



LarKC

The Large Knowledge Collider

a platform for large scale integrated reasoning and Web-search

FP7 – 215535

D4.3.2 Implementation of Plug-ins for Interleaving Reasoning and Selection

Coordinator: [Zhisheng Huang (VUA)]

With contributions from: [Zhisheng Huang (VUA), Frank van Harmelen (VUA), Stefan Schlobach (VUA), Gaston Tagni(VUA), Annette ten Teije (VUA), Yi Zeng (WICI), Yan Wang (WICI), Ning Zhong (WICI)]

Quality Assessor: [Danica Damljanovic (Sheffield)]

Quality Controller: [Frank van Harmelen (VUA)]

Document Identifier:	LarKC/2010/D4.3.2/V1.0.0
Class Deliverable:	LarKC EU-IST-2008-215535
Version:	version 1.0.0
Date:	March 24, 2010
State:	final
Distribution:	public



EXECUTIVE SUMMARY

In this document, we discuss the implementation issues of plug-ins for interleaving reasoning and selection. The work covers the following topics: i) Implementation of PION for interleaving reasoning and selection within the LarKC platform, ii) Anytime instance retrieval by ontology approximation, and iii) Unifying selection and reasoning with user interests.

For PION, we explore the implementation of variants of PION for interleaving reasoning and selection within the LarKC platform, which includes i) the DIGPION which uses the DIG interface reasoner to call an external PION system, ii) the SimplePION which provides basic implementation of the interleaving of reasoning by an OWLAPI reasoner and selection by syntactic-relevance -based selection functions, iii) the PIONwithStopRule which uses a set of stop rules in the procedure of interleaving reasoning and selection, and iv) the PIONWorkflow which is designed to be an interleaving workflow of reasoning and selection within the LarKC Platform.

In this document, we present an approach of anytime instance retrieval by ontology approximation for interleaving reasoning and language-based selection, and investigate the design and the implementation issue within the LarKC platform. For the interleaving framework with user-interest-based selection, we present several concrete selection strategies, discuss the implementation issue of unifying selection and reasoning with user interests (I-ReaSearch), and report an initial evaluation on scalability for the proposed methods.



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	4.3.2	Title	Implementation of Plug-ins for Interleaving Reasoning and Selection
Work Package	Number	4	Title	Reasoning and Deciding

Date of Delivery	Contractual	M24	Actual	31-March-10
Status	version 1.0.0		final <input type="checkbox"/>	
Nature	prototype <input checked="" type="checkbox"/> report <input type="checkbox"/> dissemination <input type="checkbox"/>			
Dissemination Level	public <input checked="" type="checkbox"/> consortium <input type="checkbox"/>			

Authors (Partner)	Zhisheng Huang (VUA), Frank van Harmelen (VUA), Stefan Schlobach (VUA), Gaston Tagni(VUA), Annette ten Teije (VUA), Yi Zeng (WICI), Yan Wang (WICI), Ning Zhong (WICI)		
Resp. Author	Zhisheng Huang (VUA)	E-mail	huang@cs.vu.nl
	Partner VUA	Phone	+31 (20) 5987823

Abstract (for dissemination)	In this document, we discuss the implementation issues of plug-ins for interleaving reasoning and selection. The work covers the following topics: i) Implementation of PION for interleaving reasoning and selection within the LarKC platform, ii) Anytime instance retrieval by ontology approximation, and iii) Unifying selection and reasoning with user interests.
Keywords	Reasoning, Selection, Approximate Reasoning, Stop Rules, User Interests



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 Email: 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 RE- SPONSABILITA LIMITATA Milano, Italy Email: emanuele.dellavalle@cefriel.it
CYCORP, RAZISKOVANJE IN EKSPERI- MENTALNI RAZVOJ D.O.O.		Michael Witbrock CYCORP, RAZISKOVANJE IN EKSPERI- MENTALNI 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 AD		Atanas Kiryakov, Ontotext Lab, Sofia, Bulgaria Email: naso@ontotext.com
SALTLUX INC.		Kono Kim SALTLUX INC Seoul, Korea Email: kono@saltlux.com
SIEMENS AKTIENGESELLSCHAFT		Dr. Volker Tresp SIEMENS AKTIENGESELLSCHAFT Muenchen, Germany Email: 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		Prof. Dr. Frank van Harmelen, VRIJE UNIVERSITEIT AMSTERDAM Amsterdam, Netherlands Email: Frank.van.Harmelen@cs.vu.nl
THE INTERNATIONAL WIC INSTI- TUTE, BEIJING UNIVERSITY OF TECHNOLOGY		Prof. Dr. Ning Zhong, THE INTERNATIONAL WIC INSTITUTE Mabeshi, Japan Email: zhong@maebashi-it.ac.jp
INTERNATIONAL AGENCY FOR RE- SEARCH ON CANCER		Dr. Paul Brennan, INTERNATIONAL AGENCY FOR RE- SEARCH ON CANCER Lyon, France Email: brennan@iarc.fr
INFORMATION RETRIEVAL FACILITY		Dr. John Tait, Dr. Paul Brennan, INFORMATION RETRIEVAL FACILITY Vienna, Austria Email: john.tait@ir-facility.org



TABLE OF CONTENTS

LIST OF FIGURES	6
LIST OF ACRONYMS	7
1 INTRODUCTION	8
2 PION FOR INTERLEAVING REASONING AND SELECTION	10
2.1 General Framework of PION	10
2.2 PION within the LarKC Platform	12
2.3 Implementation of DIGPION	12
2.4 Implementation of SimplePION	15
2.5 Implementation of PIONwithStopRules	18
2.6 Implementation of PIONWorkflow	20
2.7 Conclusion	21
3 ANYTIME REASONING BY ONTOLOGY APPROXIMATION	22
3.1 A Framework for Anytime Reasoning by Ontology Approximation . . .	22
3.2 Approximate Reasoning Plug-in	24
3.3 Two LarKC Workflows for Anytime Approximate Reasoning	28
4 I-REASearch: UNIFYING SELECTION AND REASONING WITH USER INTERESTS	31
4.1 A General Framework of I-ReaSearch	31
4.2 Interests-Based Query Refinement	32
4.3 Interleaving Selection and Reasoning Based on User Interests	33
4.4 Implementation of Interests-Based Reasoner Plug-in	33
4.5 An Initial Evaluation on the Scalability of Proposed Strategies	36
5 CONCLUSION	38
REFERENCES	38
A PION WITHIN THE LARKC PLATFORM: USER MANUAL	41
A.1 Using SPARQL-DL to Express OWL-DL Formulas	41
A.2 DIGPION	42
A.3 SimplePION	44
A.4 PIONwithStopRules	44
A.5 PIONWorkflow	45



LIST OF FIGURES

2.1	DIGPION	13
2.2	SimplePION	15
2.3	SimplePION Class Diagram	16
2.4	SPARQL Ask Processing in SimplePION	17
2.5	PION with Stop Rules	18
2.6	The Class Diagram of PIONwithStopRules	19
2.7	PIONWorkflow	21
3.1	3-step workflow for approximate reasoning experiments; every square box can be changed per experiment	23
3.2	gain, pain and gain-pain curves	25
3.3	Architecture of a reason plug-in in LarKC implementing the 3-step workflow introduced above	25
3.4	Class diagram depicting the main classes that make up the approximate reasoner plug-in	28
3.5	LarKC workflow for anytime reasoning by ontology approximation using a Selector to approximate ontologies	29
3.6	LarKC workflow for anytime reasoning by ontology approximation using a Transformer to approximate ontologies	30
4.1	the Interest-Based Reasoner	34
4.2	the Interest-Based Reasoner Class Diagram	35
4.3	SPARQL SELECT Processing in the Interest-Based Reasoner	35
4.4	Query Refinement Processing	36
4.5	A comparative study on the Scalability of Proposed Strategies	37
A.1	PION TestBed	44



LIST OF ACRONYMS

Acronym	Description
DL	Description Logics
OWL	Web Ontology Language
PION	The System of Processing Inconsistent Ontologies
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Protocol And RDF Query Language



1. Introduction

Interleaving reasoning and selection is considered to be an approach to improving the performance of the LarKC platform. The main idea of the interleaving framework is to use selectors to select only limited and relevant part of data for reasoning, so that the efficiency and the scalability of reasoning can be improved. Thus, the general scenario of interleaving reasoning and selection consists of the following three steps:

- **Selection:** Use a selector to select part of data.
- **Reasoning:** Use a reasoner to reason over the selected data;
- **Deciding:** Use a decider to decide whether or not the procedure should be stopped and return an answer or go back to the selection step to continue the interleaving processing.

There are various selection strategies in the interleaving framework. The selection can be made based on some measures of relevance among the data. For example, a syntactic relevance measure is one which is based on a symbolic co-occurrence checking [11]. A semantic relevance measure is one which uses some kind of background knowledge which provides some information about the meaning of the data [12]. The selection can be also made based on user preference or interests. In Deliverable D4.3.1 [14], we presented a framework of interleaving reasoning and selection, proposed several approaches for the interleaving framework, and examined the framework of interleaving reasoning and selection within the LarKC platform. In Deliverable D4.3.1, we reported the following work of interleaving reasoning and selection:

- **Query-based selection.** Query-based selection is one in which data/axioms selection is made by examining its relevant with its reasoning queries. PION is a system of interleaving reasoning and selection which uses some selection functions based on either a syntactic relevance measure or a semantic relevance measure with respect to its reasoning queries. In D4.3.1, we proposed various strategies of interleaving query-based selection and reasoning within the LarKC platform;
- **User-interest-based selection.** User-interest-based selection is one in which selection is made based on the interests of perspective users. This kind of selection is examined with respect to variant granularities of user interests. In Deliverable D4.3.1, we investigated the Web scale reasoning from the perspective of granular reasoning, and developed several strategies of Web scale reasoning with user interests and granularities.
- **Language-based selection.** Language-based selection is one in which selection is made by selecting part of sub-language or vocabularies. In Deliverable 4.3.1, we developed several approaches of classification with anytime behaviors based on approximate reasoning and reported the results of the experiments with several realistic ontologies.

This document is a sequel of Deliverable D4.3.1. In this document we focus on the implementation issues of the work proposed in Deliverable D4.3.1. The work reported in this document includes:



- We present several variants of PION for the framework of interleaving reasoning and selection within the LarKC platform. Those variants of PION include: i) the DIGPION, a PION which uses an external PION sever via the DIG interface reasoner plugin, ii) the SimplePION, a PION which uses an internal OWLAPI reasoner plug-in for interleaving reasoning and selection, iii) the PIONwithStopRules, a PION which uses stop rules to decide the interleaving processing, and iv) the PIONWorkflow, a PION which is designed to be a workflow which uses a reasoner plug-in for reasoning, uses several selection plug-ins for selection, and a decider plug-in for the interleaving processing.
- Anytime Reasoning by Ontology Approximation. We introduce a framework for studying anytime reasoning by ontology approximation systematically through the specification of three independent components. Such framework is specified by a 3-step workflow consisting of a number of independent modules, which can be instantiated for different reasoning tasks and approximation strategies. The crucial elements of this workflow are separate modules for *approximation*, *reasoning* and *evaluation*. After the introduction of this framework we present three concrete ideas on how this 3-step workflow can be realized in the LarKC platform and how selection and reasoning can be interleaved to achieve anytime behaviour. This includes the description of a single Reason plug-in that implements parts of the aforementioned framework and the description of two LarKC workflows for anytime reasoning by ontology approximation.
- For the interleaving framework with user-interest-based selection, we present several concrete selection strategies, discuss the implementation issue of unifying selection and reasoning with user interests (I-ReaSearch), and report an initial evaluation on scalability for the proposed methods.

This document also provides a user guide in the appendix, which can be considered as a reference manual for the plug-ins which are reported in this deliverable and have been released in the LarKC platform.

The rest of this document is organized as follows: Chapter 2 reports the work on variants of PION. Chapter 3 investigates the work of anytime reasoning by ontology approximation, Chapter 4 presents the work of the interleaving framework with user-interest-based selection. Chapter 5 concludes this document. Appendix A provides the user guide for the existing released plug-ins of the interleaving framework within the LarKC platform.



2. PION for Interleaving Reasoning and Selection

2.1 General Framework of PION

PION is a system of interleaving reasoning and query-based selection for reasoning with inconsistent ontologies. In PION, selection functions play the main role for query-based selection. The selection function can either be based on a syntactic approach, like Chopra, Parikh, and Wassermann's syntactic relevance [8] and those in PION[11], or based on semantic relevance like for example in computational linguistics as in Wordnet [7] or based on semantic relevance which is measure by the co-occurrence of concepts in search engines like Google[12].

In our framework, selection functions are designed to be query-specific, which is different from the traditional approach in belief revision and non-monotonic reasoning, which assumes that there exists a general preference ordering on formulas for selection. Given a knowledge base Σ and a query ϕ , a selection function s is one which returns a subset of Σ at the step $k > 0$. Let \mathbf{L} be the ontology language, which is denoted as a formula set. A selection function s is a mapping $s : \mathcal{P}(\mathbf{L}) \times \mathbf{L} \times N \rightarrow \mathcal{P}(\mathbf{L})$ such that $s(\Sigma, \phi, k) \subseteq \Sigma$. The query-specific approach can be understood to be the one with various rankings per query.

A selection function s is called *monotonic* if the subsets it selects monotonically increase or decrease, i.e., $s(\Sigma, \phi, k) \subseteq s(\Sigma, \phi, k + 1)$, or vice versa. For monotonically increasing selection functions, the initial set is either an emptyset, i.e., $s(\Sigma, \phi, 0) = \emptyset$, or a fixed set Σ_0 . For monotonically decreasing selection functions, usually the initial set $s(\Sigma, \phi, 0) = \Sigma$. The decreasing selection functions will reduce some formulas from the inconsistent set step by step until they find a maximally consistent set.

