LarKC

*The Large Knowledge Collider:*
*a platform for large scale integrated reasoning and Web-search*

FP7 – 215535

# D1.3.1 Initial Plug-in Annotation Language

**Coordinator: Barry Bishop, Dumitru Roman**
**With contributions from: Jacek Kopecky, Georgina Gallizo, Blaž Fortuna**
**Quality Assessor: Kono Kim**
**Quality Controller: Gualy Unel**

| | |
|---|---|
| Document Identifier: | LarKC/2008/D1.3.1 /v1.2 |
| Class Deliverable: | LarKC EU-IST-2008-215535 |
| Version: | 1.2 |
| Date: | 23.3.2009 |
| State: | Final version |
| Distribution: | Public |

# EXECUTIVE SUMMARY

Plug-ins play a key role in the context of LarKC. They are the core elements that are composed in a concrete LarKC pipeline. Since multiple providers are expected to contribute with various plug-ins to the LarKC community and a large number of available plug-ins are expected, there is a clear need for a mechanism to handle plug-ins in a flexible way and to enable discovery and composition of such plug-ins. A key requirement to enable such tasks is to have explicit specifications of the functional and non-functional properties of plug-ins. In this context, this deliverable is aimed at providing an initial mechanism for specifying such properties. We propose WSMO-Lite as a starting point for the specification of the functionality of LarKC plug-ins, and describe a list of non-functional properties that characterize the quality of service of plug-ins. Finally, we show how plug-in descriptions are used in a typical concrete LarKC pipeline.

# DOCUMENT INFORMATION

| | | | |
|---|---|---|---|
| **IST Project Number** | FP7 - 215535 | **Acronym** | LarKC |
| **Full Title** | The Large Knowledge Collider: a platform for large scale integrated reasoning and Web-search | | |
| **Project URL** | http://www.larkc.eu/ | | |
| **Document URL** | | | |
| **EU Project Officer** | Stefano Bertolo | | |

| | | | | |
|---|---|---|---|---|
| **Deliverable** | **Number** | D1.3.1 | **Title** | Initial Plug-in Annotation Language |
| **Work Package** | **Number** | 1 | **Title** | Conceptual Framework & Evaluation |

| | | | | |
|---|---|---|---|---|
| **Date of Delivery** | **Contractual** | M 12 | **Actual** | |
| **Status** | version 1.0 | | final □ | |
| **Nature** | prototype □ report X dissemination □ | | | |
| **Dissemination level** | public X consortium □ | | | |

| | | | | |
|---|---|---|---|---|
| **Authors (Partner)** | Barry Bishop (UIBK), Blaž Fortuna (Cyc), Georgina Gallizo (HLRS), Jacek Kopecky (UIBK), Dumitru Roman (UIBK) | | | |
| **Responsible Author** | **Name** | Dumitru Roman | **E-mail** | dumitru.roman@sti2.at |
| | **Partner** | UIBK | **Phone** | +43 512 507 6479 |

| | |
|---|---|
| **Abstract (for dissemination)** | Plug-ins play a key role in the context of LarKC. They are the core elements that are composed in a concrete LarKC pipeline. Since multiple providers are expected to contribute with various plug-ins to the LarKC community and a large number of available plug-ins are expected, there is a clear need for a mechanism to handle plug-ins in a flexible way and to enable discovery and composition of such plug-ins. A key requirement to enable such tasks is to have explicit specifications of the functional and non-functional properties of plug-ins. In this context, this deliverable is aimed at providing an initial mechanism for specifying such properties. We propose WSMO-Lite as a starting point for the specification of the functionality of LarKC plug-ins, and describe a list of non-functional properties that characterize the quality of service of plug-ins. Finally, we show how plug-in descriptions are used in a typical concrete LarKC pipeline. |
| **Keywords** | |

| **Version Log** | | | |
|---|---|---|---|
| **Issue Date** | **Rev. No.** | **Author** | **Change** |
| 03-03-2009 | 1 | Dumitru Roman | Added abstract, introduction section, WSMO-Lite section, example section, some content to the nfps section. |
| 06-03-2009 | 2 | Dumitru Roman | Included list of nfps from HLRS and language fixes from Barry. |
| 10-03-2009 | 3 | Dumitru Roman | Included input from Cyc. |
| 17-03-2009 | 4 | Dumitru Roman | Addressed and incorporated comments from Kono Kim |

