CORBA interface for a DL Classifier

Sean Bechhofer, Ian Horrocks and Sergio Tessaris

March 1999

Abstract

An update to the original CORBA specification document of February 1999. The specification has been altered slightly following the experience of building a prototype implementation. An IDL specification and DTD for concept expressions have been added.

1 Introduction

The rationale behind providing a Description Logic (DL) reasoner as a network resource via the OMG's Common Object Request Broker Architecture (CORBA) [1] is described in [3]. This document describes some initial experiments and a protoype implementation of the so-called "thick wrapper" which sits between the clients and the Lisp Server and provides basic reasoner functionality.

2 Architecture

The classifier is seen from the clients as an object in the CORBA namespace. This object provides methods for interacting with the classifier. The actual implementation separates the classifier interface seen by the clients, from the real reasoning engine (see Figure 1).

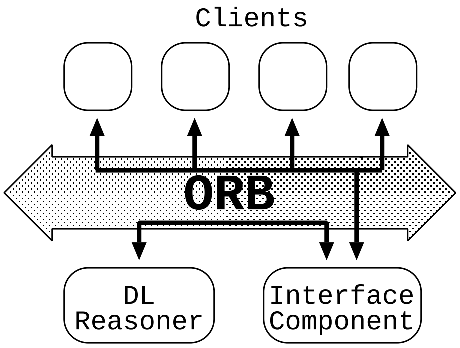

Figure 1: FaCT server Architecture

The FaCT server provides the functionalities to the CORBA clients, interacting with the FaCT classifier. The FaCT classifier engine is implemented in

Lisp and can only be accessed by a simple "Read-Eval-Print" interface (Lisp Server in Figure 1). The Lisp Server is completely hidden to the clients by the FaCT Server.

3 API description

The interface is specified using CORBA's Interface Definition Langauge (IDL). The IDL specifications are given in Appendix A. The various parts of the specification are described in more detail below.

3.1 Classifiers and Client Handlers

The Classifier is a networked resource that may be used by multiple clients. In order to control this, we introduce the notions of a classifier and client handlers. A classifier encapsulates the notion of the DL reasoner – for each incarnation of the reasoner, there will be a single classifier object. Each client that connects to the classifier is represented by a client handler. This allows us to distinguish between clients, and provide, for example, functionality that warns clients whether the knowledge base has been altered by somebody else. It also allows us to implement a simple transaction model which prevents multiple clients from attempting to alter the knowledge base simultaneously. The ask/tell functionality of the reasoner is provided through the client handlers.

3.2 Data types

The basic data types used by the interface are concept, role, c_names and r_names. Whenever a method returns a list of concept (role) names, the actual value is not just a set of names but a sequence of disjoint sets of names. Each set represents an equivalence class of concept (role) names.

concept	::=	string
	|	*top_concept* | *bottom_concept*
	|	(**and** concept$^+$)
	|	(**or** concept$^+$)
	|	(**not** concept)
	|	(**some** role concept)
	|	(**all** role concept)
	|	(**at_least** number role concept)
	|	(**at_most** number role concept)
role	::=	string
	|	*top_role* | *bottom_role*
	|	(**inverse** role)

As discussed in [3], concepts and roles are not passed as objects. The descriptions themselves have no associated functionality other than for determining what their structure is. Implementing them as objects may result in increased overheads as many message sends may be required to determine that structure. The descriptions are thus treated simply as data. Ideally, this data format would be described using IDL. However, IDL does not appear rich enough to define the kinds of recursive data structures that we'd require in order to represent the descriptions. The solution adopted is to use eXtended Markup Language (XML) [2] as the format for representing concept descriptions. A Document Type Definition (DTD) states exactly which tags should appear in concept descriptions. A prototype DTD for concept descriptions is given in Appendix B. Table 1 shows how some simple expressions with corresponding markup.

