Implementation of an XML Validator in Racer Proxy for the DIG Interface

---

Henny Pusphita Chendawan

Matriculation Number: 29040

Student Project

Submitted in partial fulfilment of Master Program
in Information and Media Technologies

Supervised by

Prof. Dr. Ralf Möller

M.Sc. Atila Kaya

Software, Technology, and Systems (STS)

Technical University of Hamburg-Harburg

April 2004

Acknowledgements

First of all, I would like to thank **Prof. Dr. Ralf Möller** from Software, Technology, and Systems (STS), Technical University of Hamburg-Harburg, for providing this topic of Student Project and thus, giving me a chance to further discover about Description Logics, DIG Interface, and explore the RACER Systems and its Proxy.

I would also thank **Atila Kaya** as my project supervisor, for all his ideas, guidance, and encouragement throughout this project.

Lastly, I am grateful for all helpful suggestions and supports from my friends during my work.

Table of Contents

1. Introduction

The main effort of research in knowledge representation nowadays is providing theories and systems for expressing structured knowledge and for accessing and reasoning with it in a principled way. Description Logics (DL) is important powerful class of logic-based knowledge representation languages.

Description logic theory provides a firm foundation for a number of implementations of DL reasoners: computer programs that take description logic formulae as input, and deliver theorems about those formulae (such as whether they are logically consistent) as a result. One of such reasoners is RACER.

The DIG Interface describes how to access RACER via HTTP and XML. As DIG defines a simple XML encoding to be used over an HTTP interface to a DL reasoner, the DL engine must provide basic HTTP support which is able to parse and generate XML content. In interacting with a DIG reasoner, a DIG client initiates one or more XML-encoded requests by using HTTP POST command. The server will then respond with a combination of an appropriate HTTP response code and, where appropriate, the results of the action or actions similarly encoded in XML. In RACER system, this HTTP-XML support is one of the features provided by Racer Proxy.

However, the present Racer Proxy does not implement XML validation during its XML processing. Validation is the process where XML processor checks that an instance document meets the requirements of its defining XML schema. For now, Racer Proxy checks only the XML Document Entity for well-formedness without taking its validity into account.

This lack of validation process may become a problem since client's request and RACER response alike may not always conform to the DIG XML schema. In fact, RACER still generates error messages which do not conform to DIG, whereas RACER has actually implemented DIG in its system as a standard interface. The problem may lie within the RACER system, but to facilitate future improvement, there is a need for a protocol which finds and locates those validation errors.

1.1. Objectives

The main objective of this Student Project is to develop a validating XML processor as an extension for Racer Proxy program. This XML validator would be applied during HTTP connection between the client and RACER Server via Racer Proxy. Its function is to provide parsing and validation of both client's request and RACER Server's response based on the schema in DIG Interface.

The aforementioned XML validator would be written in a separate class and integrated into a related package in Racer Proxy along with other classes

1.2. Structure of the Work

What is schema and what is validation in XML? Those would be explained briefly in the chapter 2, including an overview of standard technologies in XML validation that will be used for writing the validator for Racer Proxy.

Subsequently, chapter 3 would give the basic picture of Description Logic and a more focused explanation of the DIG Description Logic Interface as the fundamental background behind this project to build an XML validator.

Briefly, RACER and Racer Proxy themselves would be described in chapter 4 about analysis and design for the new validator in Racer Proxy. Chosen technology and supporting tool to write the program would be mentioned as well.

Chapter 5 would describe the technical writing of the XML validator in Java and show how it will be integrated in the Racer Proxy program.

The last chapter would present summary of this Student Project, conclusion, and suggestions for possible future improvement for Racer Proxy.

2. XML Schema and Validation

2.1. DTD and XML Schema

A well-formed XML document is a document that conforms to the XML syntax rules, meaning that it has correct XML syntax. A valid XML document is a well-formed XML document which also conforms to the rules of a particular schema.

A schema is a definition of the valid syntax of an XML-based language (i.e., it defines the valid structure of the elements and attributes in an XML documents). This definition include what elements are (and are not) allowed at any point, what the attributes for any element may be, the number of occurrences of elements, etc. A schema language is a formal language for expressing schemas.

The most-widely used schema language is *Document Type Definition* (DTD). It lists a number of element names, which elements can appear in combination with other ones, what attributes are available for each element type, etc. A DTD uses a different syntax (namely Extended Backus Naur Form) from that used by XML documents.

Because of some limitation in DTD, including the fact that DTD itself does not use XML syntax, other schema languages have been developed. A popular alternative which tries to overcome deficiency in DTD is XML Schema. The *W3C XML Schema Definition Language* is an XML language for describing and constraining the content of XML documents. As the present recommendation of World Wide Web Consortium (W3C), XML Schemas provide a means for defining the structure, content and semantics of XML documents.

The ideas of XML Schemas involve these most central top-level constructs:

- a (global) **element declaration** associates an element name with a *type.*
- a ***complex* type definition** defines requirements for attributes, sub-elements, and character data in elements of that type.
- **attribute declarations**: describe which attributes that may or must appear.
- **element references**: describe which sub-elements that may or must appear, how many, and in which order.

- a ***simple* type definition** defines a set of strings to be used as attribute values or character data .

Two types or two elements cannot be defined with the same name, but an element declaration and a type definition may use the same name.

