LarKC Plug-in Annotation Language

Dumitru Roman, Barry Bishop, Ioan Toma
Semantic Technology Institute (STI),
University of Innsbruck,
Innsbruck, Austria
{firstname.lastname}@sti2.at

Georgina Gallizo[1], Blaž Fortuna[2]
[1]High Performance Computing Center Stuttgart
(HLRS), Germany
gallizo@hlrs.de
[2]Cyc, Slovenia
blaz@cycorp.eu

***Abstract*—The aim of the Large Knowledge Collider (LarKC) project is to develop a platform for massive distributed incomplete reasoning for the Semantic Web. The LarKC Plug-ins – services that can be used in the LarKC platform play a key role in the context of LarKC. They are the core elements that are composed in a concrete LarKC pipeline – a particular configuration of LarKC plug-ins that enables massive distributed reasoning under various configurations of reasoners and other elements. 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. This paper describes an initial mechanism for specifying plug-ins as semantically enriched Web services. We propose WSMO-Lite as a basis for specifying 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.**

I. Introduction and Motivation

With the emergence of the Semantic Web, large amounts of information available on the Web could be processed based on the formal semantics attached to it. A set of languages that use logic for this purpose has been developed for the Semantic Web. However, current logic based reasoning systems do not scale to the amount of information available on the Web. This scalability limitation requires the development of a reasoning infrastructure that can scale and that can be flexibly adapted to the varying requirements of quality and scale of different potential use cases such as analyzing data from millions of mobile devices, dealing with terabytes of scientific data, etc. The Large Knowledge Collider (LarKC) project [1] addresses this problem of reasoning scalability on the Semantic Web by aiming to develop a platform for massive distributed incomplete reasoning that will remove the scalability barriers of currently existing reasoning systems for the Semantic Web.

[1] http://www.larkc.eu/

The LarKC platform is designed as a pluggable algorithmic framework implemented on a distributed computational platform. LarKC is aimed at exploiting not only logic-based inference systems, but also a large variety of methods from other fields such as cognitive science (human heuristics), economics (limited rationality and cost/benefit tradeoffs), information retrieval (recall/precision trade-offs), and databases (very large datasets). This pluggability feature of the LarKC architecture will ensure that computational methods from these different fields can be coherently integrated. A large number of such computational methods are envisioned to be made available on the Web, and it will be the responsibility of the LarKC platform to combine them in a meaningful way in order to improve the scalability of reasoning on the Semantic Web. We refer to such computational methods made available to the LarKC platform as *plug-ins*, and to their compositions in the LarKC platform as *pipelines*.

Uniform and flexible access to various types of plug-ins such as *identify*, *transform*, *select*, *reason*, and *decide* (implementing services for retrieval, abstraction, selection, reasoning, and decision, respectively) is essential. Moreover, the expected large number of plug-ins requires automatic handling as far as their discovery and composition in the LarKC pipeline is concerned. Plug-in descriptions/annotations play a crucial role in enabling such automation and flexible access. This paper is basically motivated by the lack of a clear set of elements/parameters that should be part of such descriptions, and the lack of a mechanism for attaching semantic descriptions/annotations to plug-ins. The approach we have taken in this paper to address these issue is to identify, based on the various use cases of the LarKC project, a set of common elements that characterize LarKC plug-ins, and then define an annotation mechanism that attaches semantic descriptions of various properties to plug-ins. This mechanism is based on the fact that we consider plug-ins as Web services, and therefore we can make use of existing approaches in the area of semantic Web services to provide formal descriptions of plug-ins. Our choice for service annotation language for plug-ins is WSMO-Lite – a minimal ontology language for describing service capabilities and other service characteristics. Our

choice of WSMO-Lite is motivated by its simple and extensible service model.

The rest of this paper is structured as follows. Section II provides more details on LarKC plug-ins. Section III provides a brief overview of WSMO-Lite, focusing on those parts that are relevant in the context of LarKC. Section IV defines an ontology for non-functional properties. Section V provides an example of how WSMO-Lite and the identified non-functional properties can be used for modeling a LarKC plug-in. Section VI shows how plug-ins can be chained to realize a typical concrete LarKC pipeline. Section VII summarizes the paper.

