPMN but we can envisage a more complex transformation, which could take into account BPMN annotations that are specifically designed for SOA4All (this would entail adding editor extensions for these annotations to the BPMN editor).

We illustrate the transformation with an example in Annex A.

2.7 Serialization and Storage

The previous version of the language supports LPML process serialization into an XML format by deep traversal of the graph of the LPML process model thereby outputting the entire set of process elements contained within the process model. Serialization is supported using the XStream library[8] which offers API methods to export the process and perform the necessary traversal. It also supports the reverse import of a serialized LPML process back to the corresponding in-memory process model on the fly. One of the advantages of this mechanism lies in its programmatic simplicity as well as an implementation-agnostic design of process serialization in the LPML API.

We have retained this serialization mechanism for the sake of compatibility and introduced another effective serialization in this final version. By using Java Object Serialization mechanism (JOS) and updating the LPML API accordingly, we have built object serialization directly into the LPML API. It is now possible to export and store LPML process graph as object files which can be either temporarily stored or persisted to the project's storage space. Comparing to the XStream serialization which exports the process models in XML based structured format, exporting them using ObjectOutputStream has the obvious advantage that process object graphs can be controlled to include or exclude certain fields during serialization such that transiently marked fields or any volatile intermittent values that are unsuitable for direct serialization can be selectively excluded. During deserialization, meaningful initial values can be reset for such fields when the process object graph is recreated in memory; while on the other hand, XStream based serialization only supports the serialization of a process structure 'as is'. Utility classes are created to support both type of serializations seamlessly. Alternatively, process serialization can be realized using Java Architecture for XML Building (JAXB)[9] serialization and XML-Schema to define the formal object-serialization structure of the process elements. This could be a replacement of the XStream library to achieve a certain degree of interoperability since JAXB allows one to define a finer grain customized object-XML binding mapping while XStream serialization does not offer this flexibility. However, the downside of JAXB is its inflexibility toward changing class structure. Modifications of class definitions or changes in the class hierarchy

[8] http://xstream.codehaus.org/

[9] https://jaxb.dev.java.net/

require simultaneous and often complex modifications of the binding mapping to properly reflect these changes. Due to project time constraint, it is decided to adapt JOS as the alternative over JAXB.

The project-wide Semantic Space of WP1 and process storage space can support storing and retrieving of LPML process model in XML serialization by using the storage services. Since the additional serialization JOS files are exported in binary format, if viewed as simple flat file, they can also be stored as simple files using the storage services by attaching these flat files as payload to the storage request. We have provided additional LPML validation utility in this LPML API version for runtime validation of the syntactic correctness of an exported or imported process model.

3. Technical Evaluation of Language

In this section we evaluate the final version of the proposed process language by appealing to the concepts of expressiveness and computational completeness of LPML and usability coverage. For the latter purpose we have conducted a small scale intra company and representative user survey to evaluate users' acceptance and their opinion about the available tools which uses LPML.

3.1 Expressiveness Issues

To lay the ground of analysis, we have selected a set of concepts and process constructs using the following well-known reference frameworks: the Bunge-Wand-Weber (BWW) framework (Bunge 1977) (Wand 1989), a adopted selection of the well-known workflow patterns in workflow systems as described in (W. t. van der Aalst 2003) and (W. t. van der Aalst 2005) as well as a set of relevant communication patterns pertinently adopted in Enterprise Application Integration (EAI) systems as described in the treaty (Ruh 2000). Our attempt to evaluate the computational completeness and expressiveness starts with mapping these well-known concepts and process constructs on a practical ground onto our LPML process elements and constructs described in the previous sections. Subsequently these mapped set of elements are subjected to case study in which users (modellers) have been given scenario specific modelling tasks and asked to identify whether the selected elements of the LPML in the final version cover and satisfy the modeller's requirement and usage expectations. This result and opinion gives a pretty good analysis of what artifacts contribute to essential set of elements to retain. Furthermore they drive the consideration of design of the corresponding LPML element in the API. This approach of user feedback and scenario-driven design helps to strike the correct balance of language expressiveness and suitability as well as flexibility of language elements since only the essentially necessary and semantically correct constructs and concepts will be mapped to our LPML design.