2.2. Validating XML Processor

An XML processor is a software module that reads XML documents to find out the structure and content of the XML document. It provides access to the XML document's content and structure to application programs. In the end, the processor is simply a bridge between the XML document and the application that will be using that document.

There are validating processors and non-validating processors. Validating XML processors must read and process the entire schema and all external parsed entities referenced in the document, and then, at user option, report all well-formedness and violations of the validity constraints expressed by the declarations in the schema.

This process of checking the conformance of XML document to a particular schema is called validation, and the document being validated is often called an instance document or application document. An element in an XML document is valid according to a given schema if the associated element type rules are satisfied. If all elements are valid, the whole document is called valid.

The validation process runs as follows: given an XML document and a schema, validating XML processor first checks for validity, i.e. that the document conforms to the schema requirements. If the document is valid, a normalized version is output: default attributes and elements are inserted, parsing information may be added, etc.

There are at least four levels of validation enabled by schema languages:

- The validation of the markup -- controlling the structure of a document.
- The validation of the content of individual leaf nodes (datatyping)
- The validation of integrity, i.e. of the links between nodes within a document or between documents.
- Any other tests (often called "business rules").

2.3. DOM, SAX, and JAXP

To work with XML in general-purpose programming languages we need to:

- parse XML documents into XML trees
- navigate through XML trees
- construct XML trees
- output XML trees as XML documents

Parsing is the process of reading a document and dissecting it into its elements that can then be analyzed. In XML, parsing is done by XML processor.

To allow programmers to access their information without having to write a parser themselves, a number of application programming interfaces (APIs) were created. These APIs make dealing with common XML tasks, such as parsing, easier.

The *Document Object Model* or DOM is an API specification being developed by W3C. It is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.

SAX is the *Simple API for XML*, originally a Java-only API. SAX was the first widely adopted API for XML in Java, and is a “de facto” standard. Like DOM, SAX gives access to the information stored in XML documents using any programming language (and a parser for that language). SAX and DOM APIs are both available for multiple languages (Java, C++, Perl, Python, etc.).

Although they serve the same purpose, DOM and SAX take different approaches in providing access to information. DOM creates a tree of nodes based on the structure and information in the XML document, and the information can be accessed by interacting with this tree of nodes. DOM gives access to the information stored in the XML document as a hierarchical object model.

With SAX, the parser tells the application what is in the document by notifying the application of a stream of parsing events. Application then processes those events to act on data. This makes SAX an event-based interface, which differs dramatically from the memory-intensive, tree-based DOM. Therefore, SAX is very useful when the document is large.

The *Java API for XML Processing* (JAXP) is the official API for XML processing from Sun. JAXP enables applications to parse, validate, and transform

XML documents through pure Java APIs. The reference implementation uses the high performance Java Project X as its default XML parser. However, the software's pluggable architecture allows any XML conformant parser to be used.

Although it says XML Processing, JAXP doesn't in fact provide any processing at all, but rather supplies a mechanism for returning SAX parsers and DOM documents.

There are two specific services provided by JAXP:

- A mechanism for plugging in various providers supporting DOM, SAX and XSLT.
- A mechanism to specify which provider to use.

3. The DIG Description Logic Interface

3.1. Description Logics

Description Logics are a family of knowledge representation languages which can be used to represent the terminological knowledge of an application domain in a structured and formally well-understood way. They express knowledge about concepts and concept hierarchies. Description Logic (DL) was designed as an extension to frames and semantic networks, which were not equipped with a formal logic-based semantics.

The basic building blocks are concepts, roles and individuals. Concepts describe the common properties of a collection of individuals and can be considered as unary predicates which are interpreted as sets of objects. Roles are interpreted as binary relations between objects.

Each DL defines a number of language constructs (such as intersection, union, role quantification, etc.) that can be used to define new concepts and roles. The main reasoning tasks are classification and satisfiability, subsumption and instance checking.

Description logic systems have been used for building a variety of applications including conceptual modelling, information integration, query mechanisms, view maintenance, software management systems, planning systems, configuration systems, and natural language understanding.

Today Description Logic has become a cornerstone of the Semantic Web for its use in the design of ontologies. Ontology in computer science is the attempt to formulate an exhaustive and rigorous conceptual schema within a given domain, typically a hierarchical data structure containing all the relevant entities and their relationships and rules (theorems, regulations) within that domain.

The first DL-based system was KL-ONE (by Brachman and Schmolze, 1985). Some other DL systems came later. They are LOOM (1987), BACK (1988), KRIS (1991), CLASSIC (1991), FaCT (1998), and lately RACER (2001) and KAON 2 (2005).

3.2. DIG Interface

Description logic reasoners are becoming more widely used for reasoning about resources on the Semantic Web. To allow client tools to interact with different reasoners in a standard way, a common standard interface is highly desirable.

The Description Logic Implementation Group (DIG) is a self-selected group of DL system implementers aiming for standard system architectures for DL systems. As part of its activity, DIG is developing a standardised XML interface to Description Logics systems, providing a basic API to a DL system.

The DIG interface is an emerging standard for providing access to description-logic reasoning via an HTTP-based interface to a separate reasoning process. The interface is supported by most DL reasoners and easily allows for the construction of reusable software components. Available DIG reasoners at the time of writing include RACER, FaCT, and Pellet.

The present *Level 0* specification of DIG interface essentially consists of an XML Schema describing the expressions of the DL concept language, the available TELL and ASK operations along with the expected responses and administrative information.