II. LARKC PLUG-INS

Based on the LarKC project use cases, 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., its functionality) and the *non-functional properties* that characterize the parameters related to the quality of service of the plug-in. In this paper, 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. In order to represent such services, we propose a language for specifying plug-ins as services, the main goals of which are: (1) To *characterize functional properties* of plug-ins (as services) and pipelines (as workflows, composed by plug-ins); (2) To *define the non-functional properties* of both plug-ins and pipelines (including QoS properties); (3) To *describe QoS metrics* (possible values of QoS parameters) to be used in the definition of plug-ins.

Several requirements have been identified for the specification of LarKC plug-in properties in order to 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 behavior, including the plug-ins themselves and data and hardware resources. Therefore the descriptions of plug-ins must capture both their functional and non-functional properties.

Plug-in properties can be seen from two different viewpoints: (1) from the end-user perspective (the preferences the end-user may request as overall performance of the LarKC Platform, independent of what is happening inside the pipeline execution), and (2) from a 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 must be done, considering also their individual performance, e.g. in terms of throughput).

In order to translate end-user preferences into 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 other performance requirements. For this purpose, both functional and non-functional properties of plug-ins must be available. 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. A *decide* plug-in could 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: (1) Benchmarking of individual plug-ins and (2) Benchmarking of composed pipelines.

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

- Plug-in providers for: (1) Characterizing the "service" they offer (functional and non-functional properties); (2) Annotation of QoS metrics characterizing the non-functional properties of the offered "service"
- LarKC Platform for Plug-in discovery, Plug-in invocation, Pipeline invocation, monitoring and controlling (pipeline enactment);
- *Decide* plug-in for Plug-in discovery.

Looking at the general characteristics of the plug-ins identified above there seem to be various candidates for a plug-in description language. Various techniques for describing 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 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 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. We will take WSMO-Lite as the basis for plug-in annotations and will extend it with specific plug-in non-functional properties.

III. WSMO-LITE FOR PLUG-IN 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 using it to define a gradual extension of SAWSDL. WSMO-Lite addresses the following requirements: (1) Identify the types and a simple vocabulary for semantic descriptions of services (a service ontology) as well as languages used to define these descriptions; (2) Define an annotation mechanism for WSDL using this service ontology; (3) 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), *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), and *Behavioral Descriptions* (define external and internal behavior). 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:[2]

- *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 descriptions 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 IV.

IV. AN ONTOLOGY FOR PLUG-IN NON-FUNCTIONAL PROPERTIES

Although a number of 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 proposed a set of ontologies in [1]. This set of ontologies has been analyzed 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 follow from a conceptual viewpoint. The BREIN Core Ontologies extend and/or take experience from ontologies such as OWL-S, OWL-WS, SUPER ontologies [2], and the Grid Resource Ontology [3]. The WSMO working group has defined an ontology for modeling Web services [4]. This ontology has been also analyzed 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 to different categories, similar to the grouping made in previous services projects, but adapted to the concrete LarKC plug-ins and pipeline requirements. LarKC deliverable D6.2 “Templates of periodic report on data and performance” [5] has also been 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.[3]

Performance (how fast a service request can be completed and at which rate the results are generated):

[2] We will use these elements to represent a select plug-in in Section V.

[3] 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.

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, measured in different units, depending on the type of plug-in: Identify, Transform, etc.
Cardinality (*transform* plug-in)	Cardinality of the data transformation, which can be 1:1, 1:many, many:1, or many:many.
OtherPerformanceFunction	For some plug-ins, another kind of performance function may be provided and used by the LarKC *decide* plug-in to select the most appropriate plug-in for a given query.

Accuracy (the accuracy of the results provided by the plug-ins):

Completeness (end-user, *reason* plug-in)	Indicates whether completeness is required (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 (end-user viewpoint) or achieved (by the given plug-in); Possible values: yes/no.
Recall	A measure of completeness. True Positive / (True Positive + False Positive)
Precision	A measure of exactness or fidelity. True Positive / (True Positive + False Negative).
F-measure	The weighted harmonic mean of precision and recall. (1+beta²) precision recall / (beta²precision + recall) (e.g., beta = 1).
ROC curve	Receiver Operating Characteristic is a graphical plot of the sensitivity vs. (1 - specificity).
AUC	The area under the ROC curve.

Infrastructure (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).
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 (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 are good/bad, abox/tbox, linear, sub-linear, polynomial.

Financial (cost-related and charging-related properties of a plug-in):

Cost (plug-in)	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.

