



LarKC

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

FP7 – 215535

D5.2.1 Rapid Prototype of the LarKC

Coordinator: Georgina Gallizo (HLRS)

**With contributions from: Mick Kerrigan (UIBK); Luka Bradesko,
Blaz Fortuna (Cycorp Europe)**

Document Identifier:	LarKC/2009/D5.2.1 /v1.0
Class Deliverable:	LarKC EU-IST-2008-215535
Version:	1.0
Date:	30.01.2009
State:	Final
Distribution:	Public



EXECUTIVE SUMMARY

This deliverable aims to give an overview of the Rapid Prototype of the LarKC Collider Platform. Due to the nature of the deliverable (Prototype), this is not an extensive architecture specification, but intends to complement the Prototype code as an aid to understanding the strategy followed for its design and overall architecture and functionality. This document also includes some guidelines on how to install and use the prototype.



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	5.2.1	Title	Rapid Prototype of the LarKC
Work Package	Number	5	Title	The Collider Platform

Date of Delivery	Contractual	M 10	Actual	
Status	version 1.0		final	
Nature	prototype <input checked="" type="checkbox"/> report <input type="checkbox"/> dissemination <input type="checkbox"/>			
Dissemination level	public <input type="checkbox"/> consortium <input type="checkbox"/>			

Authors (Partner)	Georgina Gallizo (HLRS); Mick Kerrigan, Barry Bishop (UIBK); Luka Bradesko, Blaz Fortuna (Cycorp Europe)			
Responsible Author	Name	Georgina Gallizo	E-mail	gallizo@hlrs.de
	Partner	HLRS	Phone	












Abstract (for dissemination)	This deliverable aims to give an overview of the Rapid Prototype of the LarKC Collider Platform. Due to the nature of the deliverable (Prototype), this is not an extensive architecture specification, but intends to complement the Prototype code as an aid to understanding the strategy followed for its design and overall architecture and functionality. This document also includes some guidelines on how to install and use the prototype.
Keywords	Prototype, Platform, Baby LarKC

Version Log			
Issue Date	Rev. No.	Author	Change
12.12.2008	0.1	Georgina Gallizo	First draft
16.12.2008	0.2	Mick Kerrigan	Added Assigned Content
10.01.2009	0.3	Blaz Fortuna, Luka Bradesko	Added Assigned Content
13.01.2009	0.4	Michael Witbrock	Editing
22.01.2009	0.5	Georgina Gallizo	Final check and ready for QA



27.01.2009	0.6	Frank van Harmelen, Zhisheng Huang	Quality Assessment
30.01.2009	0.7	Georgina Gallizo, Barry Bishop	Quality Assessment comments incorporated
30.01.2009	1.0	Georgina Gallizo	Final version to be submitted to EC

PROJECT CONSORTIUM INFORMATION

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






VRIJE UNIVERSITEIT AMSTERDAM	 <p>The logo of Vrije Universiteit Amsterdam, featuring a blue eagle with spread wings above the text 'vrije Universiteit'.</p>	Prof. Dr. Frank van Harmelen, VRIJE UNIVERSITEIT AMSTERDAM, Amsterdam, Netherlands, Email: Frank.van.Harmelen@cs.vu.nl
THE INTERNATIONAL WIC INSTITUTE, BEIJING UNIVERSITY OF TECHNOLOGY	 <p>Two circular logos: the left one is the Beijing University of Technology logo (1960), and the right one is the Web Intelligence Consortium logo (2002).</p>	Prof. Dr. Ning Zhong, THE INTERNATIONAL WIC INSTITUTE, Mabeshi, Japan, Email: zhong@maebashi-it.ac.jp
INTERNATIONAL AGENCY FOR RESEARCH ON CANCER	 <p>The logo of the International Agency for Research on Cancer (IARC), featuring a caduceus (a staff with two snakes) inside a circular frame.</p>	Dr. Paul Brennan, INTERNATIONAL AGENCY FOR RESEARCH ON CANCER, Lyon, France, Email: brennan@iarc.fr



TABLE OF CONTENTS

LIST OF FIGURES.....	8
ACRONYMS.....	9
1. INTRODUCTION	10
2. THE LARKC PLATFORM: OVERALL CONCEPT	10
3. DESIGN GOALS FOR THE RAPID PROTOTYPE.....	10
4. DESIGN STRATEGY OF THE RAPID PROTOTYPE.....	11
5. PROTOTYPE IMPLEMENTATION: THE BABY LARKC	12
5.1. API SPECIFICATION	13
5.2. SCRIPTED DECIDE PLATFORM.....	16
5.3. SELF-CONFIGURING DECIDE PLATFORM.....	17
6. INSTALLATION GUIDE AND USER MANUAL	19
6.1. BABY LARKC WITH SCRIPTED DECIDER	20
6.2. BABY LARKC WITH SELF-CONFIGURING DECIDER	20
6.2.1. <i>Installation guide</i>	20
6.2.2. <i>User guide</i>	21
7. CONCLUSIONS AND NEXT STEPS	22
8. REFERENCES.....	23



List of Figures

Figure 1 LarKC Platform and plug-in model	10
Figure 2 Design Strategy of the LarKC Rapid Prototype.....	11
Figure 3 LarKC pipeline design evolution (solid arrows are data-flow, dashed arrows are control flow).....	12
Figure 4 The LarKC data model.....	16
Figure 5 Baby LarKC with Scripted DECIDE	17
Figure 6 All assertion for a Sindice Keyword Identifier plug-in.....	18
Figure 7 Inferred possible compositions of plug-ins.	18
Figure 8 Two different pipelines which can answer same type of queries using different plug-ins.	19
Figure 9 Baby LarKC Web Interface	20



Acronyms

Acronym	Definition
EC	European Commission
LarKC	The Large Knowledge Collider
QoS	Quality of Service
WP	Work Package

1. Introduction

This deliverable aims to give an overview of the Rapid Prototype of the LarKC Collider Platform. Due to the nature of the deliverable (Prototype), this is not intended to be an extensive architecture specification, but to complement the Prototype code as an aid to understanding the strategy followed for its design and overall architecture and functionality. This document also includes some guidelines on how to install and use the prototype.