Green et al. have used similar approach to map the BWW representation model to constructs of BPEL as described in (Green 2007). Furthermore another attempt in (Recker 2008) has been made to apply such mapping approach to BPMN. In the same light our attempt to map constructs onto LPML has shown that a significant number of concepts from the BWW model do not have direct mappable constructs in LPML. Examples of these are the concepts for *things*, *system environment*, *event spaces*, etc. *Transformation* on the other hand is a core BWW concept which can be mapped directly to the groups of constructs constituting the LPML control-flow and dataflow elements. The lack of many direct mappable constructs does not necessarily denote deficiency of the reference model; in contrary the careful retention of the mappable constructs helps to simplify the LPML core that leads to noticeable reduction in language complexity for the sake of easier model creation and flexibility in expressions. The outcome of mappable process elements led to the reduced set of LPML vocabulary described previously in section 2.1.2.

3.2 Usability Issues

We have attempted evaluation of the simplicity and usability of the LPML and the studio tools by conducting intra company small scale user tests. Modellers are selected from a set of employees of different skill levels and vocational background. While over 80% of the candidate modellers have shown an interest in mash-up based service composition, it is identified that the SOA4All Studio tools, especially the Process Editor has met wide satisfaction due to its ease of use. However, there have been some fears of certain candidates about the possibility of making errors in attempting her first modelling tasks.

Though the portion of these candidates is small, it does show that the invariant aspect of inability of any tool to eliminate that fear from absolute laymen is comprehensible since every beginner must empirically 'learn' to use the tool and the GUI correctly in order to create a valid process model. There is no 'shortcut' in this path and once the learning process has completed, candidates have shown significant mastery of the tools especially the Process Editor, mainly due to its comprehensive interface design, its absolute alignment with the LPML canonical representation and its functional simplicity of technical offers.

Furthermore it is discovered that even inherent process composition errors of some more experienced candidates are uncovered because the Process Editor supports error handling indicating occasional minor error in their process model. This proves to be great help to the modeller through the built-in model validity checking function of the Editor.

In the past process modellers were clogged down by more unintuitive and complex process modelling tools at their disposal with complaints about the complexities and challenges cast on them when they perform their tasks. A clear need for simpler and properly guided service composition in easy-to-use style resembling the mash-up approach has gradually become evident, we believe the LPML approach: the LPML API and its corresponding components as well as user level tools address these issues and satisfy the requirements. Due to the evolutionary nature of the project, the final usability evaluation on the user tools in a larger scale is eminently necessary. This evaluation should both involve a broader and more comprehensive technical evaluation in the LPML as well as user survey on the basis of usability, user satisfaction and degree of task achievement on the then available tools. This evaluation will be due by the end of the project to deliver an evaluative holistic technical view of the LPML.

4. Conclusions

This document has described the final design and implementation of the LPML for the Service Construction Platform of SOA4All. It has addressed challenges that are related to scalability, integration with corresponding service construction platform components and improvements in user interaction and satisfaction. By integrating mechanism to generate LPML process fragments from other modelling languages such as BPMN, we have achieved better integration and reuse of certain existing modelling artifacts with our service modelling platform. We have introduced the loop handling to address iterative processing requirement in the language. By refining the dataflow mapping in the language, we have improved data oriented process modelling in the language with the benefits of integrating this task seamlessly in process editor for easy user interaction.

We believe that in its current version, LPML can be used in the SOA4All Service Construction Platform that provides the necessary functionalities to demonstrate how it is used for the benefit of a wide spectrum of users, from laymen to professionals. It provides the essential leverages to help to boost the service platform and the project's reach as a whole to a wider process user community and general public.