# PROJECT CONSORTIUM INFORMATION

| Participant's name | Partner | Contact |
|---|---|---|
| Semantic Technology Institute Innsbruck, Universitaet Innsbruck | | Prof. Dr. Dieter Fensel,<br>Semantic Technology Institute (STI),<br>Universitaet Innsbruck,<br>Innsbruck, Austria,<br>E-mail: dieter.fensel@sti-innsbruck.at |
| AstraZeneca AB | | Bosse Andersson<br>AstraZeneca<br>Lund, Sweden<br>Email:<br>bo.h.andersson@astrazeneca.com |
| CEFRIEL - SOCIETA CONSORTILE A RESPONSABILITA LIMITATA | | Emanuele Della Valle,<br>CEFRIEL - SOCIETA CONSORTILE A RESPONSABILITA LIMITATA,<br>Milano, Italy,<br>Email: emanuele.dellavalle@cefriel.it |
| CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O. | | Michael Witbrock,<br>CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O.,<br>Ljubljana, Slovenia,<br>Email: witbrock@cyc.com |
| Höchstleistungsrechenzentrum, Universitaet Stuttgart | | Georgina Gallizo,<br>Höchstleistungsrechenzentrum,<br>Universitaet Stuttgart,<br>Stuttgart, Germany,<br>Email: gallizo@hlrs.de |
| MAX-PLANCK GESELLSCHAFT ZUR FOERDERUNG DER WISSENSCHAFTEN E.V. | | Dr. Lael Schooler<br>Max-Planck-Institut für Bildungsforschung<br>Berlin, Germany<br>Email: schooler@mpib-berlin.mpg.de |
| Ontotext AD | | Atanas Kiryakov,<br>Ontotext Lab,<br>Sofia, Bulgaria<br>Email: naso@ontotext.com |
| SALTLUX INC. | | Kono Kim,<br>SALTLUX INC,<br>Seoul, Korea,<br>Email: kono@saltlux.com |
| SIEMENS AKTIENGESELLSCHAFT | | Dr. Volker Tresp,<br>SIEMENS AKTIENGESELLSCHAFT,<br>Muenchen, Germany,<br>E-mail: volker.tresp@siemens.com |
| THE UNIVERSITY OF SHEFFIELD | | Prof. Dr. Hamish Cunningham,<br>THE UNIVERSITY OF SHEFFIELD<br>Sheffield, UK,<br>Email: h.cunningham@dcs.shef.ac.uk |

| | | |
|---|---|---|
| VRIJE UNIVERSITEIT AMSTERDAM | | Prof. Dr. Frank van Harmelen,<br>VRIJE UNIVERSITEIT AMSTERDAM,<br>Amsterdam, Netherlands,<br>Email: Frank.van.Harmelen@cs.vu.nl |
| THE INTERNATIONAL WIC INSTITUTE, BEIJING UNIVERSITY OF TECHNOLOGY | | Prof. Dr. Ning Zhong,<br>THE INTERNATIONAL WIC INSTITUTE,<br>Mabeshi, Japan,<br>Email: zhong@maebashi-it.ac.jp |
| INTERNATIONAL AGENCY FOR RESEARCH ON CANCER | International Agency for Research on Ca<br>Centre International de Recherche sur le Ca | Dr. Paul Brennan,<br>INTERNATIONAL AGENCY FOR RESEARCH ON CANCER,<br>Lyon, France,<br>Email: brennan@iarc.fr |
| INFORMATION RETRIEVAL FACILITY | | Dr. John Tait<br>INFORMATION RETRIEVAL FACILITY<br>Vienna, Austria<br>Email : john.tait@ir-facility.org |

# TABLE OF CONTENTS

## List of Figures

## List of Acronyms

| Acronym | Description |
|---|---|
| BREIN | Business objective driven REliable and Intelligent grids for real busiNess |
| EC | European Commission |
| LarKC | The Large Knowledge Collider |
| OWL-S | Ontology Web Language for Services |
| PLA | Plug-in Level Agreement |
| QoS | Quality of Service |
| SAWSDL | Semantic Annotations for WSDL |
| SLA | Service Level Agreement |
| SUPER | Semantics Utilised for Process management within and between EnteRprises |
| WP | Work Package |
| WSDL | Web Services Description Language |
| WSMO | Web Service Modeling Ontology |

# 1. Introduction

Since one of the core objectives of LarKC is to provide an integrated pluggable platform for large-scale semantic computing, uniform and flexible access to the *identify*, *transform*, *select*, *reason*, and *decide* plug-ins (implementing services for retrieval, abstraction, selection, reasoning, and decision, respectively) is essential. Moreover, the expected large number of plug-ins requires automatic handling of plug-ins as far as their discovery and composition in the LarKC pipeline is concerned.

A typical concrete LarKC pipeline consists of a set of plug-ins composed to achieve a specific request. Two elements of plug-ins have been identified as crucial for their discovery and composition in a concrete pipeline: the capability of the plug-in (i.e. it's functionality) and the non-functional properties that characterize the parameters related to the quality of service of the plug-in. In this deliverable we aim to provide a mechanism for specifying such properties in an unambiguous way that could enable automatic discovery and composition of plug-ins in a reasoning pipeline.