The document is structured as follows:

- Chapter 2 introduces the overall concept of the LarKC Collider Platform
- Chapter 3 states the main goals of the Rapid Prototype
- Chapter 4 describes the strategy followed in the design of the Rapid Prototype
- Chapter 5 gives an overview of the prototype implementation, describing main components.
- Chapter 6 includes the installation guide and user manual of the Rapid Prototype

2. The LarKC Platform: Overall concept

As described in deliverable D1.2.1 [1], the LarKC platform provides a plug-in based framework for massive distributed incomplete reasoning at Web-scale. The overall concept of the LarKC platform is illustrated in Figure 1. The platform permits the incorporation of multiple, diverse plug-ins within the execution pipeline. LarKC plug-ins are provided by multiple stakeholders, and provide distinct capabilities that can be classified into five broad types.

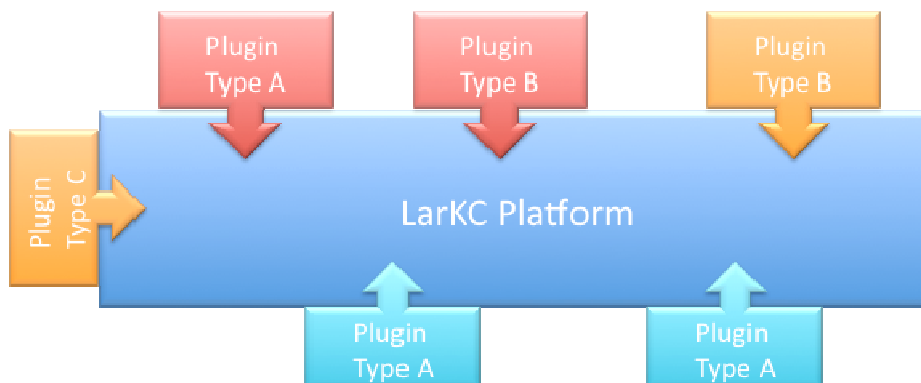


Figure 1 LarKC Platform and plug-in model

In the LarKC execution phase, plug-ins can be orchestrated and executed according to a flexible pipeline, as described in chapter 3.

As shown in Figure 3, in the pipeline, a *Decider* plug-in acts as the “intelligent” component or “meta”-component which enables quality/resource/control decisions at any point in the pipeline. The rest of the plug-ins types (*Identifier*, *Selector*, *Transformer*, *Reasoner*) are generally invoked by a *Decider* (see dotted arrows, representing control-flow) whilst data may flow directly between plug-ins (see solid arrows).

3. Design goals for the Rapid Prototype

The goal of building the Rapid Prototype (a.k.a. “Baby LarKC”) is to function as a first test for the general functionality that the LarKC platform should eventually implement.

The Baby LarKC prototype

- must demonstrate the core functionality of LarKC

- must be built on the same plug-in interfaces that will be used in the first public release of LarKC
- must use both internally and externally developed plug-ins
- must demonstrate reusability of plug-ins under different deciders
- must demonstrate reusability of plug-ins in different pipeline configurations
- must use public datasets
- must be publicly accessible for demonstration purposes
- must be available for both local and remote execution
- is not yet expected to be highly scalable
- is not yet expected to support parallelism or distribution.

4. Design strategy of the Rapid Prototype

The overall design of the LarKC Rapid Prototype has followed a two-fold approach: bottom-up and top-down. These approaches have converged on the current LarKC Rapid Prototype, also known as “Baby LarKC”.

- **Bottom-up approach:** An analysis of the ResearchCyc inference engine (provided by the partner CycEur) was performed to identify and remove those features not essential for the LarKC platform. Simultaneously, modules within the code were identified with particular components of the (emerging) LarKC platform architecture, so that they could be transformed to satisfy LarKC APIs.
- **Top-down approach:** The overall functionality the LarKC platform must provide for the LarKC plug-ins to support their functionalities and their interfaces, was identified in order to define the needed API for each of the plug-in types.