Traditional reasoning methods cannot be used to handle knowledge bases with large scale¹. Hence, selecting and reasoning on subsets of Σ may be appropriate as an approximation approach with monotonically increasing selection functions. Reasoning on a large scale knowledge base Σ can use different selection strategies to achieve this goal. Generally, they all follow an iterative procedure which consists of the following processing loop, based on the selection-reasoning-decision loop discussed above:

- i) select part of the knowledge base, i.e., find a subset Σ'_i of Σ where i is a positive integer, i.e., $i \in I^+$;
- ii) apply the standard reasoning to check if $\Sigma'_i \models \phi$;
- iii) decide whether or not to stop the reasoning procedure or continue the reasoning with gradually increased selected subgraph of the knowledge graph (Hence, $\Sigma'_1 \subseteq \Sigma'_2 \subseteq \dots \subseteq \Sigma$).

Monotonically increasing selection functions have the advantage that they do not have to return *all* subsets for consideration at the same time. If a query can be answered after considering some consistent subset of the knowledge graph KG for some value of k , then other subsets (for higher values of k) don't have to be considered any more, because they will not change the answer of the reasoner. In the following, we use $\Sigma \models \phi$ to denote that ϕ is a consequence of Σ in the standard reasoning², and use $\Sigma \approx \phi$ to denote that ϕ is a consequence of Σ in the nonstandard reasoning.

¹The meaning of *large scale* is relative and might mean different things. In this document, we use the term "large scale" to mean that the size of data is too large to be processed by a nowadays ordinary PC.

²Namely, for any model M of Σ , $M \models \phi$.



[8] proposes a syntactic relevance to measure the relationship between two formulas in belief sets, so that the relevance can be used to guide the belief revision based on Schaerf and Cadoli's method of approximate reasoning[17]. Given a formula set Σ , two atoms p, q are directly relevant, denoted by $R(p, q, \Sigma)$ iff there is a formula $\alpha \in \Sigma$ such that p, q appear in α . A pair of atoms p and q are k -relevant with respect to Σ iff there exist $p_1, p_2, \dots, p_k \in \mathcal{L}$ such that: (a) p, p_1 are directly relevant; (b) p_i, p_{i+1} are directly relevant, $i = 1, \dots, k - 1$; and (c) p_k, q are directly relevant (i.e., directly relevant is k -relevant for $k = 0$).

The notions of relevance above are based on propositional logics. However, ontology languages are usually written in some fragment of the first order logic. We extend the ideas of relevance to ontology language. The direct relevance between two formulas are defined as a binary relation on formulas, namely $\mathcal{R} \subseteq \mathbf{L} \times \mathbf{L}$. Given a direct relevance relation \mathcal{R} , we can extend it to a relation \mathcal{R}^+ on a formula and a formula set, i.e., $\mathcal{R}^+ \subseteq \mathbf{L} \times \mathcal{P}(\mathbf{L})$ as follows:

$$\langle \phi, \Sigma \rangle \in \mathcal{R}^+ \text{ iff } \exists \psi \in \Sigma \text{ such that } \langle \phi, \psi \rangle \in \mathcal{R}.$$

Namely, a formula ϕ is relevant to a knowledge base Σ iff there exists a formula $\phi' \in \Sigma$ such that ϕ and ϕ' are directly relevant. We can similarly specialize the notion of k -relevance. Two formulas ϕ, ϕ' are k -relevant with respect to a formula Σ iff there exist formulas $\phi_0, \dots, \phi_k \in \Sigma$ such that ϕ and ϕ_0 , ϕ_0 and ϕ_1 , \dots , and ϕ_k and ϕ' are directly relevant. A formula ϕ is k -relevant to a set Σ iff there exists a formula $\phi' \in \Sigma$ such that ϕ and ϕ' are k -relevant with respect to Σ .

We can use a relevance relation to define a selection function s to extend the query ' $\Sigma \approx \phi?$ ' as follows: We start with the query formula ϕ as a starting point for the selection based on syntactic relevance. Namely, we define:

$$s(\Sigma, \phi, 0) = \emptyset.$$

Then the selection function selects the formulas $\psi \in \Sigma$ which are directly relevant to ϕ as a working set (i.e. $k = 1$) to see whether or not they are sufficient to give an answer to the query. Namely, we define:

$$s(\Sigma, \phi, 1) = \{\psi \in \Sigma \mid \phi \text{ and } \psi \text{ are directly relevant}\}.$$

If the reasoning process can obtain an answer to the query, it stops. Otherwise the selection function increases the relevance degree by 1, thereby adding more formulas that are relevant to the current working set. Namely, we have:

$$s(\Sigma, \phi, k) = \{\psi \in \Sigma \mid \psi \text{ is directly relevant to } s(\Sigma, \phi, k - 1)\},$$

for $k > 1$. This leads to a "fan out" behavior of the selection function: the first selection is the set of all formulae that are directly relevant to the query; then all formulae are selected that are directly relevant to that set, etc. This intuition is formalized in this: The relevance-based selection function s is monotonically increasing. We observe that If $k \geq 1$, then

$$s(\Sigma, \phi, k) = \{\phi \mid \phi \text{ is } (k-1)\text{-relevant to } \Sigma\}$$



2.2 PION within the LarKC Platform

We have developed the following variants of PION within the LarKC platform:

- **DIGPION.** DIGPION is the one in which an external PION reasoner is called via the DIG interface plug-in within the LarKC platform. The main advantage of DIGPION is that we can rely on an externally implemented PION system for interleaving reasoning and selection within the LarKC Platform.
- **SimplePION.** SimplePION is the one in which PION is implemented as a plug-in with some simplified functions, which include the support of standard boolean answers (i.e., either "true" or "false") without using the three-valued answers such as "accepted", "rejected", and "undertermined", which have been proposed in [11].
- **PIONwithStopRules.** PIONwithStopRules is the one in which PION uses some stop rules to decide when it would stop the selection and jump to provide its reasoning result. The idea of using stop rules is inspired by the investigation of human and animal search strategies in ecology and cognitive science. Using stop rules for LarKC has been investigated in LarKC deliverable D4.2.2 [16].
- **PIONWorkflow.** PIONWorkflow is the one in which PION is designed to be a workflow which uses selection plug-ins and reasoner plug-ins. The main advantage of the scenario of PIONWorkflow is that this approach provides the possibility to use various selectors and reasoners which have been implemented independently from the interleaving framework.

2.3 Implementation of DIGPION

DIGPION is the one in which an external PION reasoner is called via the DIG interface[5]³ plug-in with the LarKC platform, which is shown in Figure 2.1. Namely, DIGPION uses an external PION reasoner which supports the DIG interface. An external PION reasoner which provides the DIG interface can be downloaded from the PION website at Vrije University Amsterdam⁴. The external PION is a reasoner/system which can return meaningful answers to queries on inconsistent ontologies. PION is powered by XDIG[13]⁵, an extended DIG Description Logic Interface for Prolog, in particular, for SWI-Prolog. The external PION supports TELL requests both in DIG and in OWL, and ASK requests in DIG. The external PION uses a standard DL reasoners such as RacerPro⁶, FaCT++⁷, KAON2⁸ for its standard reasoning over DL-based data. In the scenario of DIGPION, calling the PION reasoner is actually achieved by calling the DIGReasoner plug-in, which provides the support for the DIG interface within the LarKC platform.

³<http://dl.kr.org/dig/>

⁴<http://wasp.cs.vu.nl/sekt/pion/>

⁵<http://wasp.cs.vu.nl/sekt/dig/>

⁶<http://www.racer-systems.com/>

⁷<http://owl.cs.manchester.ac.uk/fact++/>

⁸<http://kaon2.semanticweb.org/>

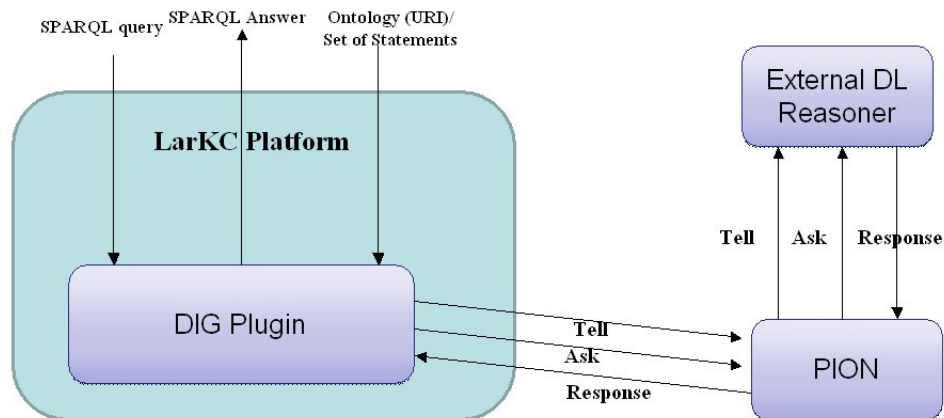


Figure 2.1: DIGPION

Thus, for DIGPION, the main issue of its implementation is how to implement the DIG Reasoner plug-in. In the following, we will briefly report the implementation of the DIG Reasoner plug-in.

Implementation of the DIG Reasoner Plug-in

The Motivation for the DIG Reasoner plug-in within the LarKC platform is as follows. All popular DL-reasoners (such as RacerPro, FACT++, Pellet, KAON2) provide the DIG interface support. The LarKC platform needs the DL/OWL reasoning support. It is convenient for developers to gain the DL reasoning support via the LarKC DIG interface. The DIG reasoner plug-in would provide an easy approach to wrapping non-java-based reasoners (such as PION, MORE, DION, etc.) via the DIG interface.

The DIG interface is a Description Logic Interface developed by DIG (Description Logic Implementation Group)[5]. The DIG interface uses an HTTP-based protocol, which is similar with SOAP. The DIG interface 1.0 was developed in 2002[2]. The DIG interface 1.1, an improved version of the DIG interface, was developed in 2003[3]. the DIG interface 2.0 was designed for the support of reasoning with OWL data, which was developed in 2006 [4]. The OWLLink was designed to be the new generation of DIG for OWL2 in 2008[15].

The current version of the DIG reasoner plug-in supports the following functionalities:

- The DIG interface 1.1⁹.
- SPARQL Ask and SPARQL Select queries
- DL Expressions (conjunction, disjunction, disjoint, negation).
- DIG queries (subsumption, instance, instances).
- Reasoning with OWL Data (by using the OWL2DIG library to translate OWL data into DIG data).

⁹We are now developing the OWLLink Reasoner plug-in within the LarKC Platform, which will be released shortly.



The main tasks of the DIG Reasoner plug-in are:

- **Data Translation.** Because the data set imported to a reasoner plug-in in the LarKC platform is designed to be a set of statements. The first step of the DIG reasoner plug-in is to translate a set of statements (ontology data) into a DIG data, so that it can be posted to the external DIG reasoner. If it is an OWL-DL data, the system will use the OWL2DIG library to translate it into a DIG data¹⁰.
- **Query Translation.** Since the query to a reasoner plug-in in the LarKC platform is designed to be a SPARQL query, that query should be translated into DIG queries, so that they can be posted into the external DIG reasoner.
- **Query and answer processing.** The DIG reasoner plug-in may have to make several DIG queries to get the complete answer to a given SPARQL query. For example, we cannot express a single DIG query which involves two variables such as *SubClassof(x, y)*. However, SPARQL is expressive to provide a query which involves multiple variables. The reasoning result of a SPARQL query can be obtained by multiple DIG query steps, in which one step is used to obtain variable binding of a single variable, then another step is used to obtain variable binding of another variable by instantiating a variable of the corresponding the DIG query.
- **Translate DIG answers into SPARQL answers.** Since the output of a reasoner plug-in is designed to be a SPARQL answer (say, a variable binding for a SPARQL select query), the system have to translate the DIG answers into their SPARQL answers.

SPARQL is too expressive for a DL reasoner can support. In SPARQL, there is no semantic interpretation for DL expressions such as *owl : sameas*, *owl : disjointwith*, etc. In [19], Sirin and Parsia propose the SPARQL-DL to solve that problem. SPARQL-DL is a DL-specific SPARQL with some DL primitives, such as *type(a, C)*, *SubClassof(C1, C2)*, *DisjointWith(C1, C2)*, *ComplementOf(C1, C2)*, and *EquivalentClass(C1, C2)*. In order to translate DL expressions into RDF triples, the SPARQL-DL uses the OWL-DL method which is recommended in [1]¹¹.

The following is an example of SPARQL-DL query, which uses *owl : interSectionOf* to express the intersection concept and uses *rdf : first* and *rdf : rest* to express a concept list in SPARQL.

```
PREFIX rdfs:      <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX owl:    <http://www.w3.org/2002/07/owl#>
ASK
  wine:Bordeaux rdfs:subClassOf _:x.
  _:x owl:interSectionOf _:y1.
  _:y1 rdf:first wine:SweetWine.
  _:y1 rdf:rest wine:TableWine.
  wine:Bordeaux rdf:type owl:Class.}
```

¹⁰Of course, this task can be done by a data transformer in terms of the LarKC platform. We leave it as one of the future work for PION as a workflow.