Since a LarKC plug-in resembles the characteristics of a service, we look at plug-ins as services, which offer a certain capability with certain service level agreements and are accessible through standardized interfaces. In order to represent such services, in this deliverable we propose a language for specifying plug-ins as services. The main goals of such a language for plug-in descriptions are:

- To characterize functional properties of plug-ins (as services) and pipelines (as workflows, composed by plug-ins);
- To define the non-functional properties of both plug-ins and pipelines (including QoS properties)
- Describe QoS metrics (possible values of QoS parameters) to be used in the definition of plug-ins, pipelines and PLAs (Plug-in Level Agreements)
- To describe PLA Templates (plug-in providers will describe plug-in properties in the form of PLA Templates), which are used for negotiation between the platform (request) and the provider (offer).

Following the analysis of the project use cases and the initial operational framework, several requirements have been identified for the specification of properties of LarKC plug-ins needed to enable and manage the execution of the reasoning pipeline. The LarKC Framework must support the management of different kinds of "resources" to get approximate and/or anytime behaviour, including the plug-ins themselves and data and hardware resources where the plug-ins are executed. Therefore the descriptions of plug-ins (both between platform and plug-in and between plug-ins) must capture both their functional and non-functional properties. Both groups of properties are identified in the following sections and will be modelled as part of the so called plug-in annotation language. Plug-in properties can be seen from two different viewpoints:

- LarKC Requirements (or Preferences) from end-user perspective: these are the preferences the end-user may request/desire as overall performance of the LarKC Platform, independently of what is happening inside the pipeline execution.
- LarKC Properties from plug-in interface perspective: in order to achieve the required overall performance and the requested result, an appropriate pipeline must be constructed. For this purpose, the selection of plug-ins with concrete functionality (modelled with Functional Parameters) must be done, considering also their individual performance (modelled with non-Functional Parameters).

In order to translate end-user preferences to individual plug-in parameters, a mapping process must take place. Depending on the concrete use case, this may take place in different moments and locations, either automatically or manually:

- For instance, in the case of a manually composed pipeline, by a (human) pipeline designer, he is responsible for choosing the most appropriate plug-ins according to the end-user query and eventually other performance requirements. For this purpose, the plug-ins must make both functional and non-functional properties available, through their semantic descriptions.
- In the case of an end user of LarKC who simply wishes to get an answer to a given query, a fully automatic process may be established to map high-level (end user) requirements to individual plug-in parameters. In this case, a "super-intelligent" *decide* plug-in would be the ideal solution. The *decide* plug-in would decompose the end user query and performance requirements into requirements on individual plug-ins, e.g. end user requires an answer in max time = 5 min => *decide* plug-in decides to execute plug-in *select* with a performance of 1 Million triples/sec + *reason* plug-in executed inside a cluster, requesting maximum 4 min. of processing time.

A performance estimation process may take place prior the final deployment and execution of the plug-ins, so that the plug-in description includes performance parameters as reliably as possible. This can be done at different levels:

- Benchmarking of individual plug-ins, e.g. plug-in idx, executed in a cluster using 3 nodes, with input abc, takes an average of 3 minutes to generate an average of 100 results.
- Benchmarking of composed pipelines, e.g. the execution of the pipeline plug-in idx + plug-in idy + plug-in idz, etc.

This estimation process should be performed depending on the level to which it is considered: In the case of individual plug-ins, the plug-in developer should provide this information as part of the plug-in description; In the case of composed pipelines, there may be different actors performing this process, such as a pipeline designer or the LarKC platform provider. Overall, there are different possibilities for storing benchmarking information, e.g. in a central repository, as part of the description of every plug-in (for individual benchmarking), or learning from historical data (from previous executions).

The LarKC plug-in annotation language proposed in this deliverable is intended to be used by:

- Providers (Plug-in providers and Resource providers) for: (1) Characterization of the "service" they offer (functional and non-functional properties); (2) Annotation of QoS metrics within the PLA Templates and/or Plug-in Template (may be the same), characterizing the non-functional properties of the offered "service" (plug-in or resource), including parameters such as price, resource requirements; (3) Characterization of the "process" they offer (a plug-in provider may offer a sub-pipeline composed by several plug-ins)
- LarKC Platform, specifying its request through: QoS metrics in PLA Templates, Plug-in discovery, Plug-in invocation, Pipeline invocation, monitoring and controlling (pipeline enactment)
- *Decide* plug-in for Plug-in discovery.[1]

Looking at the general characteristics of the plug-ins identified above there seem to be various candidates for a language for service plug-ins. Various techniques for describing such services exist, ranging from syntactical approaches such as WSDL to more semantically enhanced, ontology-based approaches such as SAWSDL, OWL-S, or WSMO. Since ultimately we want to enable discovery and

[1] [1] It is still under discussion within the project which functionality must be implemented by the platform and which by the Decider plug-in. It might be the case that the Decider plug-in is in charge of discovering the appropriate plug-ins to fulfil the user query. The platform will be in charge of plug-in invocation and pipeline enactment.

composition of plug-ins exposed as services, an ontology-based approach for plug-in specification would allow for some degree of automation in the process of plug-in discovery and composition.

