EU-IST Integrated Project (IP) IST-2003-506826 SEKT

SEKT: Semantically Enabled Knowledge Technologies

An Extended DIG Description Logic Interface for Prolog

Zhisheng Huang and Cees Visser
(Vrije Universiteit Amsterdam)

with contributions from:

Abstract.
EU-IST Integrated Project (IP) IST-2003-506826 SEKT
Deliverable D3.4.1.2 (WP3.4)

This document presents a DIG description logic interface extension for high-level programming languages like Prolog. The extension introduces the notion of an intermediate server, which provides both client side and server side processing facilities, i.e. intermediate Prolog-based extended description logic servers can call a standard DL reasoner that supports the DIG interface, in order to provide regular DL services to client applications. In addition, such an intermediate server can also act as an extended DL reasoner, which provides additional reasoning and processing capabilities to client applications.

Currently, we have implemented the extended DIG DL interface as an SWI-Prolog package. This document describes the extended framework and discusses several examples to show how it can be used to develop hybrid Prolog / DL reasoners.

Keywords: Description Logics, DIG interface, Logic Programming, Ontology Management

Document Id.	SEKT/2004/D3.4.1.2/v0.5
Project	SEKT EU-IST-2003-506826
Date	July 5, 2004
Distribution	public

Copyright © 2004 Department of Artificial Intelligence, Vrije Universiteit Amsterdam

SEKT Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-2003-506826.

British Telecommunications plc.
Orion 5/12, Adastral Park
Ipswich IP5 3RE
UK
Tel: +44 1473 609583, Fax: +44 1473 609832
Contactperson: John Davies
E-mail: john.nj.davies@bt.com

Jozef Stefan Institute
Jamova 39
1000 Ljubljana
Slovenia
Tel: +386 1 4773 778, Fax: +386 1 4251 038
Contactperson: Marko Grobelnik
E-mail: marko.grobelnik@ijs.si

University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1891, Fax: +44 114 222 1810
Contactperson: Hamish Cunningham
E-mail: hamish@dcs.shef.ac.uk

Intelligent Software Components S.A.
Francisca Delgado, 11 - 2
28108 Alcobendas
Madrid
Spain
Tel: +34 913 349 797, Fax: +49 34 913 349 799
Contactperson: Richard Benjamins
E-mail: rbenjamins@isoco.com

Ontoprise GmbH
Amalienbadstr. 36
76227 Karlsruhe
Germany
Tel: +49 721 50980912, Fax: +49 721 50980911
Contactperson: Hans-Peter Schnurr
E-mail: schnurr@ontoprise.de

Vrije Universiteit Amsterdam (VUA)
Department of Computer Sciences
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
Tel: +31 20 444 7731, Fax: +31 84 221 4294
Contactperson: Frank van Harmelen
E-mail: frank.van.harmelen@cs.vu.nl

Empolis GmbH
Europaallee 10
67657 Kaiserslautern
Germany
Tel: +49 631 303 5540, Fax: +49 631 303 5507
Contactperson: Ralph Traphöner
E-mail: ralph.traphoener@empolis.com

University of Karlsruhe, Institute AIFB
Englerstr. 28
D-76128 Karlsruhe
Germany
Tel: +49 721 608 6592, Fax: +49 721 608 6580
Contactperson: York Sure
E-mail: sure@aifb.uni-karlsruhe.de

University of Innsbruck
Institute of Computer Science
Techikerstraße 13
6020 Innsbruck
Austria
Tel: +43 512 507 6475, Fax: +43 512 507 9872
Contactperson: Jos de Bruijn
E-mail: jos.de-bruijn@deri.ie

Kea-pro GmbH
Tal
6464 Springen
Switzerland
Tel: +41 41 879 00, Fax: 41 41 879 00 13
Contactperson: Tom Bösser
E-mail: tb@keapro.net

Sirma AI EOOD (Ltd.)
135 Tsarigradsko Shose
Sofia 1784
Bulgaria
Tel: +359 2 9768, Fax: +359 2 9768 311
Contactperson: Atanas Kiryakov
E-mail: naso@sirma.bg