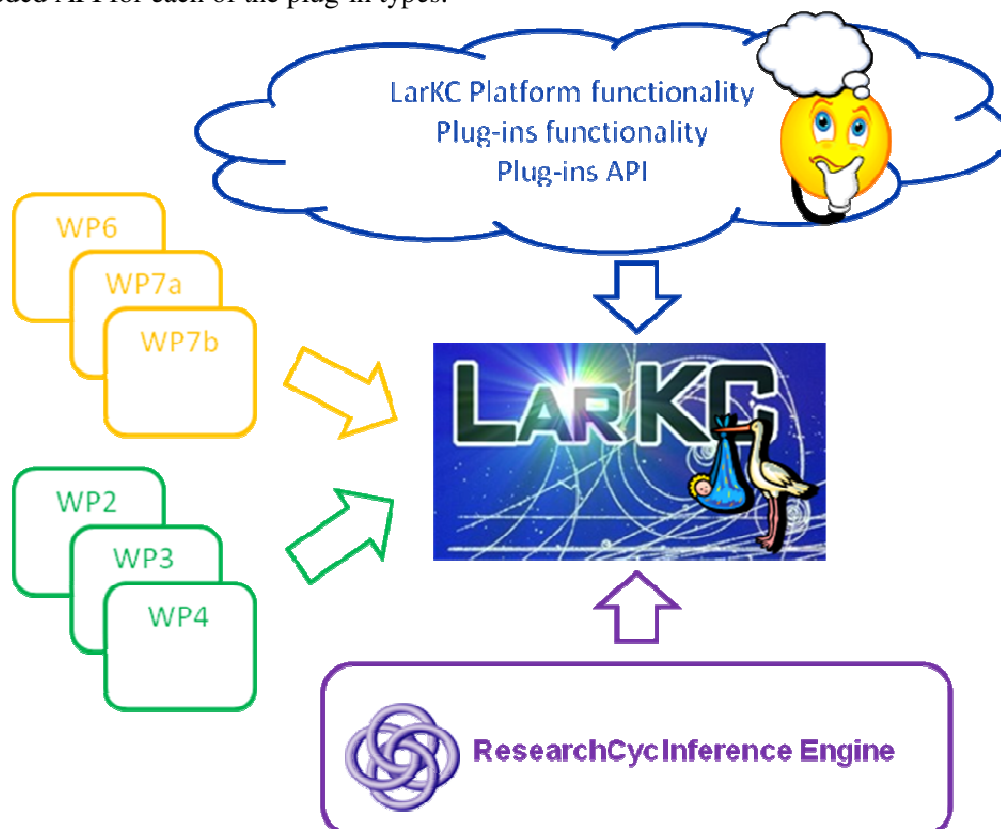


Figure 2 Design Strategy of the LarKC Rapid Prototype

As mentioned in chapter 2 and as described in [1], the pipeline concept has evolved in response to requirements identified during the design phase from a fixed-pipeline concept towards a flexible one (See Figure 3).

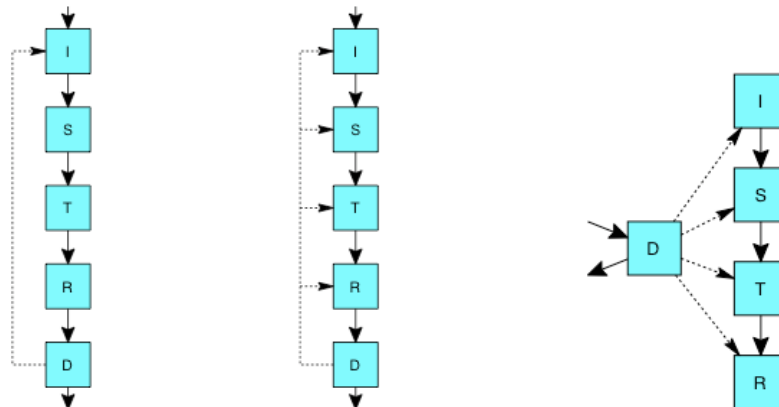


Figure 3 LarKC pipeline design evolution (solid arrows are data-flow, dashed arrows are control flow)

The initial organization of plug-ins, specifying a fixed order for the execution of the different plug-in types, had serious limitations that would have limited the flexibility of the platform and its applicability to both the LarKC use cases and outside experimentation (see left picture). In that organization, the complete pipeline would have had to be executed completely every time, without the possibility of returning to any intermediate step in the pipeline after deciding (box “D” in Figure 3). In order to solve this problem a second version of the pipeline was considered (see pipeline picture in the middle) allowing the control to flow from the *Decider* to any of the other plug-ins in the previous steps of the execution pipeline. This second version still lacked flexibility in the sense that, for many possible LarKC applications, control decisions may need to be made between steps rather than at the end of the pipeline. For example, in some scenarios it may be necessary to decide what to do after the *Selection* plug-in (box “S” in Figure 3) has been executed and before the *Reasoner* plug-in (box “R” in Figure 3) is executed, depending on the amount of applicable knowledge *Selection* identifies. Therefore, the third design version of the pipeline solves this problem by considering the *Decider* plug-in as the “intelligent” component, responsible for reviewing and coordinating the action of the other plug-ins in a reactive planning fashion (where the planning and reaction components may be degenerate if appropriate, also permitting fixed execution orders). This third version (right picture in Figure 3) illustrates the current adopted version of the pipeline.

In parallel with the pipeline concept evolution, the input from the Use cases WPs and the Plug-ins WPs has been also considered, and has been highly valuable for the design of the LarKC platform. To support this purpose, the use cases provided storyboards, and as they were available they were mapped to a pipeline execution scenario. For each of the storyboards’ steps, the necessary plug-ins to support its processing goals have been identified (see [1], chapter 3 Use-case Scenarios).

5. Prototype Implementation: The Baby LarKC

The design outlined above has been implemented in two test-rigs in order to validate the general LarKC ideas. These rigs, whose ideas will be combined in platform releases, are briefly described in sections 5.2 and 5.3.

Conceptually, the two test-rigs only differ in their code for the *Decider* plug-in, and in some minimal wrapping code for each plug-in to register the required information about itself in the meta-knowledge-base. All the other code, which defines the LarKC platform APIs, as well as the code of all of the actual plug-ins,



is exactly the same between the two test-rigs, giving us confidence that plug-ins will indeed be re-usable when controlled by different *Decider* plug-ins.

5.1. API specification

The LarKC API consists of five major plug-in types, as described in [1]. In this section we give a brief overview of these five plug-in types, their behavior, and their interfaces. For brevity the interfaces of the classes are described in Java syntax; however it should be noted that this does not constrain the language in which plug-ins can be written.

In the sections below, different types of objects, which are passed along the pipeline, are mentioned. The section Datamodel provides an overview of these data types and how they are used within the pipeline.