¹¹<http://www.w3.org/TR/owl-semantic/mapping.html>

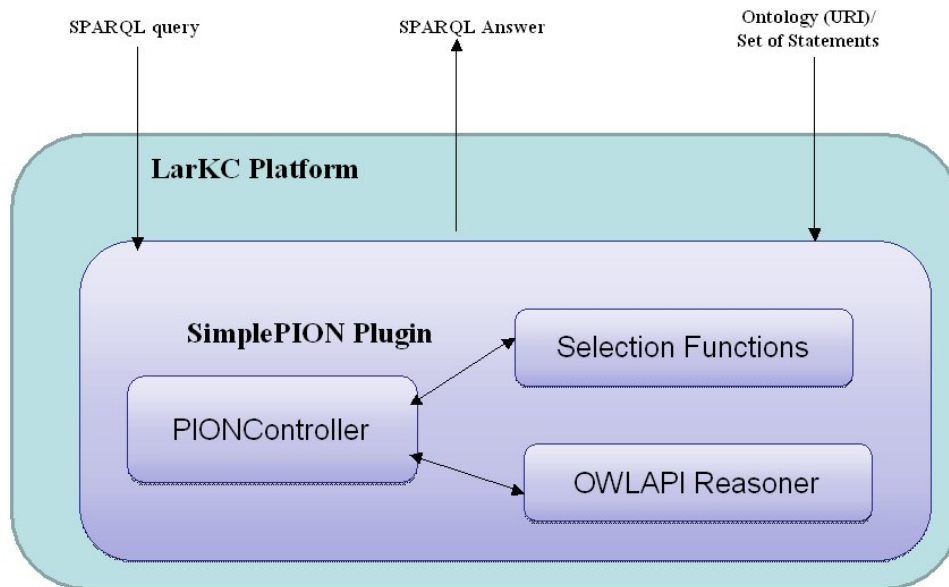


Figure 2.2: SimplePION

2.4 Implementation of SimplePION

SimplePION is the one in which PION is implemented as a plug-in with some simplified functions of PION, which is shown in Figure 2.2. SimplePION uses selection functions to select a consistent sub-ontology¹² of the OWL data (i.e., a set of OWL statements) and uses a built-in OWLAPI reasoner for the standard reasoning over the selected sub-ontology.

The current version of the SimplePION has supported the following functionalities:

- The Boolean answers (i.e., either true or false).
- SPARQL Ask and SPARQL Select queries.
- Syntactic-relevance-based selection functions.
- Reasoning with OWL Data.

The class diagram of the SimplePION describes various entities of the system as classes and the relation between these, which is shown in Figure 2.3. *SimplePIONReasoner* calls the method *GetOWLReasonerFactfromStatement* to obtain the standard OWLAPI reasoner which is based on a Pellet Reasoner. The standard OWLAPI reasoner is specified by the class *OWLReasonerFact* which contains the reasoner’s relevant information such as *OWLDataFactory*, *OWLOntologyManager*, and its *OWLOntology* which corresponds with the set of *Statements*, an input of the SimplePION reasoner. Based on that OWLAPI reasoner (i.e., an *OWLReasonerFact*), *SimplePIONReasoner* uses the method *getAnswersfromReasonerFactonPattern* to get a set of *OWLAnswer* from the query which is specified as a *BasicPattern* of the *SPARQLQuery*, an input of the SimplePION reasoner. *SimplePIONReasoner* uses the method

¹²Namely, a subset of the OWL statements. In this document, we use the term “sub-ontology” and the term “subset of the OWL data” interchangeably.

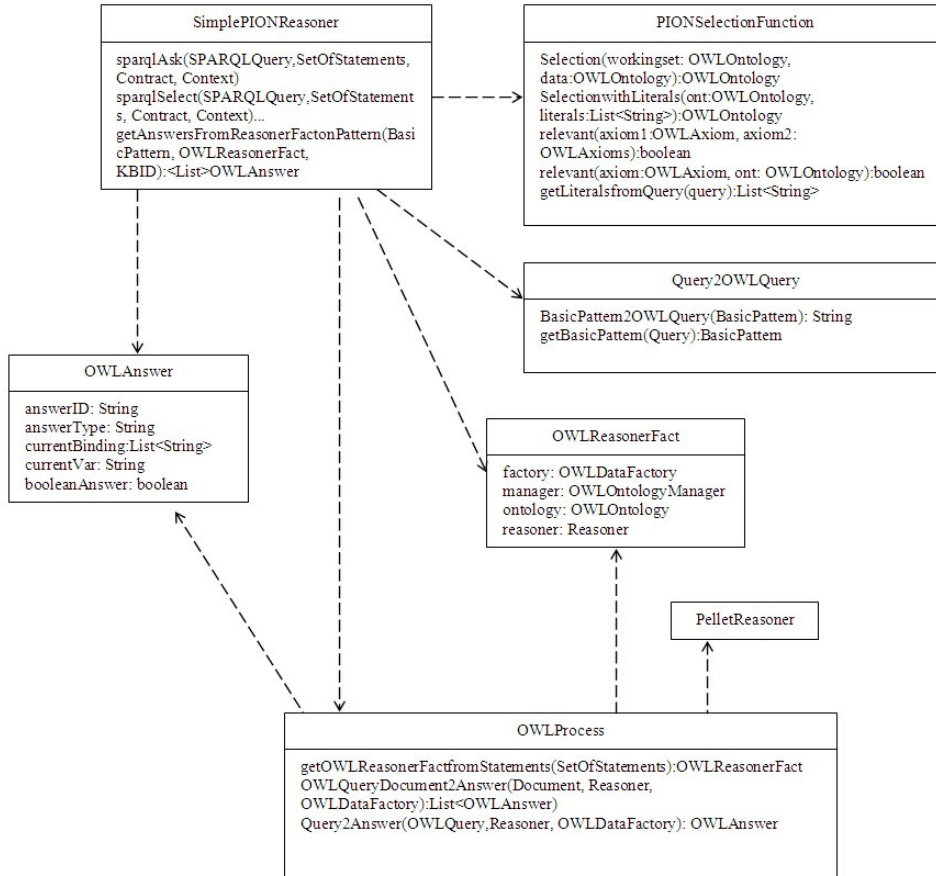


Figure 2.3: SimplePION Class Diagram

BasicPattern2OWLQuery in the class *Query2OWLQuery* to translate the *SPARQLQuery* into a string which corresponds with the internal representation of the query which can be handled further by using the OWLAPI reasoner.

The SPARQL Ask and the SPARQL Select queries are the two main functionalities which have been supported by the SimplePION. Figure 2.4 shows that how SimplePION deals with the SPARQL Ask query. The initial processing of the SPARQL Ask scenario is to i) get an OWLAPI reasoner, ii) get the OWLOntology from the set of the input statements, iii) get the basic pattern of the SPARQLQuery, and iv) get the OWLQuery from the basic pattern. The processing checks further if the OWLOntology is consistent. If the OWLOntology is already consistent, we use the standard SPARQL Ask method in the OWLAPI reasoner to obtain the answer from the OWLOntology. If the OWLOntology is not consistent, we select a sub-ontology of the OWLOntology, based on the literals in the query, and check if the selected sub-ontology is consistent. If the selected sub-ontology is inconsistent, that means that we cannot find a consistent sub-ontology. Thus, return the answer "false". If the selected sub-ontology is consistent, we use the OWLAPI reasoner to reason with the selected ontology to get the answer. If the answer is "true", that the processing returns the answer. If the answer is "false", the processing would continue the selection procedure. Namely, a new sub-ontology which is relevant to the existing selected ontology (thus, the size of the newly selected sub-ontology may become larger) is selected and its consistency is checked.

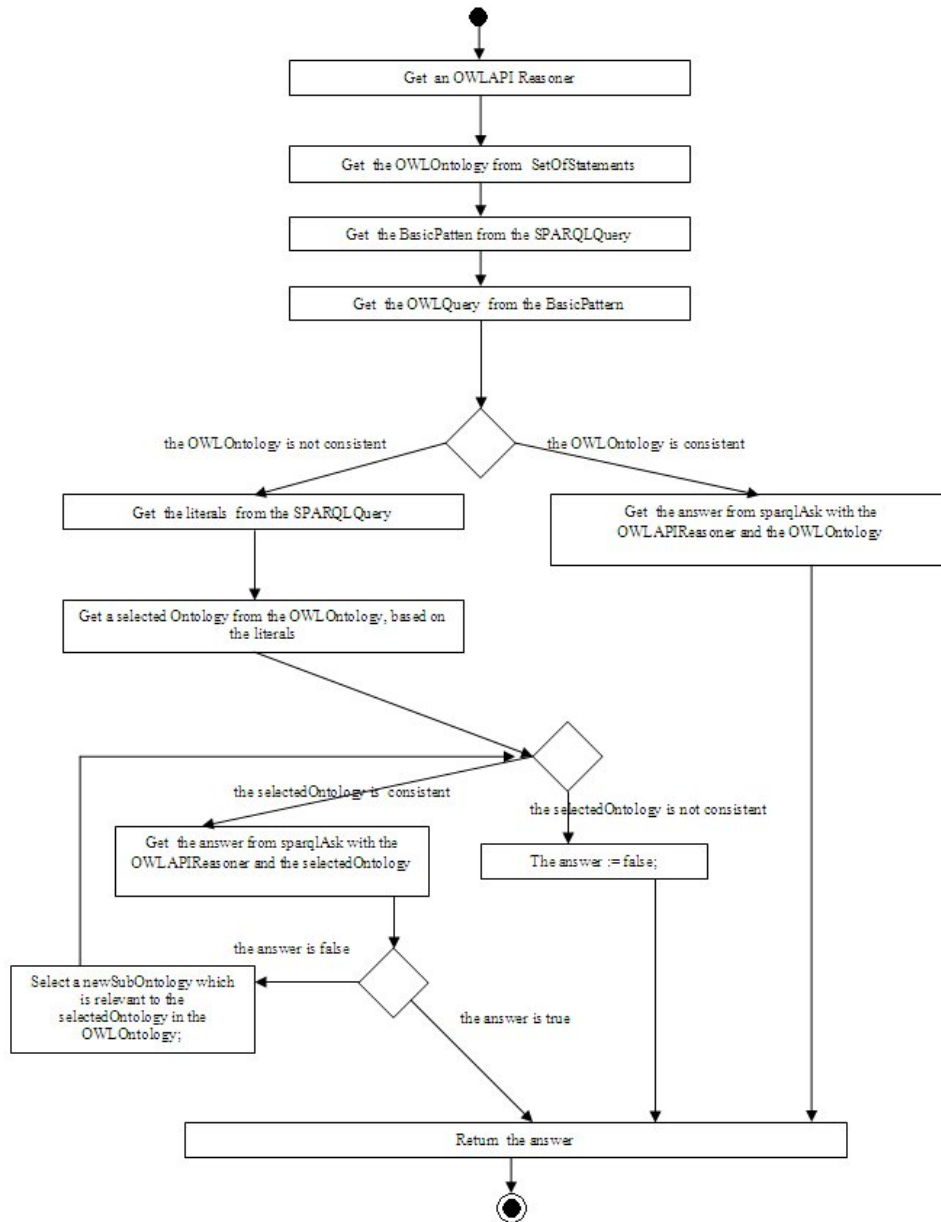


Figure 2.4: SPARQL Ask Processing in SimplePION

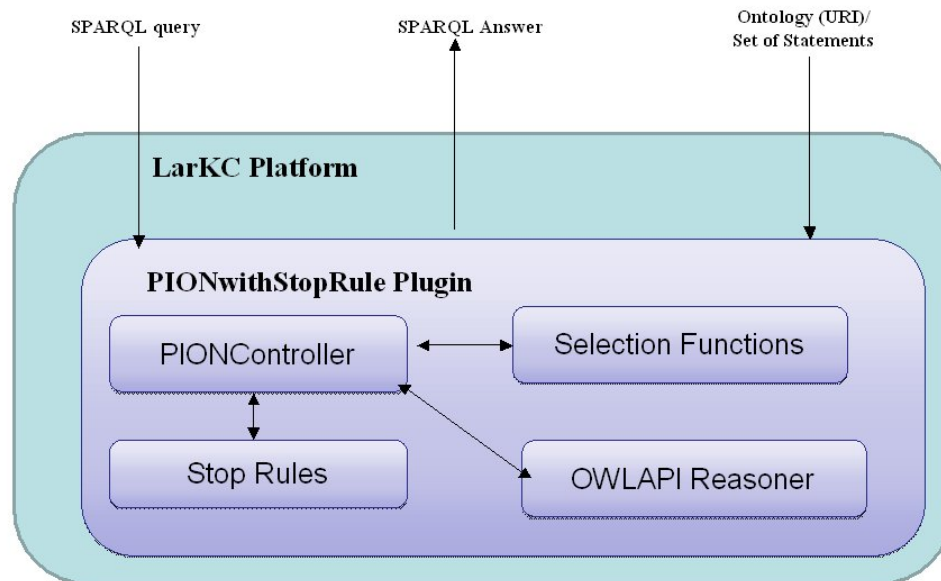


Figure 2.5: PION with Stop Rules

2.5 Implementation of PIONwithStopRules

SimplePION may need a lot of time to complete its processing, because it would exhaust the relevance checking and selection over the whole data set. A heuristic method to improve the efficiency of reasoning with some loss of expected answers is to introduce stop rules to decide when the processing should be stopped. Namely, the system can decide when the selection should be stopped without further processing. PIONwithStopRules is the one in which PION use some stop rules to decide when it would stop the selection and jump to provide its reasoning result. The idea of using stop rules is inspired by the investigation on human and animal search strategies in ecology and cognitive science. As discussed in LarKC deliverable D4.2.2, knowing when to stop is one of the most fundamental problems when engaging in any type of activity. Most real-world problems do not have a pre-defined completion criterion. The problem of search termination resurfaces in an aggravated form when a system faces more than a single problem at once. When time and effort need to be allocated to multiple tasks finding the right moments to switch between tasks constitutes a difficult optimization problem[16]. The interleaving framework can be considered as a processing which switches the tasks of reasoning and selection. Thus, using stop rules with the interleaving framework would provide a promising approach for the improvement of reasoning in PION. We will examine and report the evaluation of PIONwithStopRules in the sequel deliverable D4.7.2 entitled "Evolved Evaluation and Revision of plug-ins deployed in use-cases".

The architecture of PIONwithStopRules is shown in Figure 2.7. The PIONwith-StopRules plug-in uses the class *PIONStopRules* to specify a set of the stop rules.