Standard notation	XML markup
$\top$	<TOP/>
$P_1 \sqcap (\neg P_2)$	<AND> <PRIMITIVE NAME="P1"/> <NOT> <PRIMITIVE NAME="P2"/> </NOT> </AND>
$\exists R.P$	<EXISTS> <PRIMITIVEROLE NAME="R"/> <PRIMITIVE NAME="P"/> </EXISTS>

Table 1: XML concept descriptions

The DTD can easily be extended to include basic assertions (an example is shown in Appendix C) – XML is thus a suitable candidate as an interchange format for knowledge bases.

Concept Descriptions are thus strings, containing XML markup as described above. Concept and role names are strings: the c_names and r_names types are defined as sequence of sequences of strings, representing collections of equivalence classes as described above. The concept and role names are also expressed in XML markup. For example, the concept name C1 would be expressed as <PRIMITIVE NAME = "C1"/>.

```
typedef string ConceptDescription;
typedef string RoleDescription;

typedef string c_name;
typedef string r_name;

typedef sequence<c_name> c_list;
typedef sequence<c_list> c_names;
typedef sequence<sequence<r_name> > r_names;
```

The description of a position in the taxonomy is given by a structure with three parts representing the supers, subs and equivalent concepts.

```
struct taxonomy_node { c_names supers, subs c_list equivs; };
```

3.3 Exceptions

The following exceptions are defined. Each exception has an code and a string which may impart further information about the exception or error.

CommunicationException A general exception that signifies that something has gone wrong.

TransactionRequiredException The requested operation can be performed in the context of a transaction only. The same exception is raised when a client ask for a transaction opening while a transaction is already opened. The return code provides more information about what has occured.

OpUnimplementedException The requested operation is not implemented by the server.

ExprErrorException A concept or role expression in the arguments of the operation is syntactically wrong. This exception covers both the cases of wrong syntax and unimplemented concept or role operators (i.e. role value map). In the return value of the exception there should be the description of the wrong part of the expression.

KBModifiedException This is not an error condition but just a warning that the knowledge base has been modified since the last access of the client. Since this condition depends on the client, the server should keep track of the clients which already received the signal after a modification of the knowledge base. Note that because of the way that exceptions are raised, the client will have to reask the question if it wishes to proceed after receiving this exception.

3.4 Classifier

3.4.1 Identification

Basic functionality is provided to help in identifying servers. This is primarily for use in logging and reporting errors.

---

string **identifier** ( )

Description: Returns an identifier which provides some information to the client about the classifier.

3.4.2 Client creation

ClientHandler **newHandler** (in string id)

Description: This operation is a request for a new client handler. It takes a single argument which is a string identifying the client which can be used for logging purposes. The operation may raise a **CommunicationException** if the classifier is unable to provide a new handler.

3.5 Client Handler

The client handler encapsulates most of the reasoning functionality and provides a simple transaction model to control knowledge base updates.

3.5.1 Identification

Again, basic functionality is provided to help in identifying clients for use in logging and reporting errors.

---

string **identifier** ( )

Description: Returns the identifier that the server uses to refer to this client.

3.5.2 Knowledge base control

A simple transaction model is provided which ensures that only one client can update the knowledge base at any one time.

---

boolean **begin_transaction** ( )

Description: The function is a request for opening a transaction on the knowledge base. It returns the value **TRUE** if the knowledge base has been successfully allocated to requesting client, **FALSE** otherwise. There is no waiting queue, so if the client fails to initiate the transaction it has to retry later by itself.

---

boolean **reset_transaction** ( )

Description: The function is a request for restoring the current open transaction to the initial state. All the requested modifications are discarded as if they were not asserted. It returns the value **TRUE** only if there is a transaction opened by the requesting client. If the request succeed the transaction stays opened.

---

boolean **end_transaction** ( )

Description: The function is a request for closing a currently open transaction. It returns the value **TRUE** only if there is a transaction opened by the requesting client, and all the modifications to the knowledge base requested during the transaction have been successfully commited. In case of failure, the knowledge base state is restored as it was before the beginning of the transaction.[1]

---

boolean **abort_transaction** ( )

Description: The function is a request for aborting the current open transaction. It returns the value **TRUE** only if there is a transaction opened by the requesting client. All the requested modification are discarded and the knowledge base state is restored as it was before the beginning of the transaction.