All LarKC plug-ins share a common super class, which is the Plugin class. This class provides that functionality which is common to all plug-in types. The interface of the Plugin class can be seen below:

```
public interface Plugin {  
    public String getIdentifier();  
    public Metadata getMetaData();  
    public QoSInformation getQoSInformation();  
    public void setInputQuery(Query theQuery);  
}
```

From the above interface it can be seen that:

- All plug-ins are identified by a name, which is a string
- Plug-ins provide meta data that describes the functionality that they offer
- Plug-ins provide Quality of Service (QoS) information regarding how they perform the functionality that they offer
- All plug-ins may need access to the initial query (entry query in the LarKC platform) and thus a mutator is provided by specifying this query

Identify

Given a specific query the *Identify* plug-in will identify those Information Sets that could be used to answer the user's query. The interface of the Identify plug-in can be seen below. This plug-in type was originally named Retrieval, in the LarKC project proposal, because it provides the functionality required during the retrieval step of the pipeline. It was renamed Identify to better describe its main function: identification of possible data sources that are relevant for answering a given query.

```
public interface Identifier extends Plugin{  
    public Collection<InformationSet> identify(Query theQuery, Contract theContract);  
}
```

Thus the Identify plug-in is used to narrow the scope of a reasoning task from all available information sets to those information sets that are capable of answering the query. The plug-in takes a query as input and produces a collection of Information Sets as output according to the provided contract. The contract is used to specify the dimensions of the output, for example the number of results returned by a given call to the Identify plug-in. An example of an Identify plug-in could for example identify relevant pieces of data from a semantic data index such as Sindice [2]. Such a plug-in would take Triple Pattern Queries as input and give RDF Graphs as output. These RDF Graphs would match the input Triple Pattern query and thus would contain triples that could be used to answer the users query.



Transform

The Transform plug-in is capable of transforming some piece of data from one representation to another. This plug-in was originally named abstraction plug-in and used during the abstraction phase of the pipeline. However, given that its main functionality is to transform data from one representation to another the name was changed to the Transform plug-in. In the LarKC API we distinguish between the transformation of queries and the transformation of Information Sets. Thus there are two transform plug-in interfaces within the API, namely QueryTransformer and InformationSetTransformer. The interface of each can be seen in below

```
public interface QueryTransformer extends Plugin {  
    public Set<Query> transform(Query theQuery, Contract theContract);  
}
```

```
public interface InformationSetTransformer extends Plugin {  
    public InformationSet transform(InformationSet theInformationSet, Contract theContract);  
}
```

The QueryTransformer plug-in can take a Query as input and produce a set of queries as output according to the provided contract. The contract is used to specify the dimensions of the output, for example the number of queries to generate from the input query. An example of a query transformer would be a SPARQL to Triple Pattern query transformer. Such a plug-in would take a SPARQL query as input and extract triple patterns from the WHERE clause to build a triple pattern query.

The InformationSetTransformer plug-in can take an Information Set as input and provides the transformed information set as output according to the provided contract. The contract defines the dimensions of the output Information Set. An example of such a transformer would be GATE [3] or KIM [4]. Such a plug-in would take a Natural Language Document as input and would extract a number of triples from the document to create an RDF graph, which it would return as output. The contract in this case would describe how many triples the plug-in should extract from the natural language document (and potentially their types).

Select

The Select plug-in is responsible for narrowing the search space even further than Identify has done by taking a selection (or a sample) of the Data Set that has been made available by Identify on which reasoning should be performed. The output of a Select plug-in is a Triple Set, which is essentially a subset of the input Data Set. The interface of the Select Plug-in can be seen below:

```
public interface Selector extends Plugin {  
    public TripleSet select (DataSet theDataSet, Contract theContract);  
}
```

Thus the Select plug-in takes a Data Set as input, identifies a selection from this dataset according to its strategy and returns a Triple Set according to the contract. The contract is used to define the dimensions of the output. An example of a Select plug-in would be one that extracts a particular number of triples from each of the RDF graphs within the Data Set to build the Triple Set. The Contract in this case would define the number of triples to be present in the output Triple Set, or the number of triples to extract from the each of the RDF graphs in the Data Set.

Note that Selection is not necessarily independent of the user query. All plug-ins have a method to accept the user query and this is passed as part of pipeline construction. The query is known beforehand, so there is no need to pass this query to the selector upon every invocation.



Reason

The Reason plug-in will execute a given SPARQL Query against a Triple Set provided by a Select plug-in. The interface of the Reason plug-in can be seen below:

```
public interface Reasoner extends Plugin{  
    public VariableBinding sparqlSelect(SPARQLQuery theQuery, TripleSet theTripleSet, Contract contract);  
    public RdfGraph sparqlConstruct(SPARQLQuery theQuery, TripleSet theTripleSet, Contract contract);  
    public RdfGraph sparqlDescribe(SPARQLQuery theQuery, TripleSet theTripleSet, Contract contract);  
    public BooleanInformationSet sparqlAsk(SPARQLQuery theQuery, TripleSet theTripleSet, Contract contract);  
}
```

The Reason plug-in supports the four standard methods for a SPARQL endpoint, namely select, describe, construct and ask. The input to each of the reason methods are the same and includes the query to be executed, the triple set to reason over and the contract, which defines the behavior of the reasoner. An example of a Reason plug-in would wrap the reasoning component provided by the Jena Framework [5]. In such a plug-in the data in the Triple Set is loaded into the reasoner and the SPARQL query is executed against this model. The output of these reasoning methods depends on the reasoning task being performed. The select method returns a Variable Binding as output where the variables correspond to those specified in the query. The construct and describe methods return RDF graphs, in the first case this graph is constructed according to the query and in the second the graph contains triples that describe the variable specified in the query. Finally ask returns a Boolean Information Set as output, which is true if the pattern in the query can be found in the Triple Set or false if not.