The class diagrams of the PIONwithStopRules is shown in Figure 2.6. In the selection procedure of the PION reasoner with stop rules, the *PIONSelectionFunction* calls a method in the class *StopRules* to decide whether or not the processing should be stop. The existing implementation of the stop rules consist of *CardinalityCheckingRule* and *TimeCostCheckingRule*. *CardinalityCheckingRule* checks the cardinality of the

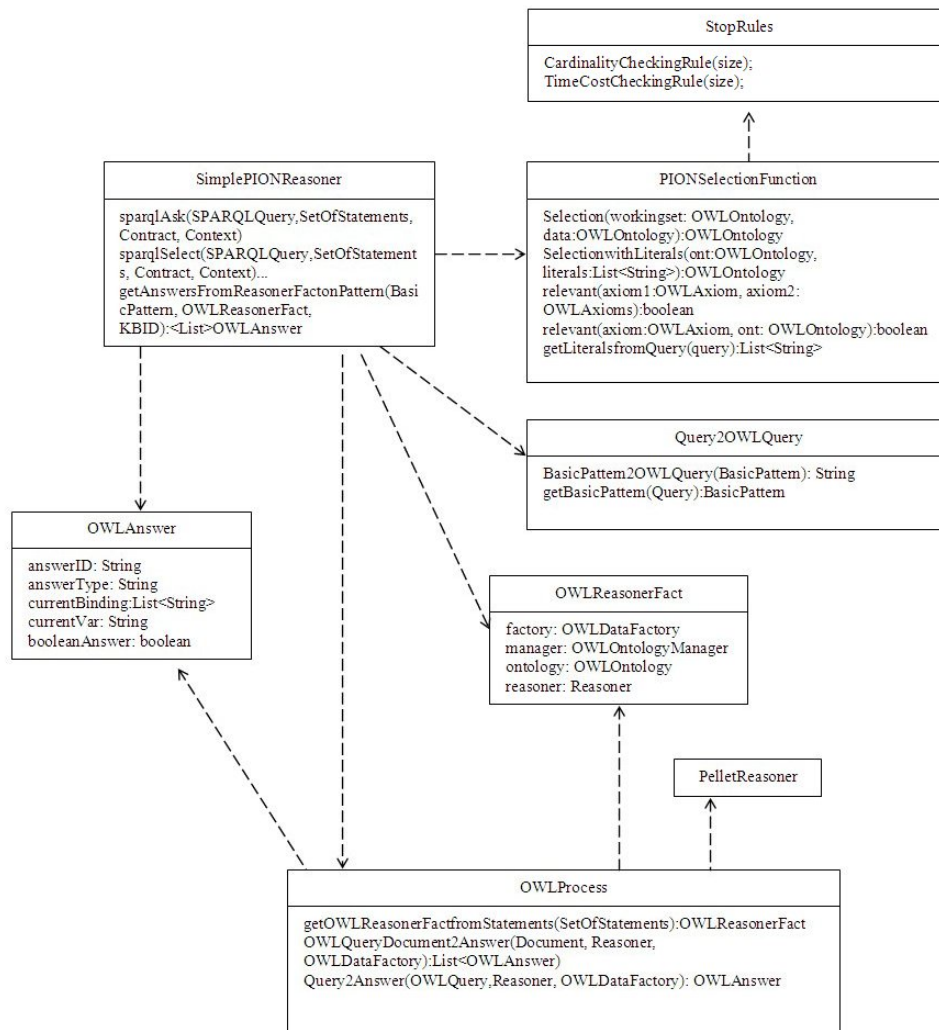


Figure 2.6: The Class Diagram of PIONwithStopRules

newly selected ontology (i.e., the size of the newly selectedOntology). If the size of the newly selected ontology is smaller than the size of the previous selected ontology, the system would stop. *TimeCostCheckingRule* checks the average time cost per newly selected axiom. If the average time cost per newly selected axiom is getting bigger, namely it is harder to find more newly selected axiom, then the system would stop.

The stop rules are specified as java codes. The following java codes show how to define the cardinality checking rule for the PIONwithStopRules.

```

public boolean CardinalityCheckingRule(long size) {

    boolean answer= false;
    // if it is true, that means that it should stop.

    last_increment = increment;
    increment = size - last_size;

    last_size = size;

    //if nothing has been found newly or the number of newly found

```



```
//axioms is decreasing, then it will stop

    if ((increment > last_increment)&&(increment > 0))
        {answer= false;}else {answer= true;};

    return answer;
}
```

The following java code defines the time cost checking rule.

```
public boolean TimeCostCheckingRule( long size)
{
    boolean answer= false;
    last_increment = increment;
    increment = size - last_size;
    last_size = size;

    Date currentDate = new Date();
    currentTime = currentDate.getTime();

    long total_time_cost = currentTime - lastTime;

    if (increment> 0)
    {
        current_time_cost = total_time_cost/increment;

        if (current_time_cost > last_time_cost)
        {answer= true;} else {answer=false;};
    }
    else {answer= true;};

    lastTime = currentTime;
    last_time_cost = current_time_cost;

    return answer;
}
```

2.6 Implementation of PIONWorkflow

PIONWorkFlow is the one in which PION is designed as a workflow which uses selection plug-ins, reasoner plug-ins, and a decider for the interleaving processing, as shown in Figure 2.7. One of the advantages of the PIONWorkflow is that it allows us to use various selectors in the processing. The processing of the PION workflow is similar with the processing of SimplePION. Figure 2.4 shows how to deal with sparqlAsk in SimplePION. That work chart can also be viewed in the workflow level as that of the PIONWorkFlow processing.

In the existing implementation of PIONWorkFlow, one can define a workflow which would launch a PION decider. At the beginning of the PION workflow processing, the decider first checks if the the ontology is consistent. If the ontology is consistent, then the decider will start the standard reasoning processing, namely, use the standard OWLAPI reasoner to obtain the result. If the ontology is inconsistent, then the

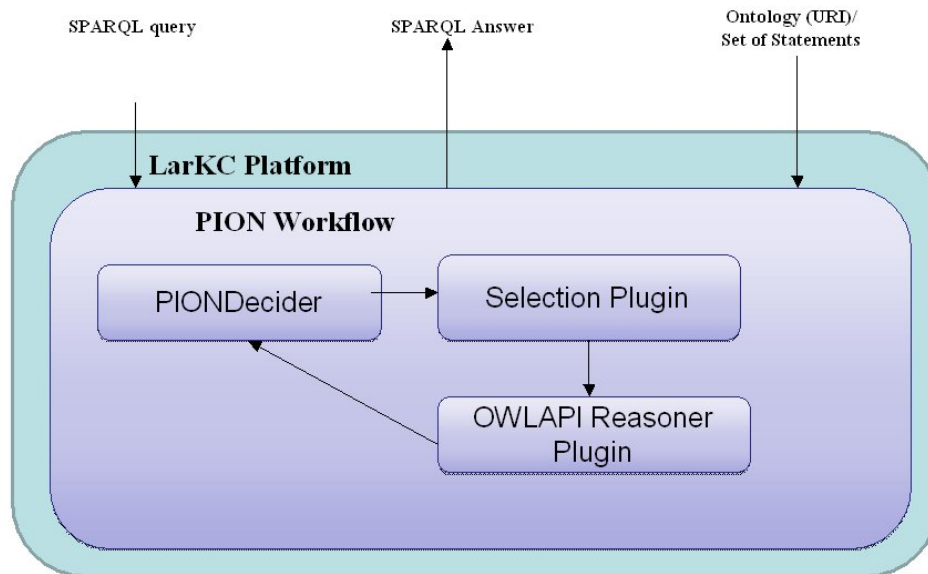


Figure 2.7: PIONWorkflow

decider will start a non-standard reasoning processing, namely an interleaving processing. In the beginning of the interleaving processing, the decider calls the selector *SelectOntologybasedOnQuery* to select a sub-ontology which is relevant to the query and checks if the selected ontology is consistent. If the selected ontology is consistent, then the decider will call *BasicOWLAPIReasoner* to reason with the selected ontology and check the result. If the selected ontology is inconsistent, then that means that we cannot find a proper and consistent sub-ontology, then the decider will stop and return an answer. For the SPARQL Ask processing, the system would return "false". If the selected ontology is consistent, then the decider will continue the interleaving processing until the inconsistent sub-ontology is selected or a positive answer is obtained (say, the answer is "true" in the SPARQL Ask processing).

2.7 Conclusion

We have investigated the implementation issues of several variants of PION with the LarKC platform. We have implemented those variants of PION and released them within the LarKC platform. The user guides of those variants of PION is available at the appendix of this document. We have not yet made the experiment of those variants of PION for the evaluation of implemented PION. We will examine and report the evaluation of those variants of PION in the sequel LarKC deliverable D4.7.2, which is entitled "Evolved Evaluation and Revision of plug-ins deployed in use-cases".



3. Anytime Reasoning by Ontology Approximation

The need for approximation on the Semantic Web raises the challenge to develop algorithms for anytime Semantic Web reasoning, and several attempts have been made to find suitable approximation strategies and study their effects in practice [20, 10, 18, 21]. Until now, this work has been limited in scope, has had a rather ad-hoc character (lacking a general framework for theory and application), and most importantly, results have often been inconclusive and show a need for a more thorough experimental analysis. A systematic evaluation of strategies and heuristics is challenging, and the results until now have been difficult to reproduce and compare.

This section, first introduces a framework for testing approximation algorithms systematically that will make the development of such methods easier, and thus increase their chances of adoption and deployment (see Section 3.1). To this end, we design a workflow consisting of a number of independent modules, which can be instantiated for different reasoning tasks and approximation strategies. The crucial elements of this workflow are separate modules for *approximation*, *reasoning* and *evaluation*. The first allows implementing approximation strategies by subset selection (eg. subsets of axioms, or of vocabulary), the second allows to specify a specific reasoning task (eg. instance retrieval, ontology classification, or consistency checking), and the final module allows to implement a suitable evaluation metric. This 3-step workflow has been realized into a workbench for studying anytime instance retrieval by ontology approximation. Both, the workbench and the results of our experiments are publicly available online ¹.

After introducing the 3-step workflow for studying approximate reasoning we present and discuss three concrete ideas on how this 3-step workflow can be realized in the LarKC platform. First, section 3.2 introduces the overall design of a LarKC Reason plug-in that implements part of the 3-step workflow. Such plug-in is described in terms of its constituent components and their functionality. Then, in Section 3.3 we present and discuss two alternative LarKC workflows that combine several plug-in types in order to realize our framework for anytime approximate reasoning in LarKC.

3.1 A Framework for Anytime Reasoning by Ontology Approximation

Our framework consists of a pipeline of three steps (see Figure 3.1), resulting in a new type of gain diagrams. These three steps allow to define (i) the particular approximation heuristic to be used, (ii) the reasoning task to which it should be applied, and (iii) the definition of a performance measure for evaluating the heuristic. This section describes each of these steps and the resulting gain diagrams.

Approximation step

The foundational results from [17] show that performing an approximate reasoning tasks on a logical theory can be transformed into executing a classical reasoner on a suitably approximated theory. Hence, the purpose of the approximation step is to take an ontology O and to return a sequence of approximations O_1, O_2, \dots, O_n . Very

¹<http://www.few.vu.nl/~gtagni/aboxreasoning/>

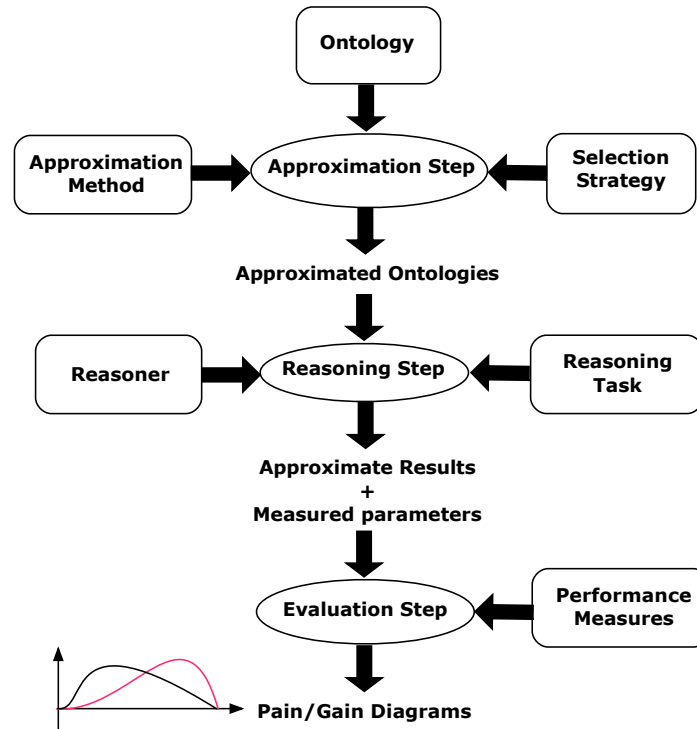


Figure 3.1: 3-step workflow for approximate reasoning experiments; every square box can be changed per experiment

often, such approximations can be phrased in terms of a selection method, operating on either the symbols appearing in an ontology (vocabulary selection), or on the set of axioms in an ontology (axiom selection), operating on either or both of the A-box and T-box of the ontology. The approximation component does not depend on a particular strategy: the only requirement is that for a given ontology, this module returns a sequence of approximations. Although our framework imposes no further constraints on the approximation step except that it produces ontologies that can be used in the reasoning step, some formal properties of such selection steps are desirable. Let O^* denote the semantic closure of an ontology, ie. all facts that can be derived according to its semantics. We then have *soundness* if each $O_i^* \subseteq O^*$, ensuring that the approximate results are correct (although possibly incomplete); *monotonicity* if $O_i^* \subseteq O_{i+1}^*$ for all $i = 1, \dots, n - 1$, ensuring that the successive approximations get more correct; and *completeness* if $O^* = O_n^*$, at which point the approximation has reached perfect quality.

Reasoning step

Approximation can be applied to different reasoning problems such as Instance Retrieval or Classification. All that our framework requires is that the reasoning step takes as input an ontology, and returns *answer-sets*. These answer sets could be instance-class memberships (for instance retrieval) or class-class subsumptions (for classification). It is these answer-sets that determine the quality of the approximation, and the computational efforts which determine the cost one has to pay.