Amongst the potential candidates for such a service plug-in annotation language, WSMO-Lite appears to be the most suitable. It provides a minimalistic ontology for capturing service properties, being at same time the next evolutionary step after SAWSDL, filling the SAWSDL annotations with concrete semantic service descriptions. We will take WSMO-Lite as the basis for plug-in annotations and will extend it with specific plug-in non-functional properties.

The rest of this document is structured as follows. Section 2 provides a brief overview of WSMO-Lite, focusing on those parts that are specific in the context of LarKC plug-ins. Section 3 defines a set of non-functional properties (in terms of an ontology for plug-in non-functional properties) that apply to LarKC plug-ins. Section 4 provides an example of how WSMO-Lite and the identified non-functional properties can be used for modelling a LarKC plug-in. Section 5 shows how plug-ins can be chained to realize a typical concrete LarKC pipeline. Section 6 summarizes the deliverable.

## 2. WSMO-Lite for Plug-ins Annotation

WSMO-Lite is a minimal ontology language for service descriptions. WSMO-Lite is inspired by the Web Services Modelling Ontology (WSMO), however, it only focuses on a subset of it using it to define a gradual extension of SAWSDL. WSMO-Lite addresses the following requirements:

- Identify the types and a simple vocabulary for semantic descriptions of services (a service ontology) as well as languages used to define these descriptions.
- Define an annotation mechanism for WSDL using this service ontology.
- Provide the bridge between WSDL, SAWSDL and (existing) domain-specific ontologies such as classification schemas, domain ontology models, etc.

When dealing with semantic services in general, several aspects can be identified:

- *Information Model* defines the data model for input, output and fault messages, as well as for the data relevant to other aspects of the service description.
- *Functional Descriptions* define service functionality, that is, what a service can offer to its clients when it is invoked.
- *Non-Functional Descriptions* define any incidental details specific to a service provider or to the service implementation or its running environment. An example non-functional property is the price; the functionality of a service is generally not affected by the price, even though the desirability may be.
- *Behavioral Descriptions* define external and internal behavior. The former is the description of a public choreography, the protocol that a client needs to follow when consuming a service's functionality; the latter is a description of a workflow, i.e. how the functionality of the service is aggregated out of other services.

In the context of WSMO-Lite and LarKC plug-ins the following two aspects defined in WSMO-Lite are relevant for the descriptions of plug-ins:

- *Functional Descriptions* are represented as *capabilities* and/or functional *classifications*. A capability defines *conditions* that must hold in a state before a client can invoke the service, and *effects* which hold in a state after the service invocation. *Classifications* define the service functionality using some classification ontology (i.e. a hierarchy of categories).
- *Non-Functional Descriptions* are represented using an ontology, semantically representing some policy or other non-functional properties.

To actually specify the above, WSMO-Lite introduces two core elements to capture the above two aspects, namely:

- *wl:FunctionalClassificationRoot* is a class that is the root of all classifications related using RDFS subclass. A classification (taxonomy) of service functionalities can be used for functional description of a service.
- *wl:NonFunctionalParameter* specifies a placeholder for a concrete domain specific non-functional property. In the case of LarKC plug-ins, such domain specific non-functional properties are defined in Section 3.

We will use these elements to represent a *select* plug-in in Section 4.

## 3. An Ontology for Plug-ins Non-functional Properties

Although a number of other research projects are using ontologies to manage QoS aspects and other non-functional properties for services, there exists no single standard ontology for QoS. The BREIN project has just released *D3.2.5 Final Report on BREIN Core Ontologies* [1]. This set of ontologies has been analysed within LarKC in order to define the LarKC ontology for plug-in annotations. It has not been considered appropriate to reuse the BREIN ontologies as such, as they are more complex and extensive than the one needed in LarKC, due to the different fields of application of both projects. However, some similarities have been found in the general concepts, which made part of the BREIN ontologies a good model to be followed from a conceptual viewpoint. The BREIN Core Ontologies extend and/or take experiences, among others, from the following ontologies:

- OWL-S to describe services;
- OWL-WS is used in BREIN in the Workflow Designer tool in order to store the abstract workflows without implementation details;
- SUPER project ontologies are used for description of business processes [2];
- Grid Resource Ontology [3] was developed by the UniGrids project, in collaboration with OntoGrid, for the definition and management of Grid resources and middleware.