5. References

[Bunge, 1977] Bunge M. (1977) Treatise on Basic Philosophy, Ontology I: the Furniture of the World

[Gil, 2006] Gil, Y., Gannon, D., Deelman, E., Shields, M., Taylor, I. (2006) Workflow composition in Workflow for e-Science, Springer

[Gorronogoitia, 2010] Gorronogoitia, Y., Radzimski, M., Lecue, F., Villa, M., Matteo, G., et al. (2010). D6.4.2 Advanced Prototype for Service Composition and Adaptation Environment.

[Green, 2007] Green, P., Rosemann, M., Indulska, M., Manning, C. (2007). Candidate Interoperability Standards: An Ontological Overlap Analysis. Data Knowledge Engineering Journal, 62, 274-291

[Lambert, 2010] Lambert, D., Pedrinaci, C. (2010). CMS WG Deliverable WSMO-Lite Extras: The Minimal Service Model and Service Templates. Working draft.

[Recker, 2008] Recker, J., Indulska, M., Rosemann, M., Green, P. (2008). An Exploratory Study of Process Modelling Practice with BPMN, BPM Center Report

[Ruh, 2000] Ruh, W., Brown, W., Maginnis, F. X. (2000). Enterprise Application Integration, John Wiley & Sons, Inc.

[Schnabel, 2008] Schnabel, F., Gorronogoitia, Y., Lecue, F., Ripa, G., Radzimski, M., et al. (2008). D6.3.2 Advanced Specification of Lightweight Context-aware Process Modelling Language.

[SOA4All D2.6.2] SOA4All Deliverable 2.6.2 SOA4All Process Editor First Prototype

[SOA4All D2.6.3] SOA4All Deliverable 2.6.3 Advanced Prototype of SOA4All Process Editor

[SOA4All D6.4.2] SOA4All Deliverable 6.4.2 Advanced Prototype for Service Composition and Adaptation Environment

[SOA4All D6.4.3] SOA4All Deliverable 6.4.3 Final Prototype for Service Composition and Adaptation Environment

[SOA4All D6.5.2] SOA4All Deliverable 6.5.2 Advanced Prototype for Adaptive Service Composition Execution

[SOA4All D6.5.3] SOA4All Deliverable 6.5.3 Final Prototype for Adaptive Service Composition Execution

[W. t. van der Aalst 2003] van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A. (2003) Workflow Patterns in Distributed and Parallel Databases 14, 5-51

[W. t. van der Aalst 2005] van der Aalst, W., ter Hofstede, (2005). A. YAWL: Yet Another Workflow Language

[Wand, et al. 1989] Wand, Y., Weber, R. (1989) An Ontological Evaluation of Systems Analysis and Design Methods in Information System Concepts 79-107

Annex A. BPMN to LPML Transformation Walk-Through

This section illustrates the automatic transformation from a BPMN process designed in Eclipse to one or more LPML files corresponding to it. The transformation mechanism is outlined in Section 2.6.

We illustrate the transformation with a simple BPMN process that is depicted in Figure 20.