Analysis step

In the analysis step of our framework we define the concepts of success and costs. These performance measures can be eg. the standard notions of recall (the number of retrieved facts in relation to all possible findings), or precision (the correctness of the given answers), or some non-standard notion of semantic proximity between the approximate answers and the perfect answers. More generally, we propose the concept of *gain*, which abstracts over the detailed measures and describes the results as ratio between possible and actual findings. *Pain* is the orthogonal notion describing the ratio between the costs of reasoning over an approximate ontology versus the non-approximate one. For specific examples of pain one could think of cost in terms of runtime or other computational resources, such as memory, user-interaction, database access, etc.

Gain-Pain diagrams: Obviously, we are interested in whether the gain (success-ratio of current answers against perfect answers) outweighs the pain (cost-ratio of current answers against perfect answers), in other words in the gain-pain difference. This ratio is plotted in our gain-pain diagrams which show at which point of the anytime computation the gain outweighs the pain (or not, as the case may be, and by how much). Figure 3.2 illustrates these measures. As the quality of the approximation increases along the x-axis from 0 – 100%, in this example the gain increases linearly while the pain increases much more slowly at the beginning, and rises more sharply in the final 20%. The combined performance measure (pain-gain curve) is calculated as the difference between these two, with the best performance achieved at about 75% of the approximation where the proportional gain maximally outweighs the proportional pain.

The ideal gain-pain curve rises sharply for the initial approximations of the input representing the desired outcome of a high gain and low pain in the early stages of the algorithm. Although such a convex gain-curve is the most ideal, even a flat gain curve at $y = 0$ is already attractive, because it indicates that the gains grow proportionally with costs, giving still an attractive anytime behaviour.

Notice that gain-curves always start in $(0, 0)$, since for the empty input both gain (e.g. recall) and pain (e.g. runtime) are 0, hence their difference is 0. Gain curves always ends in $(100, 0)$, since for the final perfect approximation both recall and runtime are 100%, hence their difference is again 0. Also notice that gain-pain curves can be negative when the proportional pain outweighs the proportional gain for certain approximations.

3.2 Approximate Reasoning Plug-in

The simplest way to implement the 3-step workflow for approximate reasoning in LarKC is as a single LarKC reason plug-in. The modular design of this plug-in allows for changing some of its components and replacing them with other implementations of the same component. For example, the approximation module is an interface that can be instantiated by different classes allowing us in this case to experiment with multiple approximation methods. The same holds for the selection module which allows us to plug different components implementing different selection strategies.

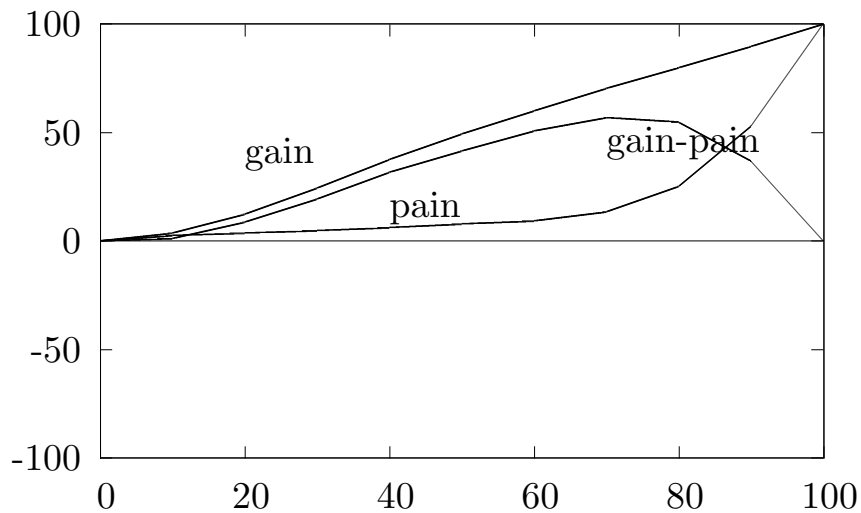


Figure 3.2: gain, pain and gain-pain curves

In the following we will discuss the overall design of a LarKC plug-in that implements an anytime, approximate reasoner that implements part of the functionality provided by the workbench described above. Figure 3.3 depicts the overall architecture of our anytime approximate reasoner plug-in illustrating the its components. Figure 3.4 gives an overview of the main classes involved in the design of the current version.

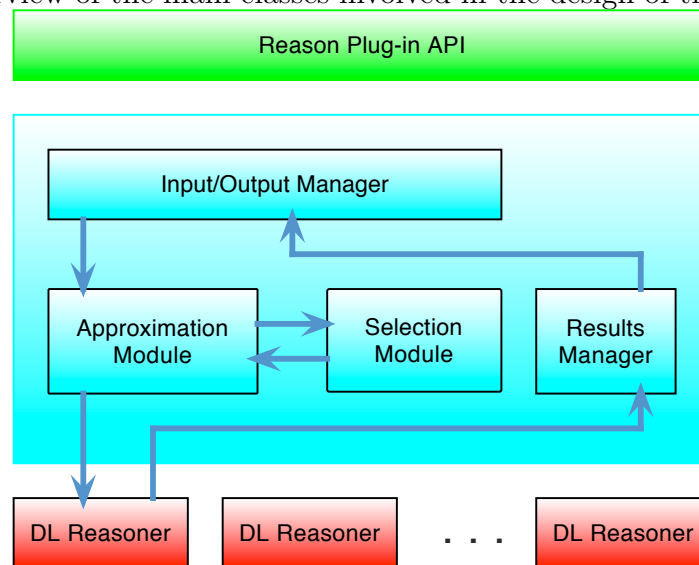


Figure 3.3: Architecture of a reason plug-in in LarKC implementing the 3-step workflow introduced above

Approximation Component This module implements the approximation step of the 3-step workflow for anytime approximate reasoning presented above. Given an ontology this modules returns a set of approximated ontologies. More specifically, this modules defines an interface for approximating ontologies. The interface can be instantiated by different approximation classes each of which implements a specific approximation method. In the current implementation the approximation component takes as input an OWL ontology and returns a sequence (possibly a singleton) of approximated ontologies by selecting a subset of the ontology’s vocabulary (atomic



concept names) and rewriting the set of terminological axioms according to the approximation method described in [17]. The approximation module also provides a way to produce incremental approximations of an ontology whereby given an initial ontology it returns a sequence of approximated ontologies (T-boxes) each of them based on an incremental subset of the vocabulary of the ontology. For example, given ontology $O = (T, A)$ the module is able to produce 10 different approximations (T_i, A) where each T_i is an approximated T-box based on 10% of the vocabulary of the ontology. The current implementation of this module approximates only the terminological part of an ontology leaving the assertional part intact. However, it is possible to replace this component by one that approximates both the terminological and the assertional parts of an ontology.

Selection Component The selection component of this plug-in refers to the selection step in the 3-step workflow presented above. This component is implemented as an interface that defines the basic functionality that must be provided by every selection strategy. For the purposes of approximate reasoning we have defined two additional sub interfaces. The first one, a *vocabulary selection strategy* interface, defines the common functionality provided by methods that return a subset of the vocabulary of the ontology. The second one, an *axiom selection strategy* interface, specifies the minimal functionality that must be provided by methods that return a subset of the set of terminological axioms defined in the ontology. A selection module takes an ontology as input and returns either a subset of the vocabulary of the ontology (vocabulary selection strategies) or a subset of the terminological axioms defined in the ontology (axiom selection strategies).

In the current implementation of our workbench we have implemented six different vocabulary selection strategies, which for the purposes of implementing this plug-in they are implemented as six different selection modules. In the following, we briefly describe each of the selection strategies.

- *Random (R)*: This function randomly selects a set of atomic concept names from the ontology's vocabulary set. The strategy is implemented by the *RandomSelection* class.
- *Most Referenced (MR)*: This function selects concept names according to the number of times they appear in terminological axioms. The class *MRSelection* implements the behaviour of this strategy.
- *Most Members (MM)*: This function selects atomic concept names based on the number of instances they have. At each approximation step concepts are sorted according to the number of instances that were retrieved in the previous step. In case there is no feedback from the previous reasoning step concepts are sorted according to the Most Referenced strategy. The rationale behind this strategy is to select as early as possible those concepts that can produce the largest number of instances, thus producing the greatest increase in recall. The main disadvantage of this strategy is that concepts must be sorted at each step of the approximation process. In addition to this, the strategy must be combined with another strategy to produce an initial ordering. This strategy is implemented in the plug-in by the *MMSelection* class.



- *Restriction Class (RC)*: This function gives higher priority to the fillers of quantified concept expressions and to their respective sub concepts. If the number of such elements is less than desired number M the additional concepts are chosen based on the number of instances asserted in the assertional part of the ontology. The rationale of this strategy is that property restrictions are used for defining classes implicitly. Consequently, these classes may contribute to retrieving a large number of instances. The main disadvantage of this strategy is that not every class in an ontology is defined through property restrictions, a characteristic that makes this strategy incomplete. Therefore, as with the previous strategy this one needs to be complemented with another strategy for selecting classes that are not defined through property restrictions. The class *RDSelection* implements this selection strategy.
- *Most Direct Subclasses (MDS)*: This function selects atomic concepts based on the number of direct subclasses they have. The first time this strategy is used, atomic concepts are sorted in decreasing number of direct subclasses and each successive call to this function returns the next set of concepts. As with the Most Referenced strategy concepts can be sorted only once at the beginning of the anytime reasoning process. The MDS strategy is implemented by the *MDSSelection* class.
- *Least Direct Subclasses (LDS)*: This function is the opposite of the MDS function. The rationale behind using this strategy is that concepts with the least number of subclasses are more specific and tend to be used to annotate large number of individuals. The class *LDSSelection* implements the behaviour of this strategy.

Input Manager This component is responsible for splitting the ontology into its terminological and assertional parts. This is required since the current implementation approximates only the terminological part of an ontology. An advantage of separating the terminological from the assertional part is that this allows the approximate reasoner to combine a single terminology with multiple pieces of instance data.

Reasoner Although the reasoner component could be built-in into the approximate reasoner plug-in we have decided to leave it outside the reasoner. The main advantage of doing so is that it allows us to (re)use multiple standard DL reasoners. In the current design access to DL reasoners is accomplished through either the OWLAPI or the OWLLink interfaces.

Using the context parameter of the Reason interface it is possible to invoke the approximate reasoner and request to answer the same query using different approximations of the same ontology. A decider, or the end-user application, would invoke the reasoner providing a SPARQL query and a data set. The reasoner would then solve the query by using an approximated ontology as computed by the approximation module and return the results back to a LarKC Decider or user application. In case further reasoning is required the approximate reasoner could be invoked again with the same query and the same input ontology. This time, the reasoner would approximate the ontology using a bigger subset of the vocabulary and then solve the query using this new approximated version of the ontology. The context parameter in the Reason interface could be used to keep state-related information between calls

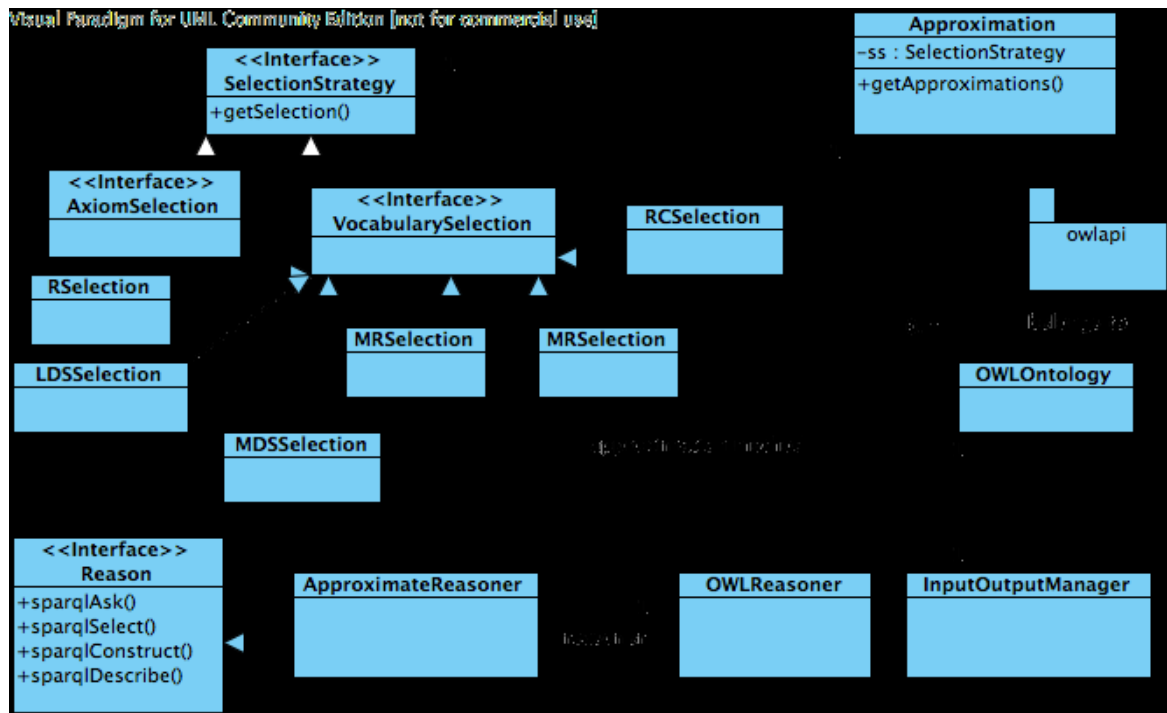


Figure 3.4: Class diagram depicting the main classes that make up the approximate reasoner plug-in

to the reasoner, for example, to control the size of the subset of the vocabulary upon which the next approximation should be computed.

3.3 Two LarKC Workflows for Anytime Approximate Reasoning

The previous section described the overall architecture of a LarKC Reason plug-in that implements part of the functionality provided by the 3-step workflow introduced above in terms of a series of modular components for *selection*, *approximation* and *reasoning*. Although the plug-in’s architecture is rather simple, with only three main (sub)components, it highlights several points of reuse. In the following we will identify these points and discuss how to implement the functionality provided by the approximate reasoning plug-in as a LarKC workflow using several plug-in types.