Decide

The Decide plug-in is responsible for building and maintaining the pipeline containing the other plug-in types, and managing the control flow between these plug-ins. The interface of this plug-in can be seen below:

```
public interface Decider extends Plugin{  
    public VariableBinding sparqlSelect(SPARQLQuery theQuery, QoSParameters theQoSParameters);  
    public RdfGraph sparqlConstruct(SPARQLQuery theQuery, QoSParameters theQoSParameters);  
    public RdfGraph sparqlDescribe(SPARQLQuery theQuery, QoSParameters theQoSParameters);  
    public BooleanInformationSet sparqlAsk(SPARQLQuery theQuery, QoSParameters theQoSParameters);  
}
```

The interface of the Decide plug-in is very similar to that of the reason plug-in (as LarKC is a platform for large scale reasoning and as such can be viewed as a reasoner). The major difference is that actual data to reason over is not explicitly specified, as the Identify plug-in is responsible for finding the data within the pipeline.

Datamodel

As can be seen in the previous sections there are a number of different types of objects that are passed along the pipeline. In this section we provide an overview of these data types and an understanding of the relationships between them.

In the LarKC pipeline the Identify plug-in is responsible for finding resources that could be capable of answering the user's query. We refer to these resources as Information Sets and identify two major sub-types that can be returned by identify, namely RDF Graphs and Natural Language Documents. The Transform plug-in is responsible for transforming between different representations and one important type of Transform plug-in can extract RDF Graphs from Natural Language Documents. Thus when we reach the

Select plug-in in the LarKC pipeline we are dealing with RDF Graphs that have been identified as potentially able to answer the user's query and which may have been extracted from Natural Language Documents. This collection of RDF Graphs is termed a Dataset and consists of a Default Graph (an unnamed RDF graph) along with any number of named RDF graphs. The Select plug-in is responsible for identifying a selection (or sample) from within a Dataset that is to be reasoned over to answer the user's query. This selection is termed a Triple Set and consists of triples from any of the RDF Graphs in the Dataset. The relationship between RDF Graphs, Datasets and Triple Sets is shown in Figure 4.

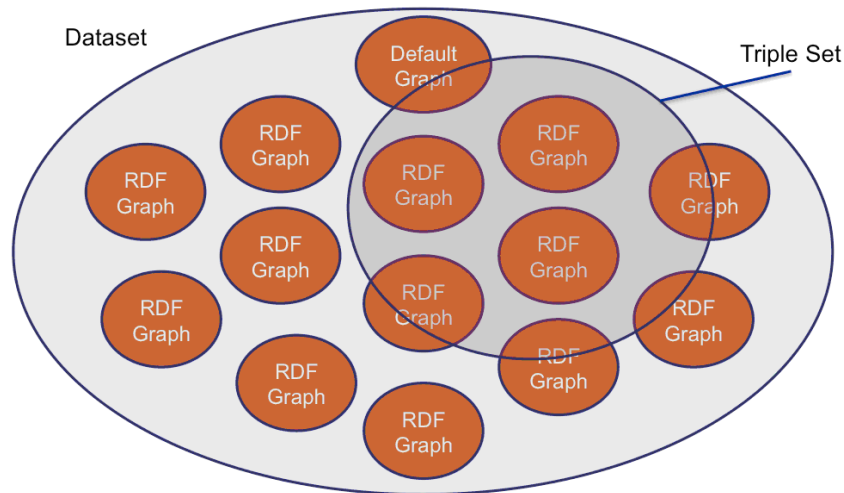


Figure 4 The LarKC data model

Having selected a Triple Set, the Reason plug-in will reason over it in order to fulfill the user's query, which can be a SPARQL Select, Construct, Describe or Ask query. The return types of each of these types of queries are also Information Sets. The SPARQL Select query will return a Variable Binding that provides an assignment of values for each of the variables contained within the user's SPARQL query. The SPARQL Construct and Describe queries both return an RDF Graph as output. For SPARQL Construct this graph is a new graph constructed based on the user's query and for SPARQL Describe this graph is made up of triples that describe the object specified in the SPARQL query. Finally a SPARQL Ask query will return a Boolean Information Set, which will contain true if the pattern described in the user's query is found in the Triple Set being reasoned over or will contain false otherwise.

5.2. Scripted DECIDE Platform

The API described in the previous section has been validated through the implementation of a Test Prototype of the LarKC Platform, with a scripted DECIDE plug-in. This prototype consists of a set of plug-ins implementing a simple LarKC pipeline, as illustrated in Figure 5.