[1] Failure on the committing process.

boolean **in_transaction** ( )

Description: Returns the value **TRUE** if the requesting client has an open transaction, **FALSE** otherwise.

3.5.3 Knowledge base modification

Before any modification, the knowledge base must be locked opening a transaction. If any of the operations are called without an open transaction, an **TransactionRequiredException** will be raised. In addition, the operations can also raise an **OpUnimplementedException** or an **ExprErrorException**.

void **clear** ( ) *

Description: The function removes all the assertions from the knowledge base.

void **shoot** ( ) *

Description: The function registers the current KB state as a snapshot for subsequent rollback operations.

void **rollback** ( ) *

Description: The function restores the state of the knowledge base to the last stored snapshot. There is no verification of the identity of the client which took the last snapshot.

3.5.4 Adding assertions

void **defconcept** (**in** c_name) *

Description: Asserts the name as being a concept name.

void **defrole** (**in** r_name) *

Description: Asserts the name as being a role name.

void **implies_c** (**in** ConceptDescription, **in** ConceptDescription) *

Description: Asserts the first concept as being included in the second concept.

void **equal_c** (**in** ConceptDescription, **in** ConceptDescription) *

Description: Asserts the first concept as equal to the second concept.

void **implies_r** (**in** RoleDescription, **in** RoleDescription) *

Description: Asserts the first role as being included in the second role.

void **equal_r** (**in** RoleDescription, **in** RoleDescription) *

Description: Asserts the first role as equal to the second role.

void **transitive** (**in** r_name) *

Description: Assert role name as transitive.

void **functional** (**in** r_name) *

Description: Assert role name as functional.

string **tells** ( ) *

Description: Returns the pending tells from this client handler, i.e. those facts which have been given to the handler in the current transaction, but which have not yet been added to the knowledge base. The tells are returned in XML format (but not as a valid XML document).

string **allTells** ( ) *

Description: Returns an XML knowledge base which corresponds to the current state of the classifier (i.e. all the tells that it has been given). Note that the knowledge base returned may contain facts which other handlers have added. A complete and valid XML document is returned.

3.6 Querying the KB

If there is an open transaction, the knowledge base manager aswers to the queries with respect to the state of the knowledge base as it was before the beginning of the transaction. The only way for a client to ensure the invariance of the knowledge base, is by opening a transaction before any query. Otherwise any other client can modify the knowledge base between two subsequent queries.

3.6.1 Logic queries

boolean **satisfiable** (**in** ConceptDescription)

Description: Tests the satisfiability of the given concept with respect to the terminology.

---

boolean **subsumes** (**in** ConceptDescription, **in** ConceptDescription)

Description: Tests the subsumption between the two given concepts with respect to the terminology.

---

boolean **equivalent** (**in** ConceptDescription, **in** ConceptDescription)

Description: Tests the equivalence between the two given concepts with respect to the terminology.

3.6.2 Extra–logic queries

---

c_names **direct_supers_c** (**in** c_name)

Description: Returns the direct super–concepts of the given concept name with respect to the taxonomy.

---

c_names **all_supers_c** (**in** c_name)

Description: Returns all the super–concepts of the given concept name with respect to the taxonomy.

---

c_names **direct_subs_c** (**in** c_name)

Description: Returns the direct sub–concepts of the given concept name with respect to the taxonomy.

---

c_names **all_subs_c** (**in** c_name)

Description: Returns all the sub–concepts of the given concept name with respect to the taxonomy.

---

r_names **direct_supers_r** (**in** r_name)

Description: Returns the direct super–roles of the given role name with respect to the taxonomy.

---

r_names **all_supers_r** (**in** r_name)

Description: Returns all the super–roles of the given role name with respect to the taxonomy.

r_names **direct_subs_r** (**in** r_name)

Description: Returns the direct sub–roles of the given role name with respect to the taxonomy.

r_names **all_subs_r** (**in** r_name)

Description: Returns all the sub–roles of the given role name with respect to the taxonomy.

taxonomy_node **taxonomy_position** (**in** ConceptDescription)

Description: Returns the position in the taxonomy of the given concept expression without classifying it.