- *Approximation as Transformation plug-in*: An *InformationSetTransformer* plug-in takes as input an *InformationSet*, eg. the URL of an OWL ontology, and returns a (transformed) version of the input *InformationSet*. One possible implementation of such interface could return an approximated version of the input ontology, i.e. the transformation step consists in approximating the ontology. Note that this is literally the case of our approximate reasoner as the approximation of an ontology is defined in terms of rewriting the terminological axioms of the ontology based on a subset of the vocabulary. Such a transformer plug-in would only have to invoke the specific selection method and allow for context information to be passed to it in order to implement incremental subsetting of the vocabulary.

- *Approximation as Selector plug-in:* Another possibility is to implement the approximation module as an instance of a *Selector* plug-in interface. In this case, the input parameter of type *SetOfStatements* represents the ontology that needs to be approximated. The plug-in's output (of type *SetOfStatements*) represents the approximated version of the input ontology. As with the Transformation-based approximation discussed above, a selection plug-in would have to invoke a specific selection method to select the subset of the ontology's vocabulary or, alternatively, each selection strategy could be implemented as a different selection plug-in that not only implements a specific selection function but also approximates an ontology based on this selection method. The selection plug-in should also be able to accept context information that allows the client class to pass state-related information, eg. the percentage of vocabulary that must be selected in the current call.
- *Selection as Selector plug-in:* In case the approximation of an ontology is based on selecting a subset of the terminological axioms of the ontology the selection module's functionality could be implemented by a *Selector* plug-in. Such plug-in would return a *SetOfStatements* representing a subset of the terminological axioms. Different axiom selection strategies could be implemented by different instances of this *Selector* interface.
- *Reasoning Component as Reasoner plug-in:* An obvious point of reuse in our approximate reasoning plug-in is the use of a reasoner component. The functionality of such module could easily be implemented by a separate LarKC plug-in that provides alternative reasoning capabilities in terms of expressivity, computational resources, reasoning paradigm, etc. In particular, for the implementation of our reasoner we are planning to use standard DL reasoners and access them through the OWL API and OWLLink plug-ins that provide a wrapper component over existing OWL reasoners.

LarKC workflows for Anytime Approximate Reasoning

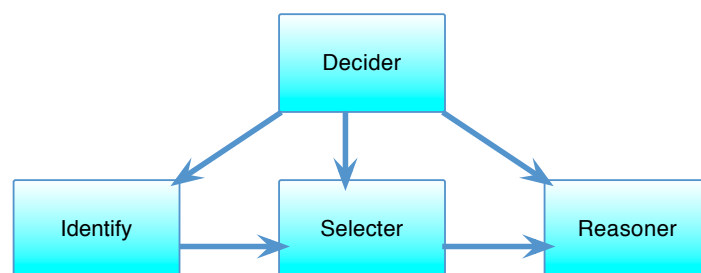


Figure 3.5: LarKC workflow for anytime reasoning by ontology approximation using a *Selector* to approximate ontologies

Figure 3.5 depicts a LarKC workflow for studying anytime reasoning by ontology approximation. The workflow consists of four plug-in types. The *Identify* plug-in is responsible for identifying the data over which reasoning will be done to answer a given query. The *Selector* plug-in receives the ontology identified by the previous plug-in and returns an approximation of that ontology. Depending on the desired selection strategy and approximation method different instances of this plug-in will be invoked and executed. Once the *Selector* plug-in computes the approximated version of an ontology an appropriate *Reasoner* plug-in can be invoked with the approximated

ontology and original query as parameters. Here, the selection of the reasoner may depend on many factors such as QoS parameters. The *Reason* plug-in is responsible for answering the given query using the given approximated ontology. The specific instance of this plug-in type could be one that invokes an external DL reasoner through the OWL API paradigm or OWLLink protocol. The last component of the workflow is a *emphDecider* plug-in. The *Decider* is responsible for controlling the execution of the workflow and deciding whether further approximation must be done in order to produce more accurate results. For this, the decider must be capable of keeping track of the state between multiple invocations to the selection plug-in. This information is part of the context information that is passed to the *Selecter* plug-in in each invocation and could include the percentage of vocabulary to be chosen in the current approximation step.

One way the Decider could determine whether to use another approximated version of the ontology could be by analyzing the data obtained from the Gain-Pain diagrams produced by the *Evaluation* step of the 3-step workflow for approximate reasoning, assuming the *Decider* (or any other component in the LarKC workflow) implements the metrics for evaluating the results returned by the reasoner.

Figure 3.6 illustrates the overall design of a second LarKC workflow where the *Selecter* plug-in has been replaced by a *InformationSetTransformer* plug-in that is responsible for approximating a given ontology. As in the previous case, different approximation methods and selection strategies could be implemented by different *InformationSetTransformer* plug-ins.

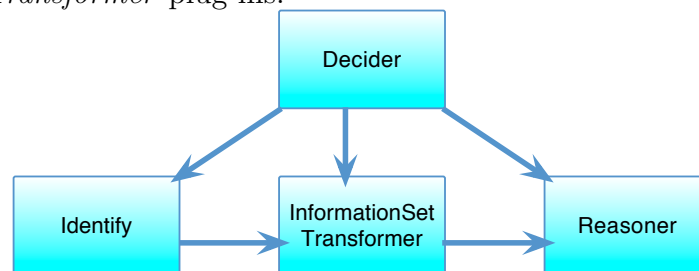


Figure 3.6: LarKC workflow for anytime reasoning by ontology approximation using a Transformer to approximate ontologies



4. I-ReaSearch: Unifying Selection and Reasoning with User Interests

In D4.3.1 and D2.3.1, we introduced how user interests can be involved in the process of unifying selection and reasoning under several granularity based strategies (More specifically, the discussion is related to the starting point strategy). Nevertheless, for the implementation as plug-ins on the LarKC platform, several ideas and initial design need to be further investigated. In addition, how the proposed strategy scales still need further discussion. In this chapter, we will focus on concrete implementations of unifying selection and reasoning (“ReaSearch” for short in the LarKC Project) with user interests (namely, Interests based ReaSearch, or “I-ReaSearch” for short). Firstly, we provide an introduction of the general framework for I-ReaSearch. Secondly, 2 concrete strategies and related implementations of unifying selection and reasoning with user interests (I-ReaSearch) are discussed. At last, some initial evaluation on scalability for the proposed methods is also provided.

4.1 A General Framework of I-ReaSearch

The “ReaSearch” approach proposed in [9] is aimed at solving the problem of scalability for Web-scale reasoning. It’s core philosophy is to select an appropriate subset of semantic data for reasoning and is trying to solve the scalability issue by incomplete reasoning since the dataset that is acquired from the Web itself are incomplete anyway. The criterion and concrete methods for selecting a good subset is one of the main tasks in the LarKC project.

Our main efforts are on the direction of “context-aware” approaches. In D2.3.1 and D4.3.1 we developed “The starting point strategy” to solve the data selection problem, which emphasize the power of user interests during the selection process and how it can be used as a factor to provide good selected subset for the reasoning process. Recently, we have developed two concrete methods and implementations in the form of plug-ins that are based on the idea of Interests Based unification of selection and reasoning (Following the notion in [9], we title the efforts as “I-ReaSearch”, which means unifying reasoning and search with Interests). The process of I-ReaSearch can be described as the following rule:

$$hasInterests(U, I), hasQuery(U, Q), executesOver(Q, D), \neg contains(Q, I) \rightarrow IReaSearch(I, Q, D),$$

Where $hasInterests(U, I)$ represents that the user “ U ” has a list of interests “ I ” and can be acquired, $hasQuery(U, Q)$ represents that there is a query input “ Q ” by the user “ U ”, $executesOver(Q, D)$ denotes that the query “ Q ” is executed over the dataset “ D ”, $\neg contains(Q, I)$ represents that the query “ Q ” does not contain the list of interests “ I ”, $IReaSearch(I, Q, D)$ represents that by utilizing the interests list “ I ” and the query “ Q ”, the process of unifying selection and reasoning is taken on the dataset “ D ”.

This approach implements the idea of “refining querying by using rules” proposed in [6]. Currently, there are two main methods under “I-ReaSearch”, namely, the implementations of $IReaSearch(I, Q, D)$ are with two directions. The first one is user interests based query refinement, and the second is interleaving selection and reasoning



(now focusing on querying) based on user interests. Both of these strategies utilizes user interests as the context, but the processing mechanisms are different.

4.2 Interests-Based Query Refinement

For the strategy of user interests based Query Refinement, it adds more constraints to the user input query according to user interests extracted from some historical sources (such as previous publication, visiting logs, etc.). The process can be described by the following rule:

$$hasInterests(U, I), hasQuery(U, Q), executesOver(Q, D), \neg contains(Q, I) \rightarrow refinedAs(Q, Q'), contains(Q', I), executesOver(Q', D).$$

In this rule, $refinedAs(Q, Q')$ represents that the original query “Q” is refined by using the list of Interests as “Q’”. $contains(Q', I)$ denotes that “Q’” contains the list of Interests “I”. $executesOver(Q', D)$ represents that the refined query “Q’” executes over the dataset “D”. Namely, “ $refinedAs(Q, Q'), contains(Q', I), executesOver(Q', D)$ ” implements $IReaSearch(I, Q, D)$ in the I-ReaSearch general framework.

Based on the upper rule, we emphasize that this approach does not select the subset for querying in advance. Instead, it utilizes the user context to provide a rewritten query.

From the implementation perspective, the idea of user interests based Query Refinement has been implemented as a reasoner plug-in (“Interest-Based Reasoner”) following LarKC reasoner API. The core implementation is around SPARQL query rewriting based on user interests acquired from “Context” defined in the parameter list of the LarKC reasoner API. Since currently, the “Context” parameter is empty in the defined interface, we implement an interface as a new class.

In this class, we defined some member variables to store basic information of the user who uses LarKC system. These variables include variables to identify the user and the data sources which contains the user’s Interests. These variables is prepared to extract interests of the user. This class also includes variables which store interests of the user directly, e.g. the literal words or the URIs of interests. The variables to identify the user include the user’s name and the user’s URI. They both represent the current user, and at least one of them should have value in actual usage. The variables to identify the data source containing the user’s interests can be a URL to give the location of the interests dataset in rdf format, a graph name if the data source is a named graph of a dataset which is in memory, or a reference to point to a variable representing a set of statements, and at least one of them should have value in actual usage.

This class also contains methods to extract interests based on the information stored in variables mentioned above. Basically, we try the variables at a certain sequence with priority, according to the extent of difficulty of getting user interests. In current design, we adopt a first-easy-last-hard sequence. Speaking in detail, the sequence is like this: 1)firstly, we get interests form the variables which directly store them, and if they have values, return them; 2)if they have no value, we query the user interests form data source given in context, and if get results, return them; 3)if the data source doesn’t exist, a real-time extraction of user interests through their background is needed. In current plug-in, the first two have been contained, the last one is still in developing and not integrated in current plug-in.



4.3 Interleaving Selection and Reasoning Based on User Interests

For the Strategy of interleaving selection and reasoning based on user interests, it emphasize a selection step before querying and reasoning on the data, since user interests might help to find a more relevant sub dataset for each specific user compared to querying on the whole. The process can be described by the following rule:

$$hasInterests(U, I), hasQuery(U, Q), executesOver(Q, D), \neg contains(Q, I) \rightarrow Select(D', D, I), executesOver(Q, D').$$

where “Select(I,D’)” represent the selection of a sub dataset “D’” from the original dataset “D” based on the interests list “I”, and *executesOver(Q, D’)* represents that the query is executed over the selected sub dataset “D’”. Namely, in the upper rule, “*Select(D’, D, I), executesOver(Q, D’)*” implements *IReaSearch(I, Q, D)* in the I-ReaSearch general framework.

From the implementation perspective, this method has been implemented as a interests based selector plug-in (“Interest-Based Selector”) following LarKC selector APIs. The implementation of the “Context” parameter in the selector APIs is the same as in “Interest-Based Reasoner”. The Interest-Based Selector plug-in takes user interests from the “Context” parameter and select a subset of the original dataset. Querying on the selected sub dataset can be performed based on any existing reasoner plug-ins with query functionality.

4.4 Implementation of Interests-Based Reasoner Plug-in

We developed a plug-in named “Interest-Based Reasoner”. Its function is to refine the input SPARQL query with user interests, then execute the refined query rather than the original query. By using this plug-in, the users can get more relative answers which are relevant to their background (such as interests which can be acquired from the context).

Figure 4.1 shows the architecture of the Interests-Based Reasoner plug-in and how it works within the LarKC Platform. There are 3 input parameters which are important for the Interests-Based Reasoner, namely, SetofStatement, Context, and SPARQLQuery. In a LarKC workflow, the SetofStatement could come from the previous plug-in (e.g. a selector). In order to acquire user interests, we implemented a ContextImpl class for the Context interface in the plug-in. ContextImpl includes some properties and methods which are essential for the function of the Interest-Based Reasoner. These properties include the URI/name of the user, the user interests data, etc. The ContextImpl class also contains some methods which are responsible for getting user interests. So, the context parameter passed to the Interest-Based Reasoner should be consistent with the definition of ContextImpl.

At run time, the plug-in accepts the context parameter and invokes methods of the class ContextImpl to get the user interests information, then pass this information to the query refinement module to refine the query, then execute the refined query to get the answer.

The context parameter is responsible for providing enough information to get user interests. To allow multiple approach to get user interests, we make redundant design

in class ContextImpl. That means, there are multiple properties, but not all of them need to have values in real use cases. So, if the user interests information is expressed as the format of word items, it can be directly pass by the context parameter (defined in the Interests-Based Reasoner plug-in); if the user interests information can be acquired from an external dataset (provide in the form of an URI, or reference type supported by the LarKC Platform), and the user name (or URI) is provided, an execution of SPARQL query is needed to get the user interests. After query refinement, another execution of the refined SPARQL query will be done to get the final answer. All the execution of SPARQL query is carried out by SPARQLQueryExecuter(a utility included in the current LarKC Platform).