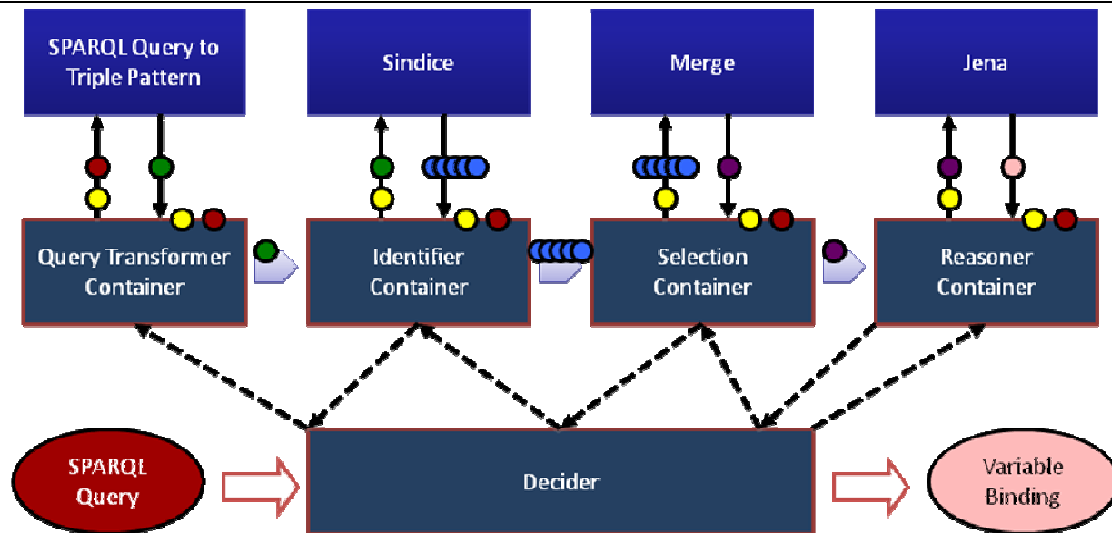


Figure 5 Baby LarKC with Scripted DECIDE

This test-implementation is available at <http://www.larkc.eu/baby-larkc>. This implements a simple LarKC pipeline. The functionality of this pipeline is to act as a SPARQL endpoint to RDF triples available anywhere on the Semantic Web.

The pipeline consists of the following plug-ins:

- **QueryTransformer:** Takes the user-input SPARQL query and transforms it into a number of triple-patterns that summarize the query patterns that make up the users' initial query.
- **Identify:** The triple-pattern queries that have been created are used to invoke Sindice, which returns URI's at which triples conforming to these patterns can be found. The contract parameter of the Identify plug-in determines how many of these URI's should be returned (streaming fashion).
- **Select:** The current Select plug-in only displays two very simple behaviors, either select all or select only the first URL. This is the location in the pipeline where the RDF triples found at the remote URLs are dereferenced to build a TripleSet.
- **Reason:** The Reason plug-in loads the provided TripleSet into a local Jena store and executes the original user SPARQL-query against them.
- **Decide:** These plug-ins are under the control of a simple scripted Decide plug-in that executes these plug-ins in the above fixed order, iterating over the plug-ins until no further information can be identified.

Because of the contract-parameter on the Identify plug-in, this pipeline yields an increasing stream of answers to the original query, as increasingly more URIs are returned by Sindice. This feature is known as "Anytime behavior".

5.3. Self-configuring DECIDE Platform

The second version of the above scenario has been implemented but this time with a more intelligent DECIDE plug-in. Using the Cyc platform (based on the ResearchCyc code), each of the plug-ins (QueryTransformer, Identify, Select, Reason), as well as a number of other plug-ins (e.g. multiple QueryTransformer's) register themselves by adding statements in a meta-knowledge-base, stating facts about their type (Identify, Select, etc.), their I/O signature and, in the future, also their QoS parameters. Using these signatures, the Decide plug-in can compose pipelines which are possible based on the available plug-ins, chooses an appropriate one based on the query, and subsequently executes it. This version implements the same API as described in section 5.1.

The platform registers plug-ins by taking their meta-data descriptions, as returned by `getMetaData()` method, and asserting them into the meta-knowledge-base (MKB). The following information is stored for each plug-in (Figure 6 shows an example of the actual assertions added for one plug-in):

- Unique ID,
- Plug-in Type (e.g. query transformer, selector, reasoner, identifier),
- Input and output types (e.g. SPARQL query, triple set URL, keyword query).

```

isa                : eu-larkc-plugin-identify-Identifier
acceptsParameterType : eu-larkc-core-query-KeywordQuery
pluginConnectsTo   : eu-larkc-plugin-pipelinefilter-JenaPipelineFilter-10
returnsParameterType : java-util-Collection_eu-larkc-core-data-InformationSet
  
```

Figure 6 All assertion for a Sindice Keyword Identifier plug-in.

After the plug-ins are registered decider connects all pairs of plug-ins (A, B) where the output of plug-in A is compatible with the input of plug-in B. Connection is represented by a transitive relation *pluginConnectsTo*, implemented using a Cyc forward rule. This materializes all the instances of this relation in the MKB, which in turn results in a faster performance at run-time. Figure 7 shows all connections inferred for a set of plug-ins. One box corresponds to one plug-in and the links between the boxes correspond to the *pluginConnectsTo* relation.