Universitat Autonoma de Barcelona
Edifici B, Campus de la UAB
08193 Bellaterra (Cerdanyola del Vallès)
Barcelona
Spain
Tel: +34 93 581 22 35, Fax: +34 93 581 29 88
Contactperson: Pompeu Casanovas Romeu
E-mail: pompeu.casanovasquab.es

Changes

Version	Date	Author	Changes
0.1	16.6.04	Zhisheng Huang	Creation
0.2	24.6.04	Zhisheng Huang	First Draft
0.3	26.6.04	Cees Visser	Second Draft
0.4	1.7.04	Zhisheng Huang	Some Changes
0.5	5.7.04	Zhisheng Huang	Changes based on new libraries

Executive Summary

The DIG description logic interface is a convenient high-level interface for DL reasoners. The interface is supported by most DL reasoners and easily allows for the construction of reusable software components.

Logic programming languages like Prolog are popular in AI. In this document we investigate how to enhance Prolog with DL reasoning support, to be used in several Semantic Web contexts. In particular we discuss an extended DIG Description Logic interface which defines both client side and server side processing functionalities for an intermediate server.

As a regular DIG DL client, intermediate Prolog-based servers can call external DL reasoners that support the DIG interface. In addition, from a client application point of view, an intermediate server can effectively act as an extended DL reasoner, which can be used to provide additional reasoning facilities.

We have currently implemented an extended DIG DL interface in SWI-Prolog and will give a detailed description of its functionality. Several examples will demonstrate how this extended DIG framework can be used to develop more powerful hybrid reasoning and processing components.

Contents

Chapter 1

Introduction

The DIG description logic interface [2], DIG interface for short, which is defined by the Description Logic Implementation Group (DIG)[1], provides a convenient high-level interface for DL reasoners. Many DL reasoners support the DIG interface and therefore more easily allow for the construction of highly portable and reusable components or extensions.

Logic programming languages in general, and Prolog in particular, are popular programming languages in AI, since they incorporate a high-level inference engine in their runtime system. Therefore, it is worthwhile to investigate the enhancement of a language like Prolog with DL reasoning facilities, to be used in for example Semantic Web contexts [5, 6, 7, 9].

In this document, we describe a DIG DL interface extension that, from a client application point of view, defines both DL reasoner services and special purpose services as provided by an intermediate extended description logic server ; as a regular DIG client, an intermediate server can call an external DL reasoner which supports the DIG interface.

The extended description logic interface has been implemented as a package for SWI-Prolog, a popular free Prolog system[2]. We will describe the extended description logic interface for SWI-Prolog in detail. Furthermore, we discuss several examples to show how additional services can be realized by using this interface package.

This document is organized as follows: Chapter 2 presents the design of the extended description logic interface. Chapter 3 describes the interface libraries in detail. Chapter 4 discusses several examples that illustrate how these libraries can be used. Chapter 5 concludes the document.

---

[1] http://dl.kr.org/dig/

[2] http://www.swi-prolog.org

Chapter 2

The Prolog Extended Description Logic Interface

2.1 The DIG DL Interface

The DIG interface is defined as a simple API for a general description logic system[2]. It uses a similar mechanism as SOAP (Simple Object Access Protocol), which has XML-based messaging protocols on top of HTTP.

Clients of a DL reasoner communicate through the use of HTTP POST requests. The body of the request is an XML encoded message which corresponds to the DL concept language. The DIG concept language is a description logic that includes the standard boolean concept operators, universal and existential restrictions, and other issues. A TELL request is used to assert DL statements in the knowledge base of the DL reasoner. An ASK request is used to perform knowledge base queries. In addition, management requests are used to maintain the knowledge base of DL reasoners or to obtain particular information of the system, like a reasoner identification. See [2] for more DIG description logic interface details.