The WSMO working group has defined an ontology for modelling web services [4]. This ontology has been also analysed and considered as a basis for the definition of the LarKC Plug-in Ontology.

Within LarKC, the non-functional properties identified have been grouped in different categories, based on the grouping made in previous services projects, mainly [1] and [4], but adapted to the concrete LarKC plug-ins and pipeline requirements. LarKC deliverable D6.2 “Templates of periodic report on data and performances” [10] has been also considered as an input for this ontology. That deliverable includes a list of parameters to measure the performance of the Urban Computing application. Some of them are purely data related, but some others are also related to pipeline and/or plug-in non-functional properties and therefore have been considered here. In the following we present a list of non-functional properties that form the core ontology of non-functional properties for plug-ins. After every parameter, in brackets it is indicated whether this parameter is applicable to the complete pipeline execution (from the end-user viewpoint), or to a single plug-in (plug-in interface viewpoint), and for some of them also to which type of plug-in. For some parameters, also possible range of values is indicated.

**Performance**: It is related to how fast a service request can be completed and at which rate the results are generated. Performance can be measured from different perspectives and therefore, a number of different parameters are included in this group.

- *MaxTime_FinalAnswer* (end-user): Maximum time required to obtain all possible results to the given query.

- *MaxTime_FirstAnswer* (end-user): Maximum time required to obtain the first answer (out of a set of results)
- *DesiredNResults* (end-user): Desired number of results, as minimum.
- *MaxNResults* (end-user): Maximum number of results required. Over this number, the execution can be stopped.
- *Throughput* (plug-in): estimated rate of generated results. It will be measured in different units, depending on the concrete type of plug-in:
  - Identify: identified resources (e.g RDF Triples) per time unit (e.g. second)
  - Transform (data): transformed resources per time unit
  - Select: selected resources per time unit
  - Reason: answers per time unit
- *Cardinality* (*transform* plug-in): Cardinality of the data transformation
  - 1:1
  - 1:many
  - many:1
- *OtherPerformanceFunction*: for some plug-ins, other kind of performance function may be provided by the plug-in provider and used by the LarKC *decide* plug-in or a pipeline designer in order to select the most appropriate plug-in for a given query.

**Accuracy:** It is related to the accuracy of the results provided by the plug-ins.

- *Completeness* (end-user, *reason* plug-in): indicates whether completeness is required (from end-user viewpoint) or achieved (by the given plug-in); Possible values: yes/no.
- *Soundness* (end-user, *reason* plug-in): indicates whether soundness is required (from end-user viewpoint) or achieved (by the given plug-in); Possible values: yes/no.
- *Recall* [5]: a measure of completeness. True Positive / (True Positive + False Positive)
- *Precision* [6]: a measure of exactness or fidelity. True Positive / (True Positive + False Negative).
- *F-measure* [7]: the weighted harmonic mean of precision and recall. $(1+beta^2)$ precision recall / ($beta^2$precision + recall) (e.g., beta = 1).
- *ROC curve* [8]: Receiver Operating Characteristic is a graphical plot of the sensitivity vs. (1 - specificity).
- *AUC*: the area under the ROC curve.

**Infrastructure**: It is related to the usage of resources and the infrastructure where the plug-ins are executed.

- *TightToEE* (plug-in): the plug-in must run in the concrete execution environment the provider imposes (e.g. there may be a plug-in designed to run only in a cluster environment with concrete resources requirements).
- *MinNNodes* (plug-in): Minimum number of processing nodes the execution environment must have in order the plug-in to run appropriately.
- *MaxNNodes* (plug-in): Maximum number of processing nodes in the execution environment to achieve the optimal performance (over that number, the performance is not improved any more). MaxNNodes = 1=> plug-in is not parallelisable
- *MinMemory* (plug-in): Minimum memory required for the correct execution
- *MaxMemory* (plug-in): Maximum memory consumption by the plug-in execution
- *DeploymentFactor* (plug-in): Minimum speed of connection needed to the plug-in external world (e.g. to data storage, to the platform)

**Scalability**: It indicates the scalability rate of the given plug-in. The measurement unit and degrees of scalability depend on the concrete plug-in type.

- *Scalability* (*transform* plug-in): Scalability rate of the *transform* plug-in (based on size of the ontology)
- *Scalability* (*reason* plug-in): Scalability rate of the *reason* plug-in. Possible values:
  - good/bad
  - abox/tbox
  - linear, sub-linear, polynomial

**Financial**: It involves terms related to cost-related and charging-related properties of a plug-in.

- *Cost* (plug-in): For simplicity in this first version of the ontology it is a single parameter. However, this may be a more complex property, including charging styles related to the concrete business model between the involved stakeholders (plug-in provider, resource provider, LarKC provider, etc). This parameter is thought for LarKC external commercial plug-ins.