A number of assumptions have been made for this initial specification:

- The specification is agnostic as to multiple client connections. Multi-threaded implementations of a reasoner may be provided, but no guarantees are made as to the semantics when clients attempt to simultaneously update and query.
- The connection to the reasoner is effectively stateless. Clients are not identified to the reasoner, thus the reasoner will not distinguish between clients and maintain any kind of consistency checking or record of which client is adding information or making requests. Conversely, a client has no way of ensuring that the reasoner has not been given additional information (such as additional axioms) since its last communication.
- There is no explicit classification request. The reasoner will decide when it is appropriate to, for example, build a classification hierarchy of concepts. This may happen after each TELL request, alternatively the reasoner may choose to defer the classification until absolutely necessary, or even when there is a lull in traffic.

Level 0 protocol is an XML- and HTTP-based standard for connecting client programs to description logic inference engines (e.g. FaCT or Racer). The use of HTTP allows client (and server) developers to use existing libraries for implementation, thus providing maximum portability.

DIG allows for the allocation of knowledge bases and enables clients to pose standard description logic queries. If a reasoner receives a message (either TELL or ASK) that contains any syntax that it does not understand, an error should be returned.

Two available XML schemas for DIG at present are:

- DIG 1.0 - namespace: http://dl.kr.org/dig/lang
- DIG 1.1 - namespace: http://dl.kr.org/dig/2003/02/lang

Version 1.1 allows for multiple knowledge bases.

3.2.1. Concept Language

DIG's concept language is based on SHOIQD−n , that is a description logic that includes the standard boolean concept operators (and, or, not), universal and existential restrictions, cardinality constraints, a role hierarchy, inverse roles, the one-of construct and concrete domains. SHOIQD−n was chosen as it is rich enough to support reasoning over the DAML+OIL language. DAML+OIL is an ontology language specifically designed for use on the Web, and it exploits existing Web standards (XML and RDF).

3.2.2. Knowledge Base Management

A DIG reasoner can deal with multiple knowledge bases (KBs) and Universal Resource Identifiers (URIs) are used in order to identify different KBs.

When a request is made to a reasoner to create a new knowledge base, the reasoner (if successful) will return a URI to the client, which can be used to identify the knowledge base during TELL and ASK requests. The URI is valid for that reasoner only, so making a request to another reasoner with the same URI will result in an error. Different clients of the same reasoner, however, may be able to share KBs by sharing URIs.

The two MANAGEMENT requests in DIG interface: request for a new KB and request to release the KB, will be displayed in the following samples.

A request for a new knowledge base consists of a single <newKB> element:

```
<?xml version="1.0" encoding="UTF-8"?>
<newKB
 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"/>
```

The response to a new KB request should consist of a single <response> element. This element contains either a single <kb> element stating the URI that the reasoner has allocated for the KB, or an <error> message in case an error occurred.

```
<?xml version="1.0" encoding="UTF-8"?>
<response
 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">
 <kb uri="urn:uuid:abcdefgh-1234-1234-12345689ab"/>
<response>
```

The above sample shows a response for successful knowledge base creation. In this case, the server has created a new UUID to refer to the knowledge base. This URI should then be used during TELL and ASK requests made against the KB.

Similarly, client can release a knowledge base through a request consisting of a single <releaseKB> element with an attribute specifying the KB to release in the <uri> attribute. Once a KB has been released, any requests made using the URI should result in an error. An example of Knowledge Base Release request is depicted below:

```
<?xml version="1.0" encoding="UTF-8"?>
<releaseKB
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"
uri="urn:uuid:abcdefgh-1234-1234-12345689ab"/>
```

For the previous Knowledge Base Release request, a possible response which indicates a successful release of that knowledge base will be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<response
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">
<ok/>
<response>
```

3.2.3. TELL Syntax

A TELL request must contain in its body a <tells> element, which itself consists of a number of tell statements. TELL requests are monotonic – i.e. once information has been told to a knowledge base, it can never be retracted or removed. A TELL request must be made in the context of a particular knowledge base (which is identified via an attribute of the enclosing <tells> element). The order of tell statements is unimportant.

Below is an example of a TELL request:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tells
 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"
 uri="urn:uuid:abcdefgh-1234-1234-12345689ab">
 <defconcept name="driver"/>
  <equalc>
     <catom name="driver"/>
     <and>
        <catom name="person"/>
        <some>
           <ratom name="drives"/>
           <catom name="vehicle"/>
        </some>
     </and>
  </equalc>
  <defconcept name="person"/>
  <defconcept name="vehicle"/>
  <defrole name="drives"/>
</tells>
```

The above sample defines three classes, vehicle, person, and driver, where driver is further defined as being precisely those persons who drive a vehicle.

The response to a TELL will be a <response> message containing either an <ok> element, signifying that the statements were received and interpreted correctly, or an <error> element which may include an optional error code, message, and detailed explanation. In addition, an <ok> message may contain warnings about the TELLs received.