2.2 General Considerations

The Prolog extended DIG interface libraries may be used to build DL reasoners that have additional reasoning capabilities, like reasoning about possible inconsistent ontologies, a working task in the SEKT project[4]. It is not necessary for extended Prolog-based DL reasoning systems to incorporate their own DL reasoning component; several well-known DL reasoners exist (e.g. Racer[3]) and an extended DL reasoner will access an existing external reasoner via its DIG interface.

In general, extended DL reasoners should be able to serve as a regular DL reasoner via

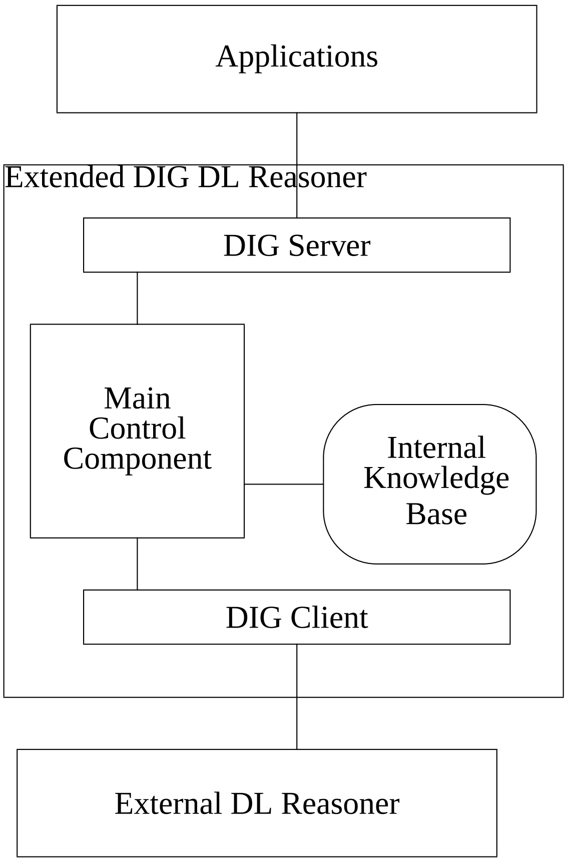

Figure 2.1: Architecture

their corresponding DIG description logic interface. Moreover, they will provide particular supplementary reasoning facilities. An intermediate extended DIG server can make systems independent of particular application specific characteristics, which significantly improves the reusability and applicability of software components; a highly decoupled infrastructure is usually beneficial for the construction of domain-specific services.

2.3 Architecture Overview

The general architecture of a Prolog-based extended DIG reasoner is shown in Figure 2.1. It consists of the following components:

- **DIG Server**: The DIG server deals with requests from ontology applications. It supports the extended DIG interface, i.e. it not only supports standard DIG/DL requests, like ’tell’ and ’ask’, but also additional processing features, like the specification or modification of selection functions, etc.

- **Main Control Component**: The main control component implements the general processing framework, like query analysis, query pre-processing, and the extension strategy, by calling the selection function and interacting with the ontology repositories.

- **DIG Client**: The regular DIG client layer calls the external DL reasoner in order to access the standard DL reasoning capabilities.

- **Internal KB**: The internal knowledge base is used to store DL statements locally for the Prolog/DL reasoner. More exactly, it consists of a set of $dig\_db\_element(+ID, ?Element)$ facts where ID specifies additional information, say version information, about the element list. These facts are used for further processing when the reasoner receives an ASK request.

Chapter 3

Prolog Extended DL Interface Libraries

The Prolog Extended DL package consists of the following libraries: dig_client, dig_server, dig_process, dig_db, and dig_client_setting. SWI-Prolog requires this package to be installed in the 'library/dig' directory.

3.1 library(dig/dig_client)

The library(dig/dig_client) provides the mechanism to call an external DL reasoner, i.e. DIG DL servers on remote hosts. It defines the following predicates:

- **dig_post**(+Data, -Reply, +Options)
  post the data to the default external DIG server with $Options$ that are permitted by the HTTP POST request. The data format can have one of the following forms:
  - file(FileName)
    data is provided by file *FileName*;
  - text(XMLText)
    data is represented as XML-encoded text;
  - elements(Elements)
    data is represented as a list of XML-parsed elements, see predicate load_xml_file of the SWI-Prolog SGML/XML parser package[8] for details. An XML-parsed element list is the main data format in the Prolog extended DL interface libraries.
    The reply *Reply* from a DIG server has the form anwer(Header, Elements) where *Header* is the header of the response and *Elements* an XML element list which corresponds to the body of the DIG server response.
- **dig_post**(+URL, +Data, -Reply, +Options)
  similar to the predicate dig_post/3, however, with the URL of the external DIG

server.