## 4. Plug-in Annotation Example

In this Section we show how to use the notions introduced in the previous sections (i.e. how to use the WSMO-Lite annotation elements and a set of non-functional properties) to model a *select* plug-in. Please note that the elements used in the notation to model the example reflect the abstraction as it was at the time of writing this deliverable.

We start by defining a general LarKC plug-in ontology (lines 1-92) which defines the various plug-in types (lines 10-21), a set of data types (lines 23-43), input and output types (lines 47-55), a special case of each data types for the case when they serve as input or output for the plug-in (lines 58-79), and a set of predefined non-functional properties (lines 82-92).

Once this general ontology is defined, we can create a service description for a *select* plug-in, called “GrabEverythingSelecter” (lines 97-114). Lines 99-103 describe the part of the WSDL file (interface and binding) that is common to all *select* plug-ins. Note the use of the SAWSDL modelReference element in the description of the interface, which gives a predefined meaning for the interface of this service (i.e. the fact that it is a *select* plug-in). In contrast to the information that is common to all *select* plug-ins, Lines 105-113 provide information specific to the “GrabEverythingSelecter”. Note the use of a SAWSDL modelReference which attaches non-functional properties to this specific *select* plug-in. To complete the specification, the ontology supplied with the “GrabEverythingSelecter” plug-in is described in lines 112-136.

```
@prefix larkc: <http://larkc.eu/plugin#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix wsl:   <http://www.wsmo.org/ns/wsmo-lite#> .
@prefix xs:   <http://www.w3.org/2001/XMLSchema#> .

# Functional properties

# Plug-in types
larkc:Plugin   rdf:type .

larkc:Identifier  rdf:type         rdfs:Class ;
                  rdfs:subClassOf  larkc:Plugin .
larkc:Transformer rdf:type         rdfs:Class;
                  rdfs:subClassOf  larkc:Plugin .
larkc:Selecter    rdf:type         rdfs:Class;
                  rdfs:subClassOf  larkc:Plugin .
larkc:Reasoner    rdf:type         rdfs:Class;
                  rdfs:subClassOf  larkc:Plugin .
larkc:Decider     rdf:type         rdfs:Class;
                  rdfs:subClassOf  larkc:Plugin .

# Data type hierarchy
larkc:Resource        rdf:type rdfs:Class .

larkc:InformationSet  rdf:type        rdfs:Class ;
                      rdfs:subClassOf larkc:Resource .
larkc:RdfGraph        rdf:type        rdfs:Class ;
                      rdfs:subClassOf larkc:InformationSet .
larkc:PatternEnabledGraph rdf:type    rdfs:Class ;
                      rdfs:subClassOf larkc:RdfGraph .
larkc:RemoteRdf       rdf:type        rdfs:Class ;
                      rdfs:subClassOf larkc:PatternEnabledGraph .
larkc:NaturalLanguageDocument rdf:type rdfs:Class ;
                      rdfs:subClassOf larkc:InformationSet .

larkc:Query           rdf:type        rdfs:Class ;
                      rdfs:subClassOf larkc:Resource .
larkc:SPARQLQuery     rdf:type        rdfs:Class ;
                      rdfs:subClassOf larkc:Query .
larkc:TriplePatternQuery rdf:type     rdfs:Class ;
                      rdfs:subClassOf larkc:Query .
larkc:KeywordQuery    rdf:type        rdfs:Class ;
                      rdfs:subClassOf larkc:Query .

# Input / Output handling
larkc:InputType   rdf:type  rdfs:Class, wsl:ClassificationRoot .
larkc:OutputType  rdf:type  rdfs:Class, wsl:ClassificationRoot .

larkc:isInputType   rdf:type    rdf:Property ;
                    rdfs:domain larkc:Resource .
                    rdfs:range  larkc:InputType .
larkc:isOutputType  rdf:type    rdf:Property;
                    rdfs:domain larkc:Resource .
                    rdfs:range  larkc:InputType .

# Input data types
larkc:InputInformationSet rdf:type          rdfs:Class ;
                          larkc:isInputType larkc:InformationSet .
larkc:InputRdfGraph       rdf:type          rdfs:Class ;
                          larkc:isInputType larkc:RdfGraph        .
larkc:InputPatternEnabledGraph rdf:type     rdfs:Class ;
                          larkc:isInputType larkc:PatternEnabledGraph .
larkc:InputRemoteRdf      rdf:type          rdfs:Class ;
                          larkc:isInputType larkc:RemoteRdf .
larkc:InputNaturalLanguageDocument    rdf:type rdfs:Class ;
                          larkc:isInputType larkc:InputInformationSet .

# Output data types
larkc:OutputInformationSet rdf:type           rdfs:Class ;
                           larkc:isOutputType larkc:InformationSet .
larkc:OutputRdfGraph       rdf:type           rdfs:Class ;
                           larkc:isOutputType larkc:RdfGraph        .
larkc:OutputPatternEnabledGraph rdf:type      rdfs:Class ;
                           larkc:isOutputType larkc:PatternEnabledGraph .
larkc:OutputRemoteRdf      rdf:type           rdfs:Class ;
```

```
                            larkc:isOutputType larkc:RemoteRdf .
larkc:OutputNaturalLanguageDocument   rdf:type rdfs:Class ;
                            larkc:isOutputType larkc:OutputInformationSet .

# Non-functional properties go here
larkc:Scalability rdf:type rdfs:Class ;
                  rdfs:subClassOf wsl:NonFunctionalProperty .
larkc:Cost        rdf:type rdfs:Class ;
                  rdfs:subClassOf wsl:NonFunctionalProperty .

larkc:hasCostPerInvocation rdf:type rdf:Property ;
                           rdfs:domain larkc:Cost .
larkc:hasCostPerInvocation rdfs:range  larkc:euro .

larkc:hasScalability rdfs:domain larkc:Scalability .
larkc:hasScalability rdfs:range  xs:string .


# The WSDL description for the GrabEverythingSelecter:

<wsdl:description>
    <!-- COMMON TO ALL SELECT PLUG-INS -->
    <wsdl:interface name="selecter"
                    sawsdl:modelReference="http://larkc.eu/plugin#Selecter">
    </wsdl:interface>

    <wsdl:binding name="larkcbinding" type="http://larkc.eu/wsdl-binding" />
    <!-- SPECIFIC TO THIS SELECT PLUG-IN -->
    <wsdl:service name="GrabEverythingSelecter" interface="selecter"
  sawsdl:modelReference="http://example.com/GrabEverythingSelecter1#GrabEverythingSelecter
                         http://example.com/GrabEverythingSelecter1#scalability
                         http://example.com/GrabEverythingSelecter1#cost
                         http://larkc.eu/plugin#InputRdfGraph
                         http://larkc.eu/plugin#OutputInformationSet" >

        <wsdl:endpoint location="rmi:<an rmi example>" />
    </wsdl:service>
</wsdl:description>


# The ontology supplied with the GrabEverythingSelecter plug-in:

@prefix g1: <http://example.com/GrabEverythingSelecter1#> .
@prefix larkc: <http://larkc.eu/plugin#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

g1:GrabEverythingSelecter rdf:type                  rdfs:Class ;
                          rdfs:subClassOf           larkc:Selecter .
g1:scalability            rdf:type                  larkc:Scalability ;
                          larkc:hasScalability      "low" .
g1:cost                   rdf:type                  larkc:Cost ;
                          larkc:hasCostPerInvocation "0.01"^^larkc:euro .
```