References

[1] The Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1998.

[2] Extensible markup language (XML) 1.0. W3C Recommendation TR REC-xml-19980210, February 1998. Editors T. Bray, J. Paoli, C. M. Sperberg-McQueen.

[3] S. Bechhofer, I. Horrocks, P. F. Patek-Schneider, S. Tessaris A Proposal for a Description Logic Interface, 1999.

A IDL Definitions

```
module img {
  module fact {
    typedef string ConceptDescription;
    typedef string RoleDescription;

    typedef string cName;
    typedef string rName;

    typedef sequence <cName> cEquivNames;
    typedef sequence <cEquivNames> cNames;

    typedef sequence <rName> rEquivNames;
    typedef sequence <rEquivNames> rNames;

    struct taxonomyNode {
      cNames supers;
      cNames subs;
      cEquivNames equivs;
    };

    exception CommunicationException {
      short code;
      string information;
    };

    exception TransactionRequiredException {
      short code;
      string information;
    };

    exception OpUnimplementedException {
      short code;
      string information;
    };

    exception ExprErrorException {
      short code;
      string information;
    };

    exception KBModifiedException {
      short code;
      string information;
    };

    interface ClientHandler {

      string identifier();

      void release();

      boolean begin_transaction();
```

```
boolean reset_transaction();
boolean end_transaction();
boolean abort_transaction();
boolean in_transaction();

void clear()
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

void defconcept(in cName nm)
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

void defrole(in rName nm)
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

void impliesC(in ConceptDescription c1, in ConceptDescription c2)
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

void equalC(in ConceptDescription c1, in ConceptDescription c2)
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

void impliesR(in RoleDescription r1, in RoleDescription r2)
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

void equalR(in RoleDescription r1, in RoleDescription r2)
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

void transitive(in rName rn)
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

void functional(in rName rn)
     raises (TransactionRequiredException,
             OpUnimplementedException,
             ExprErrorException);

string tells();

string allTells();
```

```
boolean satisfiable(in ConceptDescription c)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

boolean subsumes(in ConceptDescription c1, in ConceptDescription c2)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

boolean equivalent(in ConceptDescription c1, in ConceptDescription c2)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

cNames directSupersC(in cName cn)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

cNames allSupersC(in cName cn)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

cNames directSubsC(in cName cn)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

cNames allSubsC(in cName cn)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);


rNames directSupersR(in rName rn)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

rNames allSupersR(in rName rn)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

rNames directSubsR(in rName rn)
     raises (OpUnimplementedException,
             ExprErrorException,
             KBModifiedException);

rNames allSubsR(in rName rn)
     raises (OpUnimplementedException,
             ExprErrorException,
```

```
                  KBModifiedException);

      taxonomyNode taxonomyPosition(in ConceptDescription c)
           raises (OpUnimplementedException,
                   ExprErrorException,
                   KBModifiedException);
    };

    interface Classifier {
      ClientHandler newHandler(in string id) raises (CommunicationException);

      string identifier();

    };
  };
};
```