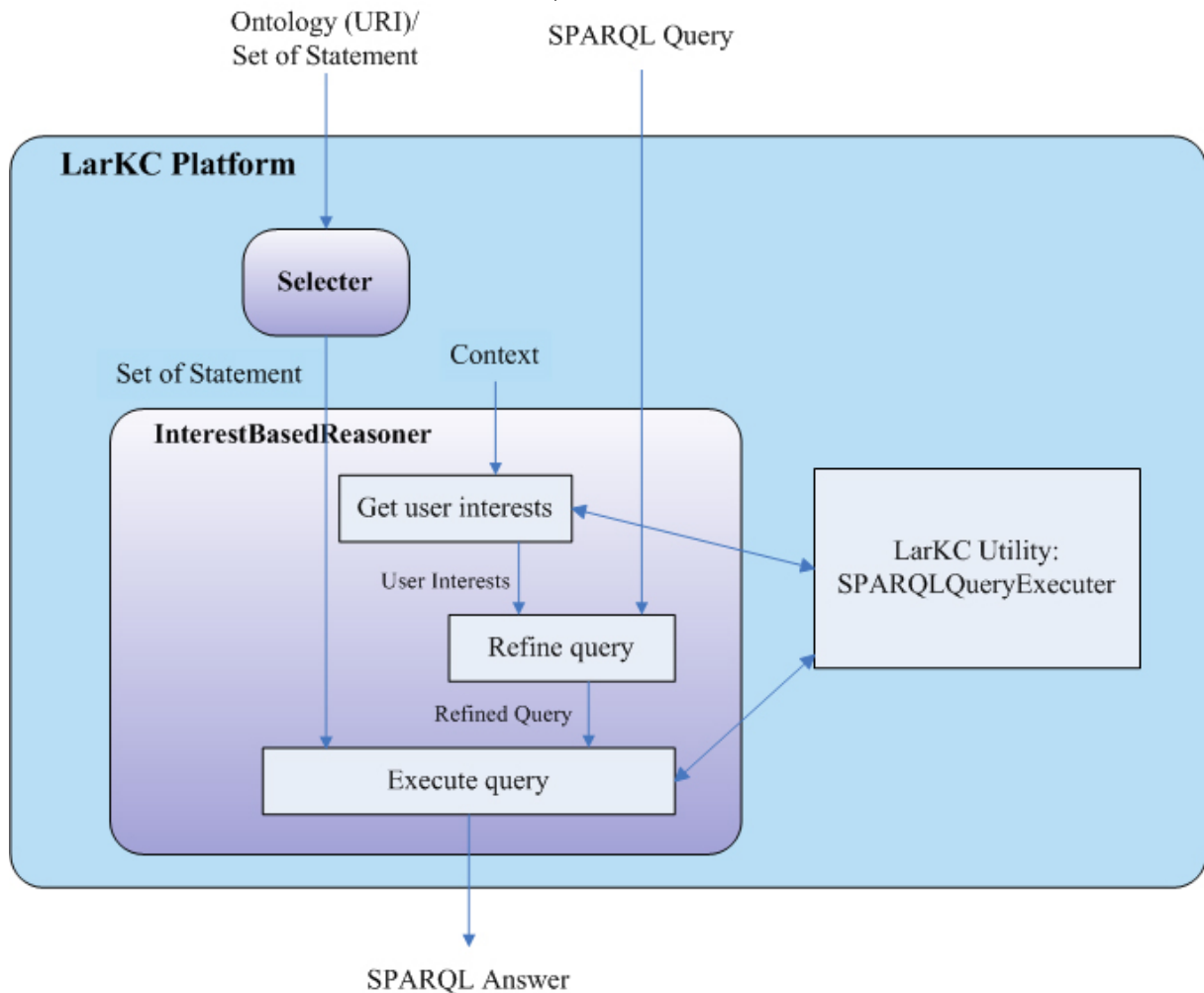


Figure 4.1: the Interest-Based Reasoner

Figure 4.2 is the Class Diagram of the Interest-Based Reasoner plug-in. The class InterestBasedReasoner is the major class of the plug-in, written according to the LarKC reasoner API. The class QueryR includes the query refinement module. The class ContextImpl is the implementation of Context interface, including modules to get user interests. The class UserInterests is defined for a user's a group of interests as a whole.

Current Interest-Based Reasoner supports SPARQL select query. Figure 4.3 shows the how a SPARQL SELECT Query is Processed in the Interest-Based Reasoner plug-in. The test condition is a Boolean value pass by a property of context parameter. To keep the property is for the convenience of testing and compare. If the value is



Figure 4.2: the Interest-Based Reasoner Class Diagram

true refine the query, then execute the refined query, then return answers. Otherwise, execute the query without refinement and return.

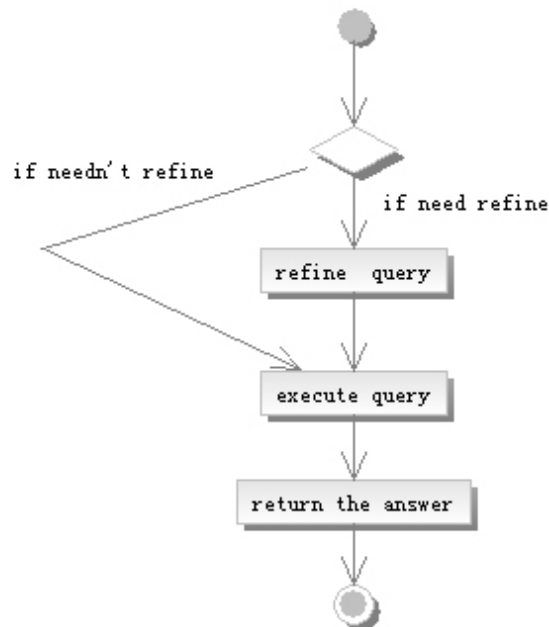


Figure 4.3: SPARQL SELECT Processing in the Interest-Based Reasoner

Figure 4.4 shows how to refine a SPARQL query. Firstly, this module analyses the input SPARQL query, then decides if the query could be refined. Actually, this step includes a group of test condition, because in some cases, the query couldn't be refined, and we need to exclude these cases. If the query could be refined, then the

query will be refined by user interests(a input parameter of query refinement module), then return the new query.

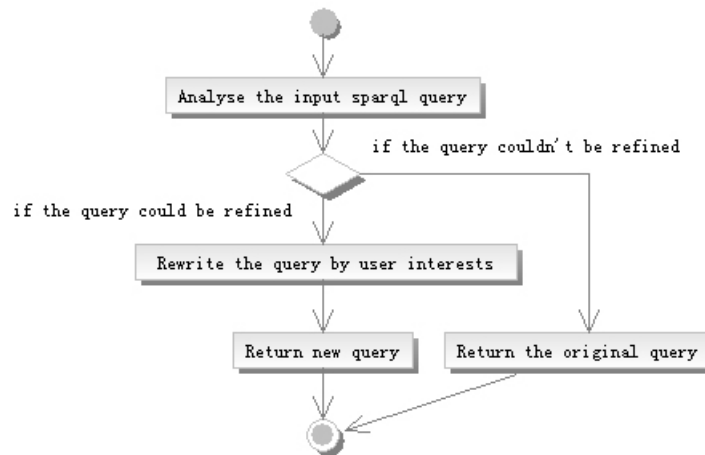


Figure 4.4: Query Refinement Processing

Currently the plug-in is still in testing phase. The relevant documents (e.g. API, User Guides, and so on) will be provided together with the release of this plug-in.

For Interleaving selection and reasoning based on user interests, we also developed an Interest-Based selector plug-in, which is introduced in D2.3.2.

4.5 An Initial Evaluation on the Scalability of Proposed Strategies

The two strategies of I-ReaSearch are based on different methods to solve the scalability problems. As an illustrative example, we take the SwetoDBLP dataset which is divided into 22 sub dataset. We evaluate the two implemented strategies by using these datasets at different scales. A comparative study is provided in Figure 4.5. Two users are taken as examples, namely Frank van Harmelen and Ricardo Baeza-Yates. Top 9 retained interests (as defined in D 2.3.1) for each of them are taken from the retained interests RDF dataset (reported in <http://wiki.larkc.eu/csri-rdf>) to unify the selection and reasoning process. Three different kinds of querying strategies are performed on the gradually growing dataset (each time add 2 subset with the same size, around 55M for each, and 1.08G over all), namely, the original query by user inputs, user interests based query refinement, and interleaving selection and reasoning based on user interests.

As shown in the figure. Since “user interests based Query Refinement” takes more constraints compared to the original input query by the user, it requires more processing time, and when the size of the dataset is growing, the processing time is also growing very rapidly, which means that this method does not scales well if we just consider the required time. Although this method takes more time, as reported in D2.3.1, the quality of acquired query results is much better than that without user interests refinement.

Since “interleaving selection and reasoning based on user interests” select relevant sub dataset in advance, the required query time reduces a lot, and as the size of

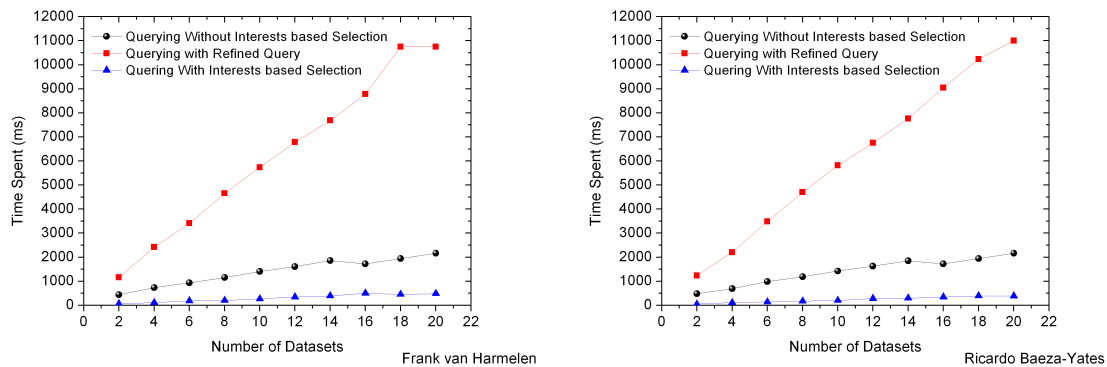


Figure 4.5: A comparative study on the Scalability of Proposed Strategies

the dataset grows, compared to the unrefined query and the one that is using “user interests based Query Refinement”, the query time is always comparatively shorter and is not increasing very fast. Meanwhile, its query results quality is the same as “user interests based Query Refinement”. Hence, this method scales better.

In this section, we just give a very initial evaluation on the scalability of the I-ReaSearch approaches (We focus on the scalability of querying). More detailed Evaluation on how the “interleaving selection and reasoning based on user interests” method scales can be found in D2.3.2.



5. Conclusion

In this document, we have investigated various approaches of interleaving reasoning and selection, which include PION, a framework of interleaving reasoning and query-based selection, and the approach of ontology approximation by interleaving reasoning and language-based selection, and I-ReaSearch, a framework of interleaving reasoning and user-interest based selection. We have discussed the implementation issues of the proposed approaches.

- For PION, we have explored the implementation of variants of PION for interleaving reasoning and selection within the LarKC platform, which includes i) the DIGPION which uses the DIG interface reasoner to call an external PION system, ii) the SimplePION which provides basic implementation of the interleaving of reasoning by an OWLAPI reasoner and selection by syntactic-relevance-based selection functions, iii) the PIONwithStopRule which uses a set of stop rules in the procedure of interleaving reasoning and selection, and iv) the PIONWorkflow which is designed to be an interleaving workflow of reasoning and selection within the LarKC Platform. We have also provided a user guide in the appendix as a reference manual of the released PION plug-ins and workflow. We are going to report the experiment of those variant PION for the evaluation of the proposed approaches in the sequel LarKC deliverable about the evaluation of implemented reasoner plug-ins.
- For the framework of interleaving reasoning and language-based selection, we have presented an approach of anytime instance retrieval by ontology approximation, and discussed the design and the implementation issue with the LarKC platform.
- For the interleaving framework with user-interest-based selection, we have presented several concrete selection strategies, discussed the implementation issue of unifying selection and reasoning with user interests (I-ReaSearch), and reported an initial evaluation on scalability for the proposed methods. The initial test shows that the approach of interleaving reasoning and user-interest-based selection can improve the efficiency of reasoning.



REFERENCES

- [1] Owl semantics and abstract syntax. Technical report, 2004.
- [2] Sean Bechhofer. The DIG description logic interface: DIG1.0. Technical report, 2002.
- [3] Sean Bechhofer. The DIG description logic interface: DIG1.1. Technical report, 2003.
- [4] Sean Bechhofer. DIG 2.0 roadmap. Technical report, 2006.
- [5] Sean Bechhofer, Ralf Möller, and Peter Crowther. The DIG description logic interface. In *International Workshop on Description Logics (DL2003)*. Rome, September 2003.
- [6] T. Berners-Lee and M. Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. HarperSanFrancisco, 1999.
- [7] Alexander Budanitsky and Graeme Hirst. Semantic distance in wordnet: An experimental, application-oriented evaluation of five measures. In *Workshop on WordNet and Other Lexical Resources, 2nd meeting of the North American Chapter of the Association for Computational Linguistics*. Pittsburgh, PA., 2001.
- [8] Samir Chopra, Rohit Parikh, and Renata Wassermann. Approximate belief revision preliminary report. *Journal of IGPL*, 2000.
- [9] D. Fensel and F. van Harmelen. Unifying reasoning and search to web scale. *IEEE Internet Computing*, 11(2):96, 94–95, 2007.
- [10] Perry Groot, Heiner Stuckenschmidt, and Holger Wache. Approximating description logic classification for semantic web reasoning. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *ESWC*, volume 3532 of *Lecture Notes in Computer Science*, pages 318–332. Springer, 2005.
- [11] Z. Huang, F. van Harmelen, and A. ten Teije. Reasoning with inconsistent ontologies. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI'05*, 2005.
- [12] Zhisheng Huang and Frank van Harmelen. Using semantic distances for reasoning with inconsistent ontologies. In *Proceedings of the 7th International Semantic Web Conference (ISWC2008)*, 2008.
- [13] Zhisheng Huang and Cees Visser. Extended DIG description logic interface support for prolog. Deliverable D3.4.1.2, SEKT, 2004.
- [14] Zhisheng Huang, Yi Zeng, Stefan Schlobach, Annette den Teije, Frank van Harmelen, Yan Wang, and Ning Zhong. D4.3.1 - strategies and design for interleaving reasoning and selection of axioms, September 2009. Available from: <http://www.larkc.eu/deliverables/>.