# 5. Plug-ins Composition

In the initial version of the platform the LarKC plug-in ontology and the descriptions of the plug-ins are loaded into the platform knowledge base when the platform starts. Other parts of the platform and the plug-ins can query this database to retrieve the knowledge about other plug-ins and about the non-functional parameters (e.g. available resources). This is functionality is primarily meant for the *decide* plug-in but can be accessed by other plug-ins and parts of platform as well.

A subset of Cyc is used as implementation of the knowledge base. Because Cyc reasoning engine is part of the knowledge base complex queries can be fired against the knowledge base. *LarKC decide* is one particular implementation of *decide* plug-in and can automatically assemble and compose plug-ins into working pipelines which can solve the queries passed to the platform. For this task, *LarKC decide* heavily relies on the Cyc reasoning engine.

In the initial version of the platform, the logic of *LarKC decide* logic is based on the input and output types of the plug-ins. Given a query, the decider finds sets of plug-ins which can be connected into a pipeline and can produce an answer to the query. The decider can also handle some of the non-functional parameters such as number of results in order to satisfy QoS parameters such as *DesiredNResults* or *MaxNResults*. This parameters serve as constraints when selecting appropriate plug-in. Example of this would be plug-ins which do not offer required level of completeness or scalability, lack resources (e.g. require too much memory), or are just too expensive to for the user. As one pipeline is running, QoS parameters are also used to determine if and when to start another branch in the pipeline in order to fulfil requested QoS (e.g. to match the required speed or number of results).

Figure 1 shows an example of all possible links between plug-ins from Baby LarKC as seen by the *decide* plug-in. Any walk on the graph which is starting either from *indetify* plug-in either from *query-transform* plug-in is a valid pipeline and can be executed. Additional QoS constraints effectively reduce the amount of links in this graph. Please note that the *decide* plug-in does not appear in the graph. There is only one active *decide* plug-in in the platform at a time which assembles the graph and executes specific paths in the graph in order to conform to the QoS constraints.