B DTD for Concept Descriptions

```
<!ELEMENT CONCEPT (PRIMITIVE|TOP|BOTTOM|AND|OR|NOT|SOME|ALL|ATMOST|ATLEAST)>

<!ELEMENT ROLE (PRIMROLE|INVROLE)>

<!ELEMENT PRIMITIVE EMPTY>
<!ATTLIST PRIMITIVE NAME CDATA #REQUIRED>

<!ELEMENT TOP EMPTY>

<!ELEMENT BOTTOM EMPTY>

<!ELEMENT AND (PRIMITIVE|TOP|BOTTOM|AND|OR|
               NOT|SOME|ALL|ATMOST|ATLEAST)+>

<!ELEMENT OR (PRIMITIVE|TOP|BOTTOM|AND|OR|
              NOT|SOME|ALL|ATMOST|ATLEAST)+>

<!ELEMENT NOT (PRIMITIVE|TOP|BOTTOM|AND|OR|
               NOT|SOME|ALL|ATMOST|ATLEAST)>

<!ELEMENT SOME ((PRIMROLE|INVROLE),
                (PRIMITIVE|TOP|BOTTOM|AND|OR|
                 NOT|SOME|ALL|ATMOST|ATLEAST))>

<!ELEMENT ALL ((PRIMROLE|INVROLE),
               (PRIMITIVE|TOP|BOTTOM|AND|OR|
                NOT|SOME|ALL|ATMOST|ATLEAST))>

<!ELEMENT ATLEAST ((PRIMROLE|INVROLE),
                   (PRIMITIVE|TOP|BOTTOM|AND|OR|
                    NOT|SOME|ALL|ATMOST|ATLEAST))>
<!ATTLIST ATLEAST NUM CDATA #REQUIRED>

<!ELEMENT ATMOST ((PRIMROLE|INVROLE),
                  (PRIMITIVE|TOP|BOTTOM|AND|OR|
                   NOT|SOME|ALL|ATMOST|ATLEAST))>

<!ATTLIST ATMOST NUM CDATA #REQUIRED>

<!ELEMENT PRIMROLE EMPTY>
<!ATTLIST PRIMROLE NAME CDATA #REQUIRED>

<!ELEMENT INVROLE (PRIMROLE|INVROLE)>
```

C DTD for Knowledge Bases

```
<!ELEMENT KNOWLEDGEBASE (DEFCONCEPT|DEFROLE|IMPLIESC|EQUALSC|
                         IMPLIESR|EQUALSR|TRANSITIVE|FUNCTIONAL)*>

<!ELEMENT DEFCONCEPT EMPTY>
<!ATTLIST DEFCONCEPT NAME CDATA #REQUIRED>

<!ELEMENT DEFROLE EMPTY>
<!ATTLIST DEFROLE NAME CDATA #REQUIRED>

<!ELEMENT IMPLIESC (CONCEPT, CONCEPT)>

<!ELEMENT EQUALC (CONCEPT, CONCEPT)>

<!ELEMENT IMPLIESR (ROLE, ROLE)>

<!ELEMENT EQUALR (ROLE, ROLE)>

<!ELEMENT TRANSITIVE (ROLE)>

<!ELEMENT FUNCTIONAL (ROLE)>
```

D Prototype Implementation

This section discusses the status of the current prototype implementation. Several classes are supplied which perform various tasks.

Please Note: This implementation is not complete. The system has not yet been thoroughly tested and will go wrong! No commitment to maintenance or bug fixing has been made. This is simply a demonstration that the approach can be made to work. However, comments are welcome.

D.1 ObjectBinder

A simple helper which binds an object in a naming service.

```
java img.util.ObjectBinder [options] URL
```

where `URL` points at an IOR. Available options are:

`-nsHost host`	Host running naming service.
`-nsPort port`	Port number of naming service.
`-name name`	The name to bind to.

D.2 BrowseCosNames

A simple helper which reports the names bound in a naming service.

```
java img.util.BrowseCosNames [options]
```

Options are:

`-nsHost host`	Host running naming service.
`-nsPort port`	Port number of naming service.

D.3 Server

The server class controls the creation of a **Classifier** object and its binding in a naming service.

```
java img.fact.server.Server [options]
```

Options are:

`-nw`	Don't run as a window.
`-logHandlers`	If run without a window, report the creation and release of client handlers.
`-logLock`	If run without a window, report transaction management events.
`-logClassifier`	If run without a window, report classifier events.
`-logFull`	If run without a window, report everything.
`-nsHost host`	Host running naming service.
`-nsPort port`	Port number of naming service.
`-lispName name`	The name that the lisp evaluator is bound to.
`-name name`	Name to bind the server to.

D.4 Compiler

A simple compiler which will compile an XML knowledge base is provided.

```
java img.fact.client.Compiler [options] filename
```

Options:

`-nsHost host`	Host running naming service.
`-nsPort port`	Port number of naming service.
`-serverName name`	The name that the server is bound to.
`-noClear`	Don't initialise the knowledge base before compiling (assertions are added).