- **dig_post**(+Protocol, +Host, +Port, +Path, +Data, -Reply, +Options) similar to the predicate dig_post/4, however, the URL is explicitly specified by means of the parameters *Protocol*, *Host*, *Port* and *Path*.
- **dig_tell**(+Elements, -Answer, +Options)
  post the list of *Elements* with the tag 'tells' to the default DIG server.
- **dig_ask**(+Elements, -Answer, +Options)
  post the list of *Elements* with the tag 'asks' to the default DIG server.

3.2 library(dig/dig_server)

The library library(dig/dig_server) provides a mechanism to build a Prolog DL system which supports the extended DL interface. It defines the following predicates:

- **dig_server**(+Request)
  process a client's *Request*. It serves as the main entry point for the server, which is launched by an http_server process. Prolog extended DIG server developers have to define their own predicate **my_dig_server_processing**(+Data, -Answer, +Options) in order to handle the corresponding *Data* (i.e. the body of a request without a header) and *Answer*. See Chapter 4 for examples how to define a predicate like **my_dig_server_processing**.
- **dig_standard_response**(+Status, +ID, -Answer)
  provide a standard answer for $Status$ and request ID. The *Status* can be one of ok, $true$, or $false$.
  As an extension of the request support, the Prolog extended DIG server library not only supports XML-encoded request data which is posted with content type 'text/xml', but also supports the content type 'application/x-www-form-urlencoded'. The latter is used to post DIG interface data from an HTML form. A general requirement for the latter is that the posted data must be identified by 'dig_xml_data'.

  The following predicates can be used to get or set server characteristics, like the server's ID or the port number:
- **dig_server_port**(-Port)
- **dig_server_id**(-DIGServerID)
- **set_dig_server_port**(+Port)

- **set_dig_server_id**(+DIGServerID)

A Prolog extended DIG server can be launched from a Prolog program by means of:

```
:- http_server(dig_server,[port(8001)]).
```

3.3 library(dig/dig_process)

The library(dig/dig_process) provides the main predicates to process DIG messages. It defines the following predicates:

- **dig_add_elements**(+Elements, +Type, -ElementsWithType)
  add the tag *Type* to *Elements*, resulting in a list of *ElementsWithType*.
- **xml_elements**(+RawText, -XMLElements, -Header)
  translate a raw text reply from the server into a body with a list of elements and a text header.
- **elements_xmltext**(+XMLElements, -XMLText)
  translate a list of XMLElements to an XML-encoded text.
- **dig_requestdata_analysis**(+RequestData, -Data, -Type)
  get the $Data$ body from $RequestData$ with type $Type$, where $Type$ can be one of : $asks$, $tells$, $getIdentifier$, etc.
- **dig_data_analysis**(+Data, +Type, -Element, -OtherData)
  select an $Element$ of type $Type$ (e.g. 'satisfiable') from $Data$, resulting in $OtherData$ without $Element$.
- **dig_response_analysis**(+Response, ?Type, -Element, -OtherData)
  select an $Element$ of type $Type$ (e.g. 'true' or 'false') from the element list $Response$ which is usually defined by an $answer(Header, Response)$ from the external DL reasoner, resulting in $OtherData$ without $Element$.
- **dig_xmlns**(XMLNameSpace): get the DIG XML namespace list
- **dig_tmp_working_file**(Type, FileName) specifies the filename of a temporary file.

3.4 library(dig/dig_db)

The library(dig/dig_db) provides facilities to maintain the internal knowledge base of the extended Prolog DL system. It defines the following predicates:

- **dig_assert_data**(+ID, +Data)
  assert a data statement into the knowledge base with identifier ID. The default ID is *general*.