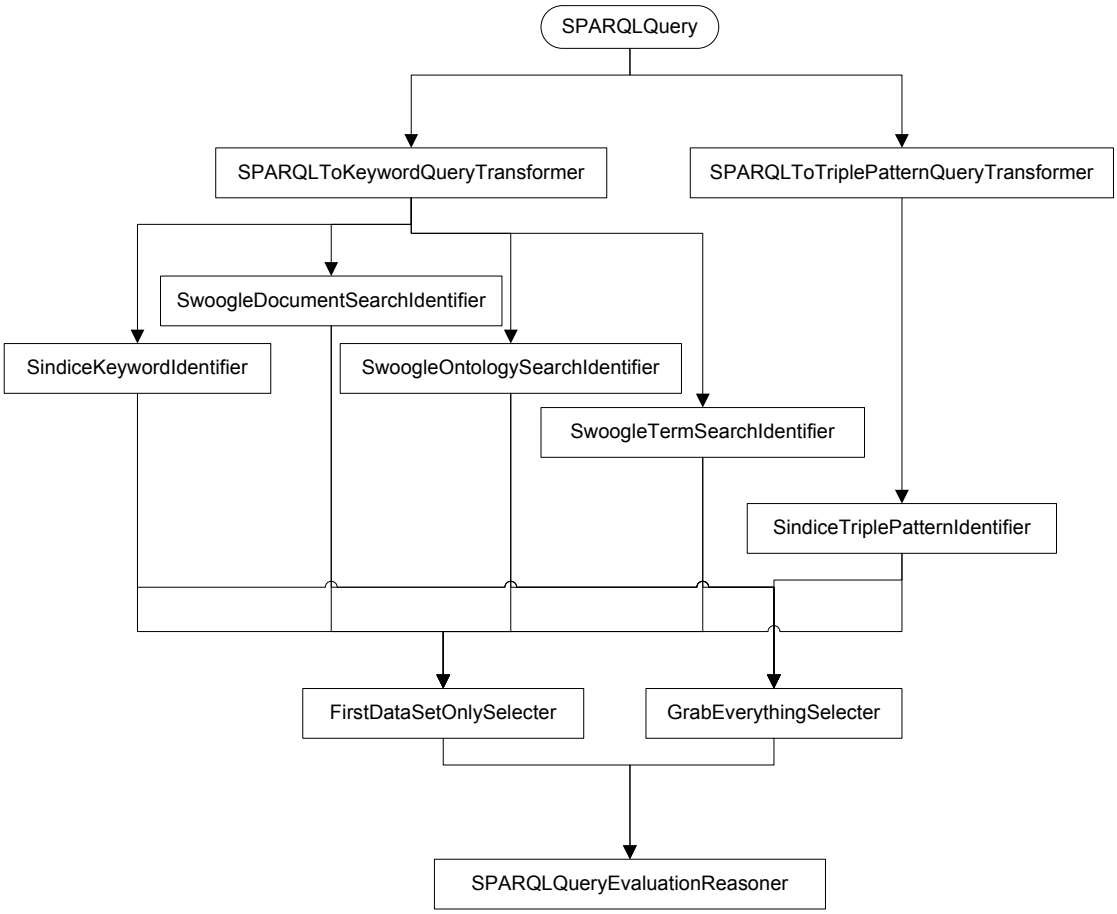


**Figure 1. All possible connections between plug-ins from Baby LarKC as seen by decide plug-in.**

## 6. Conclusions

In this deliverable we provided a first version of the plug-in annotation language. We based our plug-in descriptions on the WSMO-Lite annotation mechanism and provided a set of plug-in specific non-functional properties. We showed an example of a *select* plug-in and showed how it can be modelled with the plug-in annotation language. The annotation language was further used in the example of intelligent *decide* plug-in which can automatically assemble/compose plug-ins into working pipelines and can monitor them to fulfil the provided QoS constraints.

## 7. References

[1] The BREIN Project: http://www.eu-brein.com/, D3.2.5 Final Report on BREIN Core Ontologies
[2] http://www.ip-super.org
[3] http://www.unigrids.org/ontology.html
[4] D2v1.4. Web Service Modeling Ontology (WSMO) (WSMO Working Draft 16 February 2007) (http://www.wsmo.org/TR/d2/v1.4/20070216/)
[5] http://en.wikipedia.org/wiki/Recall_(information_retrieval)
[6] http://en.wikipedia.org/wiki/Precision_(information_retrieval)
[7] http://en.wikipedia.org/wiki/Precision_and_recall#F-measure
[8] http://en.wikipedia.org/wiki/Receiver_operating_characteristic
[9] http://en.wikipedia.org/wiki/Standard_error_(statistics)


[10] LarKC Deliverable: Kono Kim et al., D6.2 Templates of periodic report on data and performances