V. 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. We start by defining a general LarKC plug-in ontology that defines the various plug-in types, a set of data types, input and output types, a special case of each data type for the case when they serve as input or output for the plug-in, and a set of predefined non-functional properties:

```
@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
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 .
```

Once this general ontology is defined, we can create a service description for a *select* plug-in, called "GrabEverythingSelecter". The following listing describes 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).

```
# 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" />
```

In contrast to the information that is common to all *select* plug-ins, the following listing provides information specific to the "GrabEverythingSelecter". Note the use of a SAWSDL modelReference which attaches non-functional properties to this specific *select* plug-in.

```
<!-- 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>
```

To complete the specification, the ontology supplied with the "GrabEverythingSelecter" plug-in is specified as follows:

```
# 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.
```

VI. PLUG-IN 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 functionality is primarily meant for the *decide* plug-in but can be accessed by other plug-ins and parts of the platform as well.

A subset of Cyc [5] is used for the implementation of the knowledge base. Because the Cyc reasoning engine is part of the knowledge base, complex queries can be asked of it. A *LarKC decide* is one particular implementation of a *decide* plug-in and can automatically assemble and compose plug-ins into working pipelines that 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 reasoning of *LarKC decide* is based on the input and output types of the plug-ins. Given a query, the decider finds sets of plug-ins that 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*. These parameters serve as constraints when selecting appropriate plug-ins. An example of this would be plug-ins that do not offer the required level of completeness or scalability, lack resources (e.g. require too much memory), or are just too expensive 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 fulfill the 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 LarKC as seen by the *decide* plug-in. Any path on the graph that starts either from the *indentify* plug-in or from the *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 any one time that assembles the graph and executes specific paths in the graph in order to conform to the QoS constraints.

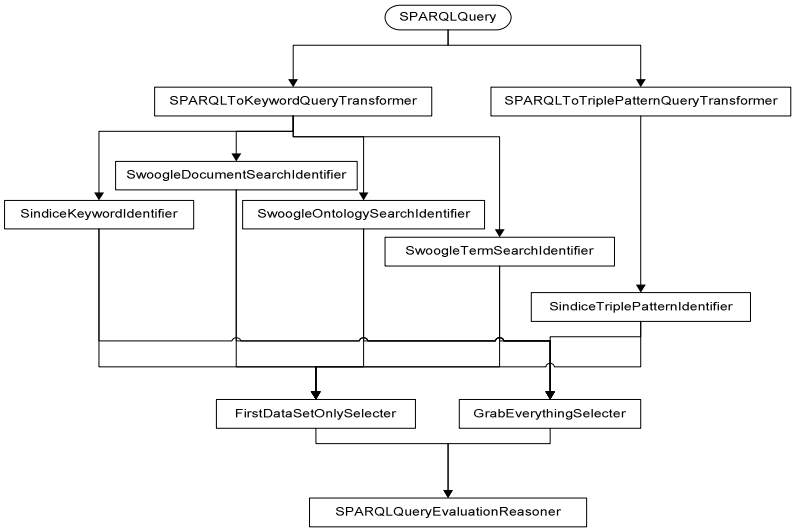

Figure 1. Possible connections between plug-ins as seen by a *decide* plug-in.

VII. SUMMARY AND CONCLUSION

In this paper we provided a first version of the LarKC 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 modeled with the plug-in annotation language. The annotation language was further used in the example of an intelligent *decide* plug-in that can automatically assemble/compose plug-ins into working pipelines and can monitor them to fulfill the provided QoS constraints.

REFERENCES

[1] I. Kotsiopoulos (Ed.): *Final Report on BREIN Core Ontologies*, 2008. BREIN Deliverable D3.2.5. Available at http://www.eu-brein.com/index.php?option=com_docman&task=doc_download&gid=57.

[2] Semantics Utilized for Process management within and between Enterprises (SUPER) Project, 2009. Information available at http://www.ip-super.org

[3] Uniform Interface to Grid Services Ontology, 2009. Available at http://www.unigrids.org/ontology.html.

[4] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, D. Fensel: Web Service Modeling Ontology. Applied Ontology 1(1): 77-106 (2005).

[5] K. Kim (Ed.): *Templates of periodic report on data and performances*. LarKC Project deliverable D6.2, 2009. Available at http://www.larkc.eu/wp-content/uploads/2009/01/lark_d62_templates-of-periodic-report-on-data-and-performances.pdf.

[6] The Cyc Knowledge Server, 2009. Available at http://www.cyc.com/cyc/technology/whatiscyc.