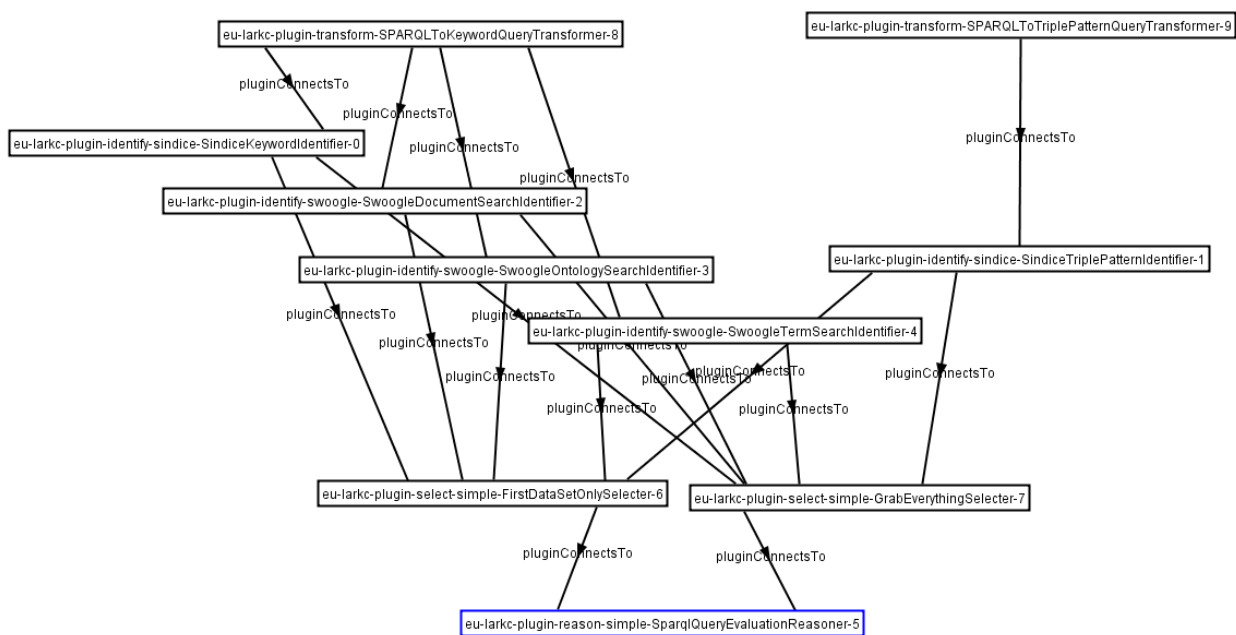


Figure 7 Inferred possible compositions of plug-ins.

When a new query is fired against the platform the decider first checks the type of the query (e.g. SPARQL, CycL) and the output which needs to be returned (e.g. set of variable bindings, RDF graph). The types are used to compose a query for the Cyc reasoning engine, asking for a composition of plug-ins which can take the specified input and produces the required output. The reasoning engine uses the information stored in the MKB about registered plug-ins and their compatibles to answer the query. The result of the query and its proof is used to get possible pipeline compositions which are capable of taking the provided query and producing the expected output.

Figure 8 shows an example of two pipelines which can answer a select SPARQL query and were both automatically composed from the available plug-ins. The left pipeline is based on the Sindice search engine and the right pipeline is based on ResearchCyc. In the Sindice case, a query transformer (T) first transforms the SPARQL query into a keyword query which is then used by an identifier (I) to call the Sindice API and extracts a set of URLs pointing to RDF graphs. A selector (S) loads the RDF graphs and forwards them as a set of triples to the Jena reasoner (R). The reasoner produces a set of variable bindings for using the provided set of triples and the initial SPARQL query. In the ResearchCyc case, the identifier (I) checks the SPARQL query to see if the terms from the query appear in the KB. Then the selector (S) identifies relevant parts of the KB and sends it forward to the reasoner (R). Since the ResearchCyc reasoner can only work with queries written in CycL a query transformer (T) is required to transform the initial SPARQL query CycL. Taking the output from the selector and the transformed query the reasoner produces a set of variable bindings.

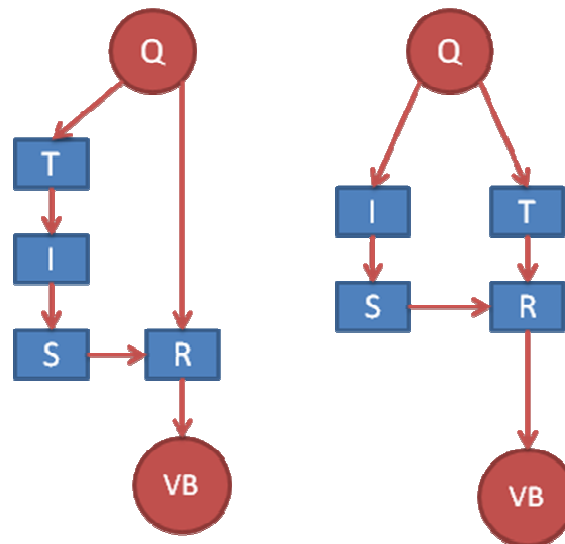


Figure 8 Two different pipelines which can answer same type of queries using different plug-ins.

An advantage of approach taken in the Cyc based decider over the scripted decider from the previous section is that it can be easily extended. For example, plug-in QoS parameters can be easily added as extra assertions into the meta-knowledgebase and the query QoS requirements can be added to the Cyc query which infers plug-in pipeline compositions. Since the system is based on the well-tested Cyc reasoner it can scale to larger meta-knowledgebases with many plug-ins and/or QoS parameters and requirements. It can also do this fast due to special functions for handling the types of queries used and materialization of the important relations in the meta-knowledgebase.

6. Installation guide and user manual

Baby LarKC is available in two forms: as a web-interface, enabling remote execution on a server run by the LarKC consortium, and as downloadable code that can be installed and executed locally. This dual mode reflects the long term plans for the delivery format of the LarKC platform. The current code of the Baby LarKC can be found at <https://svn.gforge.hlr.de/svn/larkc/trunk>, where subdirectory platform_v02 contains LarKC with the scripting decider and platform_v03 directory contains the Cyc inference engine based decider and also full platform support based on research Cyc. The code in both cases is written in pure Java. It can be downloaded using a SVN client like Tortoise SVN (<http://tortoisesvn.tigris.org/>) and then compiled and tested locally.

At the time of writing this deliverable, this repository is LarKC project internal. For requesting access to the gforge LarKC Development Environment, please contact the administrator (Georgina Gallizo, gallizo@hlrs.de).

6.1. Baby LarKC with Scripted Decider

Remote Baby LarKC with the scripted decider is available on the web at <http://www.larkc.eu/baby-larkc> and provides a web interface to the scripted LarKC pipeline. The interface can be seen in Figure 9. The user specifies a query on the provided text area and clicks the execute query button and receives back the results in an anytime fashion as they are available. Example queries for SPARQL Select, Construct, Describe and Ask are available on the four tabs above the text area.