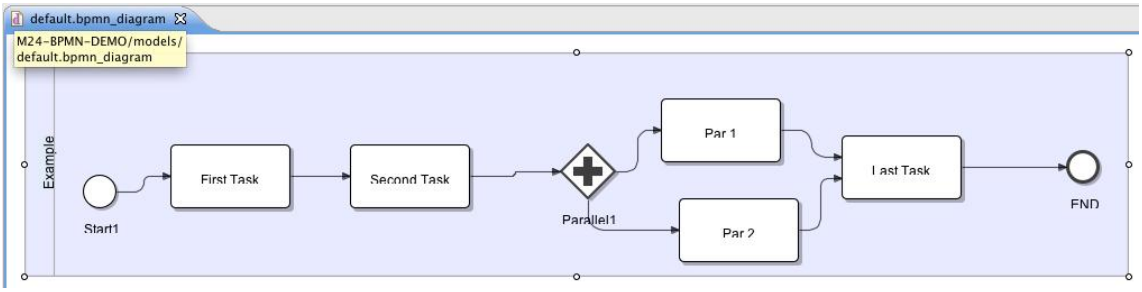


Figure 20: BPMN Sample Process

The first step that a user must take for generating the corresponding LPML is to select this file in the Eclipse workspace and “right-click” on it in order to display the contextual menu, where the LPML generation can be found, as shown in Figure 21.

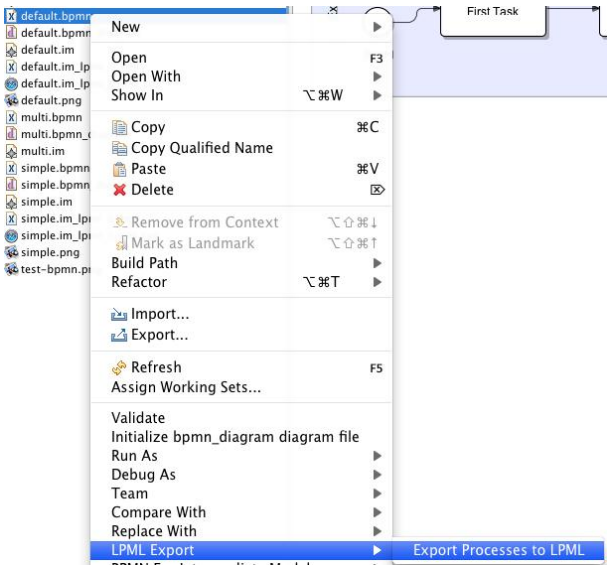


Figure 21: Contextual Transformation Menu

Selecting this option will trigger the chain of actions involved in the transformation and the user will be presented with feedback on the transformation status with the help of pop-up dialog boxes. This information is depicted in the order in Figure 22 below.