D.5 Client

The client class opens a window which allows connection to a running server (through the creation of a **ClientHandler**.

```
java img.fact.client.LittleClient
```

D.6 Running the System

There are four pieces required to make everything work:

1. A CORBA Naming Service.
2. A Lisp Server (running FaCT).
3. A Server.
4. A Client.

Each of these could run on a different machine. In order to illustrate this, we illustrate how the system could be run using four different machines[2].

The server and clients require Java 1.2.

D.6.1 Naming Service

First, start the naming service. For the sake of this example, we'll assume the naming service is to run on a PC called potato.

```
potato>start tnameserv -ORBInitialPort 1050
```

This starts a nameserver on **potato**, running on port 1050[3].

[2]Five machines are actually used here as the binding of the lisp server object is performed by a machine not involved in anything else

[3]If no port is supplied, the service defaults to running on port 900)

D.6.2 Lisp Server

The Lisp server will currently run on a UNIX machine (e.g. `kiss`). The server uses a special version of lisp which has ILU bound in. This can be found in `/opt/imgapp1/software/ILU/<arch>/bin/ILUacl`.

Start lisp up:

```
kiss>/opt/imgapp1/software/ILU/sunos/bin/ILUacl
```

Lisp code for the server can be found in `/opt/imgapp1/software/ILU/eval-lisp/server.lisp`. Load this:

```
USER(1): (load "/opt/imgapp1/software/ILU/eval-lisp/server.lisp")
```

Start the server:

```
USER(2): (lisp-server:start-server "IOR.ref")
```

This will start up a server and places an IOR in the file `IOR.ref`. Make sure that this file is accessible from a URL, say `http://carrot.cs.man.ac.uk/fact/IOR.ref`

D.6.3 Binding the Lisp Server

Use the **ObjectBinder** to bind the lisp server to the Naming Service. Note that this binding could be done on a completely different machine to that which runs the lisp server or the naming service.

```
leek>java img.util.ObjectBinder -nsHost potato -nsPort 1050 -name
FaCTLisp http://carrot.cs.man.ac.uk/fact/IOR.ref
```

D.6.4 Starting the server

Now start the server.

```
cabbage>java img.server.Server -nsHost potato -nsPort 1050 -lispName
FaCTLisp -name FaCTServer
```

This will create a new **Classifier** object and bind it in the naming service with the given name.

D.6.5 Starting the Client

We can now start the client.

```
turnip>java img.fact.client.LittleClient
```

This brings up a client window. When **Connect** is selected, the client throws up a form requesting values which allow the client to find the server. In this particular example, the values used would be:

Host:	`potato`
Port:	`1050`
Server Name:	`FaCTServer`

The client name can be anything and is simply used for logging and identification purposes. The client asks the server for a new **ClientHandler** which it then uses to communicate with the server.

The client provides some simple commands for adding definitions to the knowledge base and compiling XML assertions. A simple graphical view of the concept hierarchy is also provided.

D.7 ORB Implementation

The prototype has been implemented using the Java 1.2 ORB implementation. This is a fairly lightweight implementation, but it offers sufficient functionality to be able to conduct some experiments. It's primarily intended for client implementations, but the server has been built using the classes generated by `idltojava` and the ORB implementation bundled in the JDK version 1.2.

D.8 Current Status and Limitations

The lightweight ORB functionality means that error handling is poor, particularly if things go wrong at the server side. A simple transaction mechanism is in place, but the **KBModifiedException** is never raised.

The DTD (`fact.dtd`) describing the XML format for concept descriptions and knowledge bases needs to be in the directory that the server and/or client are run from. Ideally, this should be picked up from some external source e.g. a URI.

There seem to be some occasional problems with the graph drawer when trying to draw big models.

D.9 Other Components Used

The implementation makes use of the following external components:

- Sun's JavaCC Compiler compiler.
- Sun's Project X XML parser and DOM implementation is used to parse and decode the XML data representations of concept descriptions. This is licensed for internal, noncommercial use. Other XML parsers are available and could easily be slotted in.
- VGJ (Visualizing Graphs with Java) from Auburn University is used for some simple rendering of graphs. This is available under GNU public license. This is not an integral part of the architecture, but is used to demonstrate the kinds of things that a client could do.