The following sample illustrates a response to the previous TELL request. Here, the reasoner has received the TELLs, but is reporting the use of an undeclared class.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<response
 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">
 <ok>
   <warning message="Undeclared Name" code="99">
   Class Cat used but not declared
   </warning>
 </ok>
</response>
```

The concrete forms of the operators in TELL Language are given below:

Primitive Concept Introduction	<defconcept name="CN"/> <defrole name="CN"/> <deffeature name="CN"/> <defattribute name="CN"/> <defindividual name="CN"/>
Concept Axioms	<impliesc>C1 C2</impliesc> <equalc>C1 C2</equalc> <disjoint>C1... Cn</disjoint>
Role Axioms	<impliesr>R1 R2</impliesc> <equalr>R1 R2</equalr> <domain>R E</domain> <range>R E</range> <rangeint>R</rangeint> <rangestring>R</rangestring> <transitive>R</transitive> <functional>R</functional>
Individual Axioms	<instanceof>I C</instanceof> <related>I1 R I2</related> <value>I A V</value>

Table 3.1 TELL Language

3.2.4. ASK Syntax

An ASK request must contain in its body an <asks> element, which itself consists of a number of ask statements. Each ASK statement must have an attribute <id> which supplies a unique identifier for the query (within the context of the particular collection of queries). This allows the presentation of multiple queries in one request, which may in turn allow the reasoner to optimise the processing of these queries. Each <asks> element must also have an attribute that identifies the knowledge base that the queries are being posed against. The value of this attribute should be a URI which identifies a KB within the reasoner.

The following sample shows an ASK request.

```
<?xml version="1.0"?>
<asks
 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"
 uri="urn:uuid:abcdefgh-1234-1234-12345689ab">
 <satisfiable id="q1">
   <catom name="Vehicle"/>
 </satisfiable>
<descendants id="q2">
<and>
<catom name="person"/>
<some>
<ratom name="drives"/>
<catom name="vehicle"/>
</some>
</and>
</descendants>
<types id="q3">
<individual name="John Smith"></individual>
</types>
</asks>
```

The above sample contains three queries. The first asks about the satisfiability of the Vehicle concept, the second asks for all those concepts subsumed by the description given, i.e. all the drivers. The third query asks for the known types of the given individual.

The concrete forms of the operators for ASK are given below:

Primitive Concept Retrieval	`<allConceptNames/>`
	`<allRoleNames/>`
	`<allIndividuals/>`

Satisfiability	<satisfiable>C</satisfiable> <subsumes>C1 C2</subsumes> <disjoint>C1 C2</disjoint>
Concept Hierarchy	<parents>C</parents> <children>C</children> <ancestors>C</ancestors> <descendants>C<descendants/> <equivalents>C</equivalents>
Role Hierarchy	<rparents>R</rparents> <rchildren>R</rchildren> <rancestors>R</rancestors> <rdescendants>R<rdescendants/>
Individual Queries	<instances>C</instances> <types>I</types> <instance>I C</instance> <roleFillers>I R</roleFillers> <relatedIndividuals>R</relatedIndividuals> <toldValues>I A</toldValues>

Table 3.2 ASK Language

3.2.5. RESPONSE Syntax

The schema contains a description of the responses expected of the server to ASK requests. The response to an ASK request must contain in its body a <responses> element, which itself consists of a number of responses – one for each query in the ASK. Each particular response must have an attribute <id> which corresponds to the identifier of a submitted query.

The following sample shows a possible response to the queries above:

```
<?xml version="1.0"?>
<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 code="99" id="q3" message="No Such Individual">
    The individual named does not exist in the knowledge base.
 </error>
 <true id="q1"/>
 <conceptset id="q2">
   <synonyms>
     <catom name="bus driver"/>
     <catom name="passenger service vehicle operator"/>
   </synonyms>
   <synonyms>
     <catom name="car driver"/>
   </synonyms>
 </conceptset>
</responses>
```

The answer to the first query is a boolean while the second yields a collection of concepts, including two synonymous concepts. The third query yields an error as the reasoner does not know about the individual given. The responses do not necessarily occur in the order of submission.

The concrete forms of the operators for RESPONSE are given below:

Error	`<error/>`
Boolean	`<true/>` `<false/>`

Concept Set	`<conceptSet>`
	`<synonyms>S11... S1N</synonyms>`
	`<synonyms>SN1... SNM</synonyms>`
	`</conceptSet>`

Role Set	`<roleSet>`
	`<synonyms>R11... R1N</synonyms>`
	`<synonyms>RN1... RNM</synonyms>`
	`</roleSet>`

Individual Set	`<individualSet>I1... IN</individualSet>`

Individual Pair Set	`<individualPairSet>`
	`<individualPair>I1 I2</individualPair>`
	`<individualPair>J1 J2</individualPair>`
	`</individualPairSet>`

Values	`<sval>s</sval>` `<ival>i</ival>`

Table 3.2 RESPONSE Language

As it was primarily the result of a collaboration between two teams of leading DL engine implementers, FaCT and RACER, DIG's requirements were primarily driven by the goals of providing access to the capabilities of these two engines, and by the kinds of reasoning tasks that were of interest to the respective research teams. This situation seems to have influenced greatly the emphasis placed on DIG. For example, FaCT has very weak capabilities for reasoning about instances of classes, while it is very strong in reasoning about class and property hierarchies. Concurrently, the instance reasoning capabilities of DIG are less well-developed.

Although it is not a problem for the current uses of DIG, as additional DL reasoners are created by other research groups and companies, these restrictions are more likely to become barriers. But until now, researches to develop better specifications for DIG are still conducted.

4. Analysis and Design

4.1. What is RACER?

According to description in website http://www.sts.tu-harburg.de/~r.f.moeller/, **RACER** (Renamed ABox and Concept Expression Reasoner):

- is a Semantic Web inference engine for
  - § developing ontologies
  - § query answering over RDF documents and wrt. specified RDFS/DAML ontologies
  - § registering permanent queries (e.g., for building a document management system) with notification of new results if available (publish-subscribe facility)
- is a Description Logic reasoning system with support for
  - § TBoxes with generalized concept inclusions
  - § ABoxes
  - § Concrete domains (e.g., linear (in-)equalities over the reals)
- proves modal logic Km with graded modalities and axioms

The RACER system is a knowledge representation system that implements a highly optimized Tableau Calculus (a decision procedure solving the problem of satisfiability) for a very expressive description logic. It offers reasoning services for multiple TBoxes and for multiple ABoxes as well.

RACER implements the HTTP-based quasi-standard DIG for interconnecting DL systems with interfaces and applications using an XML-based protocol. On one hand RACER is available as a standalone version with no additional licenses required. This standalone version is also called RACER executable or RACER Server. The Racer Server can read DAML+OIL and OWL knowledge bases either from local files or from remote Web servers (i.e., a Racer Server is also an HTTP client).

4.2. Racer Proxy

In the context of multiple graphical interfaces and client programs (e.g., agents), a DL server such as RACER will be used by more than one client. For dealing with multiple clients, the RACER Server includes a subsystem called the Racer Proxy.

Racer Proxy is a program which administers communication between different RACER clients and one or more RACER servers, and also makes additional functions available for the clients. The clients, e.g. OilEd or RICE, are not connected directly to a RACER server. Instead, they are connected only to the Racer Proxy, which is the one connected to the RACER servers.

The Racer Proxy is started as a front-end to an associated RACER Server. It is configured to use a port number for external client access in the same way as a RACER Server. The port number of the associated RACER Server must be specified at proxy startup time. Then, a Racer Proxy is accessed just like a RACER Server, and it just forwards requests to a corresponding RACER Server. The task of the Racer Proxy is to provide locks for inference services of the RACER Server(s) that it "manages" such that instructions and queries of multiple clients are properly coordinated.

As described in its documentation, RACER Proxy implements the following functions:

1. Forwarding of clients' inquiries to the RACER server by means of TCP and HTTP interface.
2. Support of multiple users at the same time by synchronisation of the different inquiries.
3. Accelerating evaluation of clients' inquiries by automatically distributing those individual inquiries on different RACER servers.
4. Extending the Publish Subscribe mechanism by notification by E-Mail or TCP.

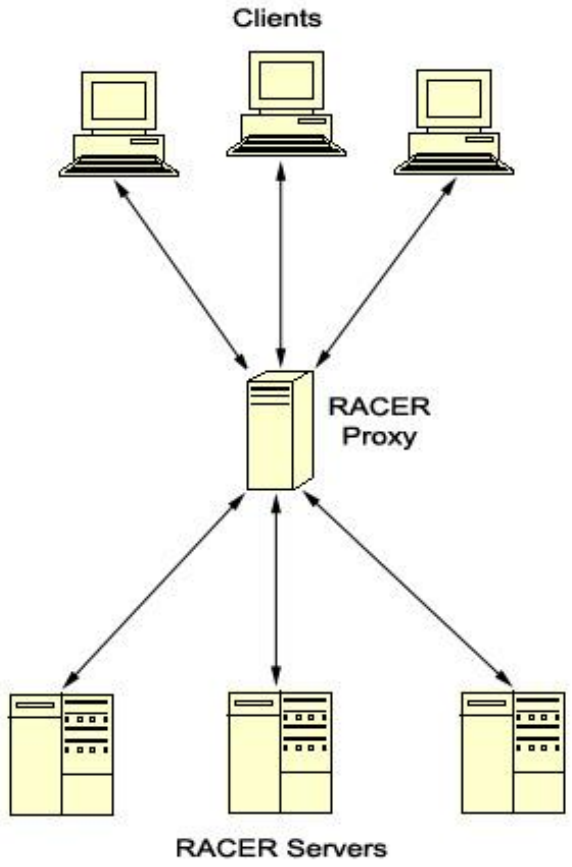

Figure 3.1 Client-Server Connection via Racer Proxy

4.3. Configuring HTTP-XML Processing in Racer Proxy

As it has been mentioned before, RACER implements DIG Interface to accommodate the clients accessing its DL reasoner via HTTP-XML connection. All requests and responses between client and the server will pass Racer Proxy, where the messages are manipulated into XML documents.

Methods which deal with XML document processing in Racer Proxy are found in classes `RacerHTTPRequest` and `RacerHTTPResponse` – both from racerproxy.util package.

Initial connection and receiving request from client to RACER Server using HTTP are handled by the class `HTTP_XMLClient` in racerproxy package. This class will invoke the method `readFromStream` in class `RacerHTTPRequest.` This method will parse message body from client, thus providing access to the message body as an XML document.

Similarly, the response from RACER Server will be parsed by the `readFromStream` method in class `RacerHTTPResponse`, which is invoked by class `Racer`. Below is the original method from class **`RACERHTTPRequest`**:

```
/**
   * Reads the http-message from the given stream and parses
the content to
   * an xml-tree.
   * @param istream the input-stream
   * @throws IOException is thrown if an error while reading
occurs or if the
   * message does not provide the appropriate syntax.
   */

public void readFromStream(DataInputStream istream) throws
IOException
   {
      httpRequest.readFromStream(istream);
      try {
            xmlDoc =
documentBuilderFactory.newDocumentBuilder().parse
(ByteArrayInputStream(httpRequest.getMessageBody()));
      }
      catch (Exception e){
   throw new IOException("xml-document could not be parsed:
" +
                           e.getMessage());
      }
    }
```

And below is the original method as part of class **`RACERHTTPResponse`**:

```
/**
   * Reads the http-message from the given stream and parses
the content to
   * an xml-tree.
   * @param istream the input-stream
   * @throws IOException is thrown if an error while reading
occurs or if the
   * message does not provide the appropriate syntax.
   */

public void readFromStream(DataInputStream istream) throws
IOException{
    httpResponse.readFromStream(istream);
    if(httpResponse.getMessageBody() != null){
      try{
        xmlDoc =
documentBuilderFactory.newDocumentBuilder().parse(new
ByteArrayInputStream(httpResponse.getMessageBody()));
      }catch(Exception e){
        System.out.println("fd:"+httpResponse);
```

```
        System.out.println(new
String(httpResponse.getMessageBody()));
        throw new IOException("xml-document2 could not be
parsed: " + e.getMessage());
      }
    }
  }
```

As seen from the codes above, Racer Proxy initially does not handle well-formed XML documents which do not conform to DIG Schema. The following statement:

```
xmlDoc =
documentBuilderFactory.newDocumentBuilder().parse(…);
```

only parses the XML Document without validating it.

Therefore, another method to parse XML document which can do validation as well is needed. This new method preferably is written inside a new class instead of embedding it in the currently available classes, so the code can become reusable and make it easier for further modification or development.

The following diagram depicts how a new class named **`XML_Validator`** will handle XML request and response from client and server respectively.

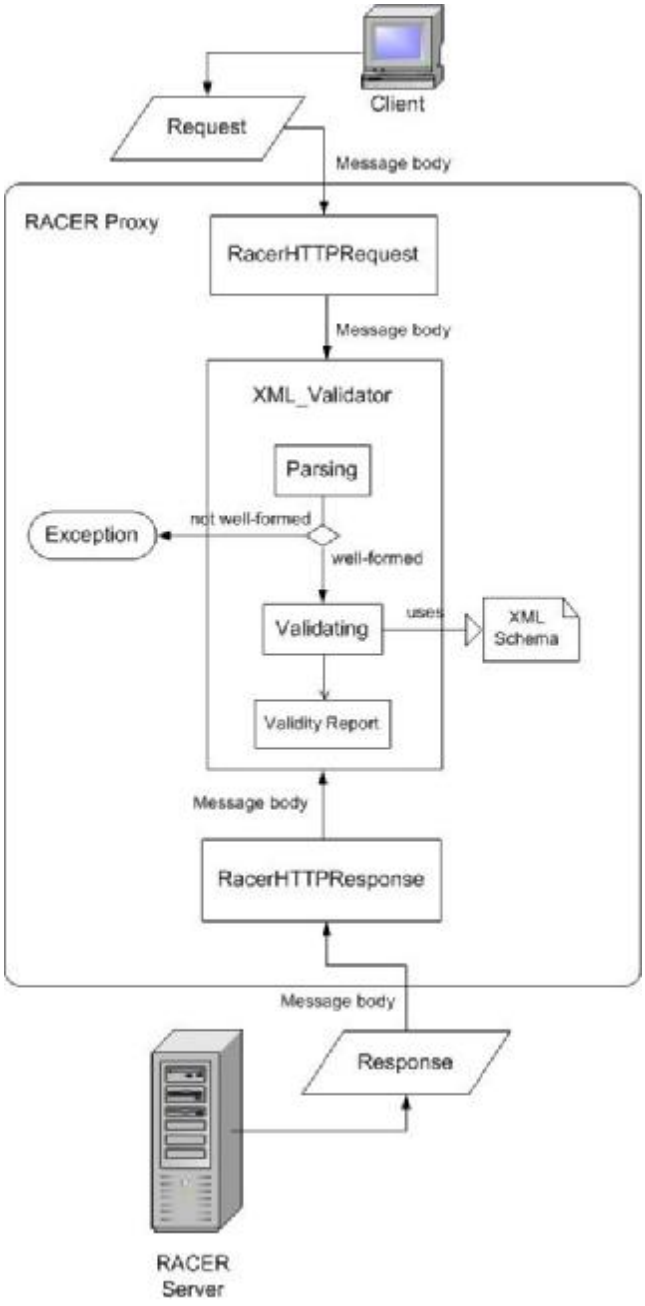

Figure 4.1 Graphical representation of a new class XML_Validator in Racer Proxy

With the presence of an XML Validator, parsing documents is no longer done by methods in the class `RacerHTTPRequest` or `RacerHTTPResponse`. Instead, the task would be switched to this new **`XML_Validator`** class, which would perform a validation as well.

First, this class will parse the input message into an XML document. If the message can be successfully parsed (i.e. the document is well-formed), then it will

proceed to validate the document based on a specified schema and output the validation result. Otherwise, an exception will be thrown.

4.3. Technology and Supporting Tools

4.3.1. JAXP Parser

In this project, the new XML validator will utilize JAXP parser to parse and validate the XML document. The version is JAXP 1.2.6 as contained in Java WSDP 1.5. Actually, the DIG Interface project has provided a specific XMLBeans which can be used to for parsing, creating, and manipulating instances of the DIG Schemas. However, it would be a better idea to write another XML processor which could validate the document based on any other schemas.

4.3.2. Logging Using the Log4J

For the logging purpose, the log4J from Apache Software would be used to facilitate outputting the message into different destinations. Log4j is a popular open source logging package for Java. It is fully configurable at runtime using external configuration files.

Log4j has three main components: loggers, appenders and layouts. By these three types of components, developers are able to log messages according to message type and level, and to control at runtime how these messages are formatted and where they are reported.

Logging requests are made by invoking one of the printing methods of a logger instance. These printing methods are: Debug, Info, Warn, Error, Fatal, and Log. By definition, the printing method determines the level of a logging request. The set of possible levels consists of DEBUG, INFO, WARN, ERROR and FATAL.

Log4j allows logging requests to print to multiple destinations. In log4j speak, an output destination is called an appender. Currently, appenders exist for the console, files, GUI components, remote socket servers, JMS, NT Event Loggers, and remote UNIX Syslog daemons. It is also possible to log asynchronously. More than one appender can be attached to a logger.

5. Implementation

5.1. Logging Configuration

The XML Validator would be implemented in Racer Proxy using log4j for its logging purpose. All messages resulted from parsing and validation process would be written to a file instead of displaying them in the console window. This way, the log messages are put in one localized archive and user could examine this archive at any time.

The log4j environment is fully configurable programmatically. However, it is far more flexible to configure log4j using configuration files. Currently, configuration files can be written in XML or in Java properties (key=value) format.

The following validatorLogger.config will be loaded at the same time with initialization of Racer Proxy application, and used as the configuration file for the subsequent XML Validator class:

```
log4j.rootLogger=INFO, FA

log4j.appender.FA=org.apache.log4j.FileAppender
log4j.appender.FA.File=app_log.log

log4j.appender.FA.layout=org.apache.log4j.PatternLayout
log4j.appender.FA.layout.ConversionPattern=%d{MM/dd/yy
HH:mm:ss} %p - %m%n %n
```

Since the root logger has a level INFO, then all logging requests with this level and higher would have the same properties. The following XML validator would use message with level ‘info’, ‘warn’, and ‘error’, therefore every log message would be printed in a file named “app_log.log”. This configuration could be modified freely to accommodate future requirements.

5.2. Writing an XML Validator

The class name for this XML validator would be **`XML_Validator`**. First thing to do in writing this validator is importing the `DocumentBuilderFactory` and `DocumentBuilder` classes. The `DocumentBuilder` class is used to obtain an

`org.w3c.dom.Document` document from an XML document, while the `DocumentBuilderFactory` class is used to obtain a `DocumentBuilder` parser.

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
```

Another particular class that needs to be imported is Logger from log4j.

```
import org.apache.log4j.Logger;
```

After all the needed classes are imported, there would be some variables that need to be declared. The `Document` doc is the object into which the message will be parsed.

```
…
public class XML_Validator {

 static Logger logger = Logger.getLogger(XML_Validator.class);

 static final String JAXP_Schema_Source =
"http://java.sun.com/xml/jaxp/properties/schemaSource";

 private Document doc;

…
}
```

The specific method which will parse and validate the input message is **`validateSchema`**. This method returns a `Document` type object after parsing and validating the specified input based on the specified schema location.

Besides the parameters `XMLDoc` as XML source and `SchemaUrl` as the schema location, this method required a parameter `sourceType` to determine whether the input message is a client's request or a server response. Based on that type, a request and a response may receive different treatment in case an exception is caught. When it is a request which is not well-formed, the application will not proceed with the request processing and instead throw an IOException.

```
public Document validateSchema(ByteArrayInputStream XMLDoc,
String SchemaUrl, String sourceType) throws IOException   {
 try{

…
…
```

```
…

}
 catch (SAXException e) {
        // Happens when the document is not well-formed
        // When it is a bad request, cease operation
        if (sourceType.toLowerCase() = = "request") {
          logger.error("SAXException for HTTP XML request: " +
                       e.getMessage() + "\n" +
                      "XML document could not be parsed.
                       Request will be ignored.");
            throw new IOException();
        } else logger.error("SAXException for HTTP XML " +
                            sourceType.toLowerCase() + ": " +
                            e.getMessage());
 }
 catch (IllegalArgumentException iae) {
         // Happens if the parser does not support JAXP
version
         logger.warn(iae + "\nDownload the latest copy of JAXP
                            from http://java.sun.com");
 }
 catch (java.io.IOException ioe)    {
        // An I/O exception of some sort has occurred
        // When it is a bad request, cease operation
        if (sourceType.toLowerCase() = = "request") {
           logger.error("IOException for HTTP XML request: " +
                        ioe.getMessage() + "\n" +
                       "XML document could not be parsed.
                        Request will be ignored.");
            throw new IOException();
        } else logger.error("IOException for HTTP XML " +
                            sourceType.toLowerCase() + ": " +
                            ioe.getMessage());

 }
 catch (ParserConfigurationException e) {
        // Indicates a serious configuration error
      logger.warn("ParserConfigurationException for HTTP XML "
          + sourceType.toLowerCase() + ": " + e.getMessage());
 }

 //return the message as well-formed XML document
 return doc;

}
```

The following explanation will concern the steps written inside the `try` block of `validateSchema`, where the message will be parsed and validated.