- [15] Thorsten Liebig, Marko Luther, Olaf Noppens, Mariano Rodriguez, Diego Calvanese, Michael Wessel, Matthew Horridge, Sean Bechhofer, Dmitry Tsarkov, and Evren Sirin. OWLlink: DIG for OWL2. In *OWLED*, 2008.
- [16] Hansjorg Neth, Lael J. Schooler, Jorg Rieskamp, Jose Quesada, Jie Xiang, Rifeng Wang, Lijuan Wang, Haiyan Zhou, Yulin Qin, Ning Zhong, and Yi Zeng. D4.2.2 - analysis of human search strategies, September 2009. Available from: <http://www.larkc.eu/deliverables/>.
- [17] Marco Schaerf and Marco Cadoli. Tractable reasoning via approximation. *Artificial Intelligence*, 74(2):249–310, 1995.
- [18] S. Schlobach, E. Blaauw, M. El Kebir, A. ten Teije, F. van Harmelen, S. Bortoli, M. Hobbelman, K. Millian, Y. Ren, S. Stam, P. Thomassen, R. van het Schip, and W. van Willigem. Anytime classification by ontology approximation. In Ruzica Piskac et al., editor, *Proceedings of the workshop on new forms of reasoning for the Semantic Web: scalable, tolerant and dynamic*, pages 60–74, 2007.
- [19] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *OWLED 2007: Proceedings of the Third International Workshop on OWL: Experiences and Directions, Innsbruck, Austria*, volume 258 of *CEUR Workshop Proceedings*. Sun SITE Central Europe, 2007.
- [20] Heiner Stuckenschmidt. Partial matchmaking using approximate subsumption. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, 2007.
- [21] H Wache, P Groot, and H Stuckenschmidt. Scalable instance retrieval for the semantic web by approximation. *Lecture notes in computer science*, 3807:245, 2005.



A. PION within the LarKC Platform: User Manual

A.1 Using SPARQL-DL to Express OWL-DL Formulas

We need SPARQL-DL to express the SPARQL queries with the LarKC platform. In order to translate DL expressions into RDF triples, we use the OWL-DL method which is recommended in [1]¹. Here are several examples of OWL Expressions in SPARQL-DL:

```
?- subclassOf(Wine, PotableLiquid)
// to ask whether or not wine is a subclass of potable liquid

    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX wine: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
    PREFIX food: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/food#>
ASK
    WHERE { wine:Wine rdfs:subclassOf food:PotableLiquid.}

?- subclassOf(Bordeaux, and(SweetWine, TableWine))
// to ask whether or not Bordeaux is a SweetWine and TableWine

    PREFIX rdfs:      <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX rdf:       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

    PREFIX owl:     <http://www.w3.org/2002/07/owl#>
ASK
    wine:Bordeaux rdfs:subclassOf _:x.
        _:x owl:intersectionOf _:y1.
        _:y1 rdf:first wine:SweetWine.
        _:y1 rdf:rest wine:TableWine.
        wine:Bordeaux rdf:type owl:Class.}

?- subclassOf(?X, Wine)
// to list all subconcepts of Wine

    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX wine:
<http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
        SELECT ?X
        WHERE { ?X rdfs:subclassOf wine:Wine.}

?- subclassOf(Bordeaux, ?X), subclassOf(?X,Wine),
subclassOf(?X,?Y).

    PREFIX rdfs:http://www.w3.org/2000/01/rdf-schema#..
    ..
    PREFIX wine:
<http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
SELECT ?X ?Y
```

¹<http://www.w3.org/TR/owl-semantics/mapping.html>



```
WHERE {  
    wine:Bordeaux rdfs:subClassOf ?X.  
    ?X rdfs:subClassOf wine:Wine.  
    ?X rdfs:subClassOf ?Y.  
    ?Y rdf:type owl:Class.}
```

A.2 DIGPION

The procedure for setting up LarkC with a DIG reasoner:

- Check out the LarkC platform itself with the DIG reasoner plugin at the following SourceForge site:

```
https://larkc.svn.sourceforge.net/svnroot/larkc/trunk
```

- The LarkC DIG plug-in requires an external DIG reasoner, like Racer, FACT++, KOAN2, Pellet. Before starting the test, make sure that the external DIG reasoner has been installed at your computer (i.e., localhost) and is running at a known port. The java program DIGReasonerTest.java at the DIG plug-in source directory provides several typical examples how an external DIG reasoner can be called to reason with ontologies at the LarkC platform.

Before executing the test program, you can change the following setting in the program:

```
String hostname= "localhost";  
int port = 8080; //default port for racer  
String path = "/";
```

You can declare the ontology data by the following setting:
The ontology data can be claimed by a code, like this:

```
String ontologyFileName = "http://www.cs.vu.nl/~huang/  
larkc/ontology/wine.rdf";
```

or

```
String ontologyFileName = "file:///E:/larkc/ontology/wine.rdf";
```

if the ontology data is located at the local harddisk.

Using the following test utility to post the query to the external DIG reasoner and get the answer in the test program:

```
ReasonerTest(ontologyFileName, query32,  
hostname, port, path);
```

To claim an external DIG reasoner:



```
DIGReasoner reasoner = new DIGReasoner();  
reasoner.hostname = hostname;  
reasoner.port = port;  
reasoner.path = path;
```

To conduct a reasoning task:

```
BooleanInformationSet answer = reasoner.sparqlAsk(sparqlQuery,  
graph, contract, context);
```

Ignoring the contract and the context for the time being.

To test the DIG plug-in with PION, do the following:

1. Launch an external DL reasoner (like Racer) at the port 8000 of the localhost;
2. Launch the external PION at the port 8001 of the localhost;
3. Change the setting of the external DIG server by changing the following line at the test program: `int port = 8001`
4. Claim the ontology url and the query;
5. Launch the test utility program in the LarKC platform: `ReasonerTest(ontologyFileName, query, hostname, port, path);`

A query example:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>  
PREFIX ex: <http://wasp.cs.vu.nl/larkc/ontology/ex#>  
PREFIX owl: <http://www.w3.org/2002/07/owl#>  
ASK  
WHERE { ex:themadcow rdf:type ex:vegetarian.}
```

You can see that PION can return a meaningful answer 'false'. Compare it with that from a standard DL reasoner. You can see that when querying an inconsistent ontology, the standard DL reasoner always returns an error message, like this:

```
<responses xmlns="http://dl.kr.org/dig/2003/02/lang"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://dl.kr.org/dig/2003/02/lang  
http://dl-web.man.ac.uk/dig/2003/02/dig.xsd">  
<error id="http://wasp.cs.vu.nl/larkc/ontology/ex#themadcow  
http://www.w3.org/1999/02/22-rdf-syntax-ns#type  
http://wasp.cs.vu.nl/larkc/ontolog/ex#vegetarian"  
message="ABox http://dl.kr.org/dig/kb-1048 is incoherent."/>  
</responses>
```

You can use the PION TestBed page [piontest2.htm](#) to select different strategies for reasoning with inconsistent ontologies by PION: selection functions (syntactic relevance, concept syntactic relevance, or semantic relevance by google distances), over-determined processing methods (first maximal consistent set, or path pruning with Google distances), extension strategies (linear extension or k-extension), which is shown in Figure A.1.

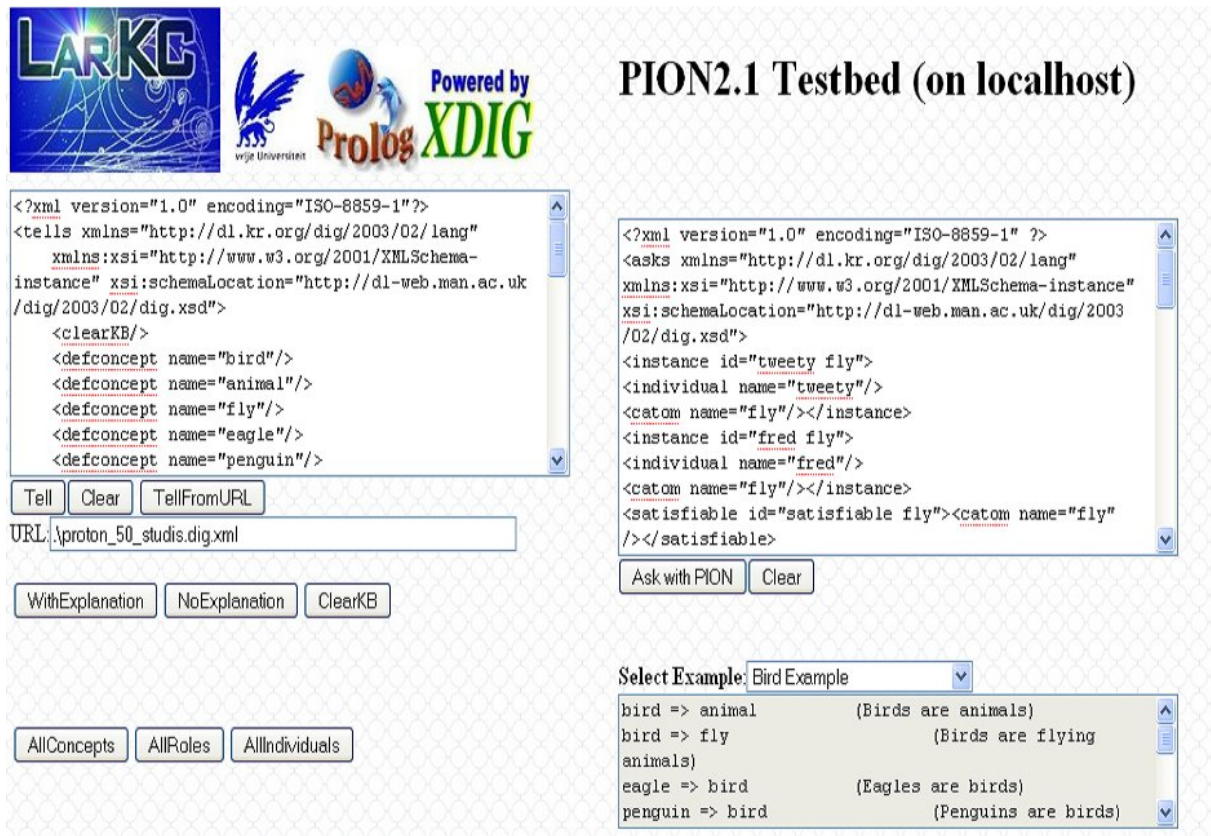


Figure A.1: PION TestBed

A.3 SimplePION

The program *SimplePIONTest.java* is an example how to call the SimplePION plug-in. The method *ReasonerTest(OntologyFileName, Query)* of *SimplePIONTest* is an utility to get the reasoning answer of the reasoner *simplePION* by providing an OWL ontology which is located at a URI and a query which is specified as a string.

The reasoner *SimplePION* relies on a standard OWLAPI reasoner which is supported by a Pellet reasoner. Thus, in order to call a *SimplePION* reasoner, make sure that the library of *OWLAPIreasoner* and *Pellet* reasoner is available at the classpath.

A.4 PIONwithStopRules

Using the reasoner *PIONwithStopRules* is similar with that of the reasoner *SimplePION*. The program *PIONwithStopRules.java* is an example which shows how to call the *PIONwithStopRules* plug-in in java. The method *ReasonerTest(OntologyFileName, Query)* of *PIONwithStopRulesTest* is an utility to get the reasoning answer of the reasoner *PIONwithStopRule*.

The parameter *seletedStopRule* of the reasoner plug-in allows us to decide which stop rule should be used for the stopping checking. The following java code shows how to make the setting of *selectedStopRule* with the time cost checking rule:

```
PIONwithStopRules reasoner = new PIONwithStopRules();
```



```
reasoner.seletedStopRule="TimeCostCheckingRule";
```

In order to use the cardinality checking rule, the string =”TimeCostCheckingRule” should be replaced with the string ”CardinalityCheckingRule”.

A.5 PIONWorkflow

The PIONWorkflow can be launched from the PION decider which calls two selectors, *SelectOntologyBasedOnQuery* for selecting a subontology which is relevant to the query and *SelectOntologyFromOntology* for selecting a subontology from an ontology, and the the reasoner *BasicOWLAPIReasoner* which provides the standard reasoning over the selected sub-ontology.

The program *PIONWorkflowTest.java* is an exmaple how to build a PION workflow and make the test. The method *PIONWorkflowProcess(OntologyFileName, Query)* of *PIONWorkflowTest* is an utility to get the reasoning answer of the reasoning workflow *PIONWorkflow*. The *OntologyFileName* can be specified as a string like this:

```
String ontologyFileName= "http://www.cs.vu.nl/~huang/larkc/ontology/mad_cows.owl";
```

The query can be specified as a string like this:

```
String query1 ="PREFIX rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"+  
"PREFIX rdfs:      <http://www.w3.org/2000/01/rdf-schema#>"+  
"PREFIX kb:       <http://cohse.semanticweb.org/ontologies/people#>" +  
"PREFIX owl:    <http://www.w3.org/2002/07/owl#>" +  
"PREFIX sparql1: <http://pellet.owldl.com/ns/sdle#>"+  
"ASK {"+  
"kb:madcow rdfs:subClassOf kb:vegetarian.}";
```