- **dig_db_element**(+ID, ?Element)
  check or get an *Element* from the knowledge base ID.

- **dig_db_post**(+ID, +AdditionalOptions, -Answer, +Options)
  post the entire knowledge base ID and $Options$ with the $AdditionalOptions$, where $AdditionalOptions$ is either $withClearKB$ or $withoutClearKB$.

- **dig_clear_data**(+ID)
  clear the knowledge base ID.

3.5 library(dig/dig_client_setting)

The library(dig/dig_client_setting) can be used for the settings of the corresponding external DL server, i.e., the DIG client. It consists of the following predicates:

- **dig_host**(-Host): get the $Host$ of the external DIG server
- **dig_port**(-Port): get the $Port$ of the external DIG server
- **dig_path**(-Path): get the $Path$ of the external DIG server
- **set_dig_host**(-Host): set the $Host$ of the external DIG server
- **set_dig_port**(-Port): set the $Port$ of the external DIG server
- **set_dig_path**(-Path): set the $Path$ of the external DIG server
- **dig_url**(-URL): get the default URL of the external DIG server.

3.6 Implementation

The Prolog extended DIG interface has been implemented in SWI-Prolog[1]. SWI-Prolog is a free software Prolog compiler. Being free, small and mostly standard compliant, SWI-Prolog has become very popular for education and research. The SWI-Prolog extended DIG library packages are available from http://wasp.cs.vu.nl/sekt/dig.

---

[1] http://www.swi-prolog.org

Chapter 4

Examples

In this chapter we will show a number of examples that demonstrate how the extended DIG libraries can be used to build Prolog/DL reasoners according to different strategies.

4.1 Forwarding Strategy

This is the most straightforward example of a Prolog DL reasoner; the Prolog DL reasoner forwards each client request to the external DL reasoner and each reply of the external DL reasoner is sent back to the client application.

The strategy can be defined by the predicate ’my_dig_server_processing’ as illustrated in the following program:

```
:-use_module(library('dig/dig_client')).
:-use_module(library('dig/dig_server')).
:-use_module(library('http/thread_httpd')).

my_dig_server_processing(RequestData, Answer, Options):-
        dig_post(RequestData, Answer, Options).

:- dig_server_port(Port),
   http_server(dig_server, [port(Port)]).
```

The message flow of this forward strategy is shown in Figure 4.1.

4.2 Accumulation Strategy

In the accumulation strategy, the Prolog DL reasoner accumulates all statements from TELL requests in internal knowledge bases until an ASK request is received, after which

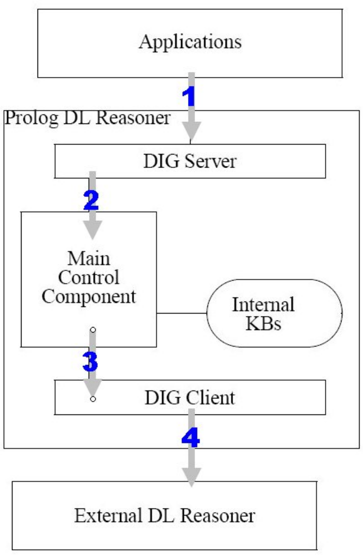

Figure 4.1: extended DIG Message Forwarding

the Prolog DL reasoner 'tells' the external DL reasoner the accumulated data, and performs the corresponding external DL reasoner query.