Figure 9 Baby LarKC Web Interface

The prototype can also be downloaded from the LarKC SVN repository (see URL above) and installed in an Apache tomcat installation by unzipping the prototype into the webapps directory of tomcat.

6.2. Baby LarKC with Self-configuring Decider

6.2.1. Installation guide

To compile and run LarKC platform with Self-configuring Decider based on Cyc one must:

1. Check out the code from platform_v03 subdirectory on SVN.
2. Compile the newly downloaded code by running Ant (<http://ant.apache.org/>) over build.xml script, which can be found in the root directory. Ant script will automatically prepare all the directories and files needed to run LarKC and copy them to "release" subdirectory.
3. Run the platform by executing LarKC.bat file.



Here is an example of the steps described above:

```
> svn checkout https://svn.gforge.hlrs.de/svn/larkc/trunk/platform_v03 platform
> cd platform
> ant
> cd release
> LarKC.bat
```

When the platform is up and running, this should be printed out on the console:

```
.
.
2 Initializing HL backing store caches from units/tiny/.

Loading late-breaking OE changes ....

Enabling base TCP services to port 3600.
Initialization time = 5.733 secs.

Start time: Sat Jan 10 17:03:34 CET 2009
Lisp implementation: Cycorp Java SubL Runtime Environment
JVM: Sun Microsystems Inc. Java HotSpot(TM) 64-Bit Server VM 10.0-b22
Current KB: 2
Patch Level: 10.126698
Running on: SHODAN
OS: Windows Vista 6.0 (amd64)
Working directory: D:\temp\platform
Total memory allocated to VM: 490MB.
Memory currently used: 156MB.
Memory currently available: 333MB.

Ready for services.
LarKC(1):
```

6.2.2. User guide

USING THE PLATFORM

Currently the LarKC platform can be accessed over HTTP, using the OpenCyc API. We are not putting much focus on that, because this part will soon be replaced by more specific server. Most likely it will be a SPARQL endpoint obeying the SPARQL protocol.

The port on which the platform is listening is defined in *port-init.lisp* file and is set to 3600 by default.

Example code for how the SPARQL query can be sent to the platform is available at <https://svn.gforge.hlrs.de/svn/larkc/trunk/tests/orchestratorApiTest>. It is a complete Eclipse project with all necessary libraries.



This is the snippet of the code that connects to the platform:

```
CycAccess ca = new CycAccess("localhost", 3600);
HashMap inferenceParams = new HashMap();
CycConstant mtLarkc = ca.makeCycConstant("LarkcMt");

String SELECT_QUERY = "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n" +
    "PREFIX foaf: <http://xmlns.com/foaf/0.1/>\n" +
    "SELECT DISTINCT ?name ?person2 where {\n" +
    "?person rdf:type foaf:Person .\n" +
    "?person foaf:name 'Tim Berners-Lee' .\n" +
    "?person2 rdf:type foaf:Person .\n" +
    "?person2 foaf:knows ?person .\n" +
    "?person2 foaf:name ?name .\n" +
    "}";

String sQuery = "(#$deciderSelect \"" + SELECT_QUERY + "\" ?Z)";
query = ca.makeCycList(sQuery);
answer = ca.askNewCycQuery(query, mtLarkc, inferenceParams);
System.out.println("DECIDER:" + answer.toPrettyString(""));
```

WRITING PLUG-INS

All the plug-ins inside the LarKC platform have to implement one of the plug-in interfaces described in section 5.1, for example *Reasoner*. Some of the basic data types, implementing *Query* or *InformationSet*, are already provided. For example *SparqlQuery* and *TripleSet*.

For now, each of the plug-ins has to be hard coded in the *PluginInit* class in order to be registered in the platform. This is only temporary and will be replaced with external XML plug-in description files. Each of the plug-ins must also provide the required metadata.

Here is a sample implementation of the *getMetaData* method from one of the reasoning plug-ins:

```
// prepare meta-data about this plug-in
MetaDataImpl metaData = new MetaDataImpl(Reasoner.class.getCanonicalName());

// add what it can get on the input and what it gives on the output
metaData.addMethod( new MethodMetaDataImpl("sparqlSelect",
    new String[] { SPARQLQuery.class.getCanonicalName(),
        JenaSPARQLQuery.class.getCanonicalName() },
    new String[] { TripleSet.class.getCanonicalName(),
        VariableBinding.class.getCanonicalName() });

return metaData;
```

7. Conclusions and next steps

The Rapid Prototype described all along this document is already a stable version, validating the design tasks performed during the first 10 months of the project., and fulfilling the goals we set for this initial prototype (section 3). It will be the basis for the Public Release of the first LarKC Prototype, to be released end of project month 14 (May 2009). The essence of the prototype will be maintained, and minor modifications integrated according to the feedback received during the internal testing in the remaining period (M11-M14).



The next WP5 deliverable, D5.3.1 Requirements Analysis and report on lessons learned during prototyping, will include a complete set of requirements towards the definition of the complete LarKC architecture, and will report on the experiences lived and lessons learned during the design and development of the Rapid Prototype described in this deliverable.

8. References

- [1] D1.2.1 Initial operational framework [http://www.larkc.eu/wp-content/uploads/2008/11/larkc_d121 - initial-operational-framework_final.pdf](http://www.larkc.eu/wp-content/uploads/2008/11/larkc_d121_initial-operational-framework_final.pdf)
- [2] <http://www.sindice.com>
- [3] <http://gate.ac.uk/>
- [4] <http://www.ontotext.com/kim>
- [5] <http://jena.sourceforge.net/>