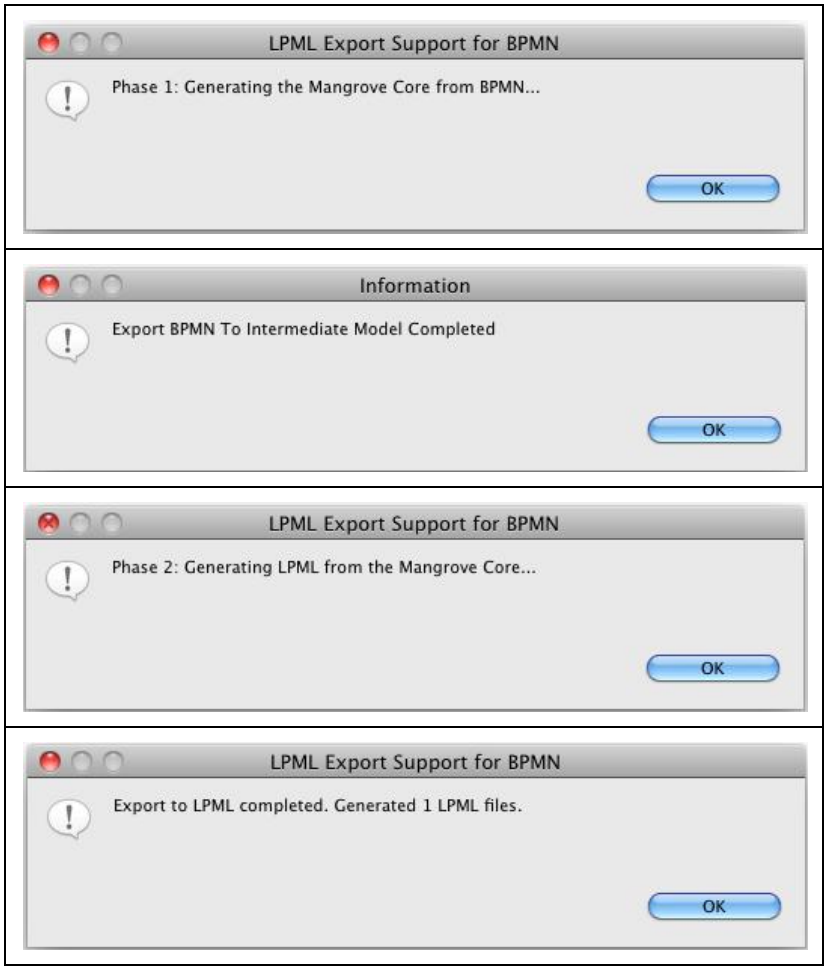


Figure 22: BPMN to LPML Progress Display

After the entire execution sequence is completed, the generated LPML file[s] will be added to the workspace and made available on the file system. In this example scenario, only one file is selected as the BPMN process consists of a single pool. For multiple pools, the transformation will generate one LPML file per pool, during a single transformation process.

The LPML file generated for this example is presented in Figure 23, as displayed by the LPML viewer.

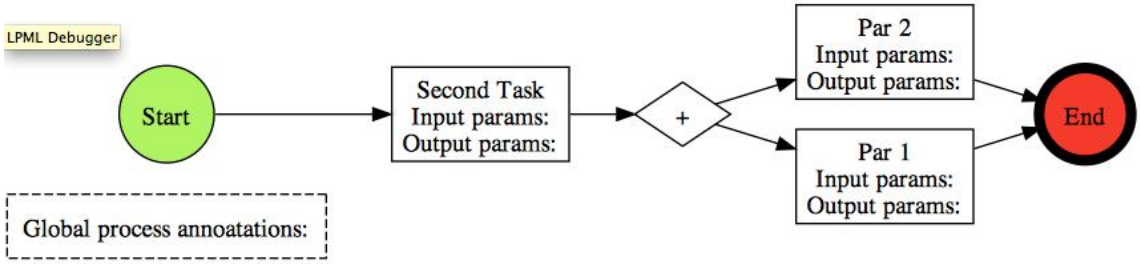


Figure 23: Generated LPML

Annex B. Proposed Extensions to Partonomy Model

As motivated above for dataflow and conditions, it is necessary to know how HTTP status codes and header fields will be included in the partonomised message model. We recommend the following extensions, which would be compatible with our SPARQL-based approach:

First the most consistent means to extend, beyond the SOAP/HTTP message body, the message lifted would be to explicitly model, with message parts, all parameters in which information can be communicated in RESTful service interaction:

```
:BodyParameter rdfs:subClassOf :MessagePart .

:SOAPMessage rdfs:subClassOf :BodyParameter, :Message .


:URIPathParameter rdfs:subClassOf :MessagePart .

:substitutes rdfs:domain :URIPathParameter ;
             rdfs:range rdf:Literal .


:URIQueryParameter rdfs:subClassOf :KeyValueParameter .

:KeyValueParameter rdfs:subClassOf :MessagePart .

:hasKey rdfs:domain :KeyValueParameter ;
        rdfs:range rdf:Literal .

:hasDefaultValue rdfs:domain :KeyValueParameter ;
                 rdfs:range rdf:Literal .


:URIMatrixParameter rdfs:subClassOf :KeyValueParameter .

:HeaderParameter rdfs:subClassOf :KeyValueParameter .
```

Second, we could make more use of the standard W3C HTTP in RDF vocabulary by extending Message with the following:

```
http:Response rdfs:subClassOf :Message .
  # therefore subject of http:responseCode

:hasOutput rdfs:domain :Operation ;
          rdfs:range http:Response .

:hasOutputFault rdfs:domain :Operation ;
          rdfs:range http:Response .
```