The strategy can be defined by means of the predicate 'my_dig_server_processing' as shown in the following Prolog program:

```
:-use_module(library('dig/dig_client')).
:-use_module(library('dig/dig_server')).
:-use_module(library('dig/dig_process')).
:-use_module(library('dig/dig_db')).
:-use_module(library('http/thread_httpd')).

%deal with the tell request
my_dig_server_processing(RequestData, Answer, [connection(close)]):-
       dig_requestdata_analysis(RequestData, Data, Type),
       Type=tells,
       !,
       dig_assert_data(general, Data),
       dig_standard_response(ok, _ID, Answer).


%deal with the ask request
my_dig_server_processing(RequestData, Answer, _Options):-
       dig_requestdata_analysis(RequestData, _Data, Type),
       Type=asks,
       !,
       dig_db_post(general, withClearKB, _Answer, [connection(close)]),
       dig_post(RequestData, Answer, [connection(close)]).

%deal with other request
my_dig_server_processing(RequestData, Answer, Options):-
```

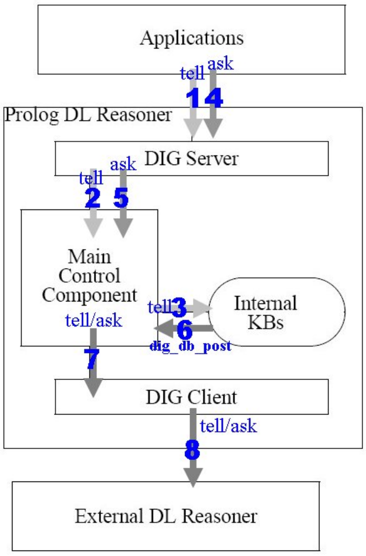

Figure 4.2: Accumulation Strategy

```
       dig_post(RequestData, Answer, Options).

:- dig_server_port(Port), http_server(dig_server,[port(Port)]).
```

The message flow that corresponds to this strategy is shown in Figure 4.2.

4.3 Query Processing Examples

Usually an intermediate Prolog/DL reasoner wants to perform some pre-processing with respect to client application requests. For example, a Prolog DL reasoner may want to filter particular types of queries and ignore other request types. Alternatively, a Prolog DL reasoner may want to select queries with a particular ID for further processing. In the following, we will show several examples how queries can be pre-processed when using the Prolog DIG packages.

4.3.1 Query Type Processing

The next program fragment shows how a satisfiability query is selected in an ASK request.

```
:-use_module(library('dig/dig_client')).
:-use_module(library('dig/dig_server')).
:-use_module(library('dig/dig_process')).
:-use_module(library('dig/dig_db')).
```

```
:-use_module(library('http/thread_httpd')).

%deal with the ask request, select a satisfiability query only.
my_dig_server_processing(RequestData, Answer, _Options):-
       dig_requestdata_analysis(RequestData, Data, Type),
       Type=asks,
       dig_data_analysis(Data, satisfiable, E, _OtherData),
       dig_ask(elements([E]), Answer, [connection(close)]).

%handle other requests
my_dig_server_processing(RequestData, Answer, Options):-
      dig_post(RequestData, Answer, Options).

:- dig_server_port(Port),
    set_dig_server_id('SWI-Prolog XDIG Server (Satisfiability Queries Only)'),
    http_server(dig_server,[port(Port)]).
```

We use the predicate $dig\_data\_analysis(Data, satisfiable, E, \_OtherData)$ to obtain a satisfiability query E, after which we perform an ASK request on the selected element list E to the external DL reasoner via the DIG client. The predicate $set\_dig\_server\_id$ is used to set a description of the Prolog extended DIG server.

4.3.2 Query ID Processing

In the following example, the reasoner selects queries that contain an ID 'myquery' in the ASK request.

```
......

%select queries with ID 'myquery'
my_dig_server_processing(RequestData, Answer, _Options):-
       dig_requestdata_analysis(RequestData, Data, Type),
       Type=asks,
       E = element(_Type, [id='myquery'], _C),
       dig_data_analysis(Data, _, E,  _OtherData),
       dig_ask(elements([E]), Answer, [connection(close)]).

%deal with the error message
my_dig_server_processing(RequestData, Answer, _Options):-
      dig_requestdata_analysis(RequestData, Data, Type),
      Type=asks,
      Answer=answer(_, text('<error description="cannot find myquery"/>')).

%handle other requests
my_dig_server_processing(RequestData, Answer, Options):-
      dig_post(RequestData, Answer, Options).
```

We use $E = element(\_Type, [id =' myquery'], \_C)$ to specify the element with ID 'myquery' and use the predicate $dig\_data\_analysis(Data, \_, E, \_OtherData)$ to retrieve the corresponding element from the data. Upon failure, the error message 'cannot find myquery' is displayed.

4.4 Extended DIG Processing

We can use the Prolog DIG interface package to extend the capabilities of the standard DIG DL interface so that it can deal with additional operators (i.e. tags). For example, we can define a Prolog server which can deal with the operator 'entailment' in an ASK request. We know that entailment can always be transformed into satisfiability. The relationship between entailment and satisfiability is given by:

$$\Sigma \models \phi \text{ iff } \Sigma \cup \{\neg\phi\} \text{ is not satisfiable.}$$

Namely, a formula set Σ entails a formula ϕ if and only if the set $\Sigma \cup \{\neg\phi\}$ is not satisfiable.

We can define the predicate 'my_dig_server_processing' in a Prolog-based server as follows:

```
my_dig_server_processing(RequestData, Answer, _Options):-
       dig_requestdata_analysis(RequestData, Data, Type),

       % ASK request
       Type=asks,

       % get the entailment data
       dig_data_analysis(Data, entailment, E, _OtherData),

       % deal with the query ID
       (E = element(entailment, [], E1), ID='NIL' ;
        E = element(entailment, [id=ID], E1)),

       % construct the negation of the query (satisfiabilty check)
       dig_add_elements(E1, not, E2),
       dig_add_elements(E2, satisfiable, E3),

       % post the satsifiability query to the external DL reasoner
       dig_ask(elements(E3), Answer1, [connection(close)]),

       % analyse the answer, the answer for the entailment is
       % always the opposite of the answer for the satisfiability
       % check.
       Answer1 = answer(_Header, AnswerBody),
       dig_response_analysis(AnswerBody, Type1, _Element,_O),
       opposite(Type1, Type2),
       dig_standard_response(Type2, ID, Answer).
```

```
opposite(true, false).
opposite(false, true).
```

Chapter 5

Conclusions

In this document, we have defined an extended DIG Description Logic interface for Prolog, and outlined the corresponding architecture of this interface. The DIG interface extension defines both client side and server side processing for an intermediate server. Client applications as well as intermediate extended DIG servers can access external DL reasoners, and intermediate servers may provide additional reasoning capabilities.

In addition, several examples illustrate how the Prolog extended DIG package can be used to develop an infrastructure for hybrid Prolog/DL semantic web systems. The main objective of the presented approach was to re-use existing DL reasoners and to provide a modular architecture in order to be able to incorporate domain-specific processing facilities.

Bibliography

[1] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, Peter Patel-Schneider, (eds.), *The Description Logic Handbook Theory, Implementation and Applications*, Cambridge University Press, Cambridge, UK, 2003.

[2] Sean Bechhofer, Ralf Mller, and Peter Crowther. The DIG Description Logic Interface. In DL2003 International Workshop on Description Logics, Rome, September 2003.

[3] Volker Haarslev, and Ralf Möller, Description of the RACER System and its Applications, Proceedings of the International Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August 2001, pp. 132-141.

[4] Zhisheng Huang, Frank van Harmelen, Annette ten Teije, Perry Groot, and Cees Visser, Reasoning with Inconsistent Ontologies: a general framework, SEKT report D3.4.1.1, 2004.

[5] Maarten Menken, Prolog Sesame Client, VUA, 2003. http://www.cs.vu.nl/∼mrmenken/prologsesame

[6] Jan Wielemaker, SWI-Prolog/XPCE Semantic Web Library, http://www.swi-prolog.org/packages/semweb.html.

[7] Jan Wielemaker, The SWI-Prolog RDF parser, http://www.swi-prolog.org/packages/rdf2pl.html.

[8] Jan Wielemaker, The SWI-Prolog SGML/XML parser, http://www.swi-prolog.org/packages/sgml2pl.

[9] Jan Wielemaker, Guus Schreiber, Bob J. Wielinga, Prolog-Based Infrastructure for RDF: Scalability and Performance. International Semantic Web Conference 2003, pp. 644-658.