JAXP Parser utilized in this class will use `DocumentBuilder` classes, and the `System` property is set to:

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
"com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl");
```

Since this class was initially compiled under Java WSDP 1.5 specification, the `DocumentBuilderFactoryImpl` is found at `"com.sun.org.apache.xerces.internal.jaxp"` instead of `"org.apache.xerces.jaxp"`.

Next, a `DocumentBuilderFactory` is created with:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.
                                 newInstance();
```

To parse a XML document with a namespace, the `setNamespaceAware()` feature has to be set to `true`. By default, the `setNamespaceAware()` feature is set to `false`.

```
factory.setNamespaceAware(true);
```

As the main concern, the `setValidating()` feature of the `DocumentBuilderFactory` has to be set to `true` to make the parser a validating parser. By default, the `setValidating()` feature is set to `false`.

```
factory.setValidating(true);
```

Set the `schemaLanguage` and `schemaSource` attributes of the `DocumentBuilderFactory`. The `schemaLanguage` attribute specifies the schema language for validation, which is the JAXP. The `schemaSource` attribute specifies the XML schema document to be used for validation, and here it uses the given parameter `SchemaUrl`.

```
factory.setAttribute("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
"http://www.w3.org/2001/XMLSchema" );
factory.setAttribute(JAXP_Schema_Source, SchemaUrl);
```

Afterwards, create a `DocumentBuilder` parser.

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

This returns a new `DocumentBuilder`, with the parameters configured in the `DocumentBuilderFactory`.

Create and register an `ErrorHandler` with the parser.

```
Validator handler=new Validator();
builder.setErrorHandler(handler);
```

Then the XML message is parsed with the `DocumentBuilder` parser.

```
doc = builder.parse(XMLDoc);
```

If the document is well-formed, then it will continue with validation process, but if the document is not well-formed or another error occurs, an exception like described earlier will be thrown.

The following code shows how the validation result is reported. When any constraint violation against the specified schema is found, the logger will be invoked to inform user about the cause of this invalid request or response.

```
  if (handler.validationError==true) {
      logger.error("XML " + sourceType.toLowerCase() +
  " has validation error: " + handler.saxParseException.getMessage());
  }
  else
      logger.info("XML " + sourceType.toLowerCase() +
                  " is valid based on DIG interface.");

 }
```

The last thing to write in `XML_Validator` class is the `Validator`, an `ErrorHandler` of the type `DefaultHandler`, which registers errors generated by the validation.

```
private class Validator extends DefaultHandler
    {
       public boolean  validationError = false;
       public SAXParseException saxParseException = null;
       public void error(SAXParseException exception) throws
SAXException
       {
              validationError = true;
              saxParseException = exception;
       }
       public void fatalError(SAXParseException exception) throws
SAXException
       {
              validationError = true;
              saxParseException = exception;
       }
       public void warning(SAXParseException exception) throws
SAXException
       {      }
     }
```

5.3. Modification to Racer Proxy Classes

To initialize configuration properties of log4j logger used in `XML_Validator` class, the following statements will be added in `RacerProxy` class which is executed when the application is started.

```
…
import org.apache.log4j.PropertyConfigurator;

public class RacerProxy {
  private Vector clientConnectors = new Vector();
  private ConfigLoader cl;
  private RacerController rc;

  public static final Logger logger = new Logger();;


  public RacerProxy() {

    try{
      //Lade Config-Datei
      try{
        PropertyConfigurator.configure("log4jLogger.config");
        cl = new ConfigLoader("racerProxy.config");
      }catch(java.io.IOException ioe){
        throw new RacerProxyException("Config-file could not
be loaded: " + ioe.getMessage());
      }
…
```

After adding the XML_Validator in racerproxy.util package, parsing message from client is not done in the class `RacerHTTPRequest` any longer. Instead, the method `readFromStream` in that class will call method `validateSchema` from `XML_Validator` and pass the required parameters to let it do parsing and validation of the XML request.

```
public void readFromStream(DataInputStream istream) throws
IOException
   {   httpRequest.readFromStream(istream);
       if(httpRequest.getMessageBody() != null)
       {  ByteArrayInputStream Mssg = new
ByteArrayInputStream(httpRequest.getMessageBody());
      xmlDoc = validator.validateSchema(Mssg, "http://dl-
web.man.ac.uk/dig/2003/02/dig.xsd", "request");
       }
   }
```

In the same way, for class **RacerHTTPResponse**, the method `readFromStream` only has to call `validateSchema` from `XML_Validator` and pass the required parameters to parse and validate server's response.

```
public void readFromStream(DataInputStream istream) throws
IOException
  {
    httpResponse.readFromStream(istream);
    if (httpResponse.getMessageBody() != null)
    {
     ByteArrayInputStream Mssg =
new ByteArrayInputStream(httpResponse.getMessageBody());
     xmlDoc = validator.validateSchema(Mssg, "http://dl-
web.man.ac.uk/dig/2003/02/dig.xsd", "response");
    }
  }
```

The following sequence diagram illustrates simplified process of what is done to any request or response in Racer Proxy, especially concerning XML validation process, after the `XML_Validator` is added.

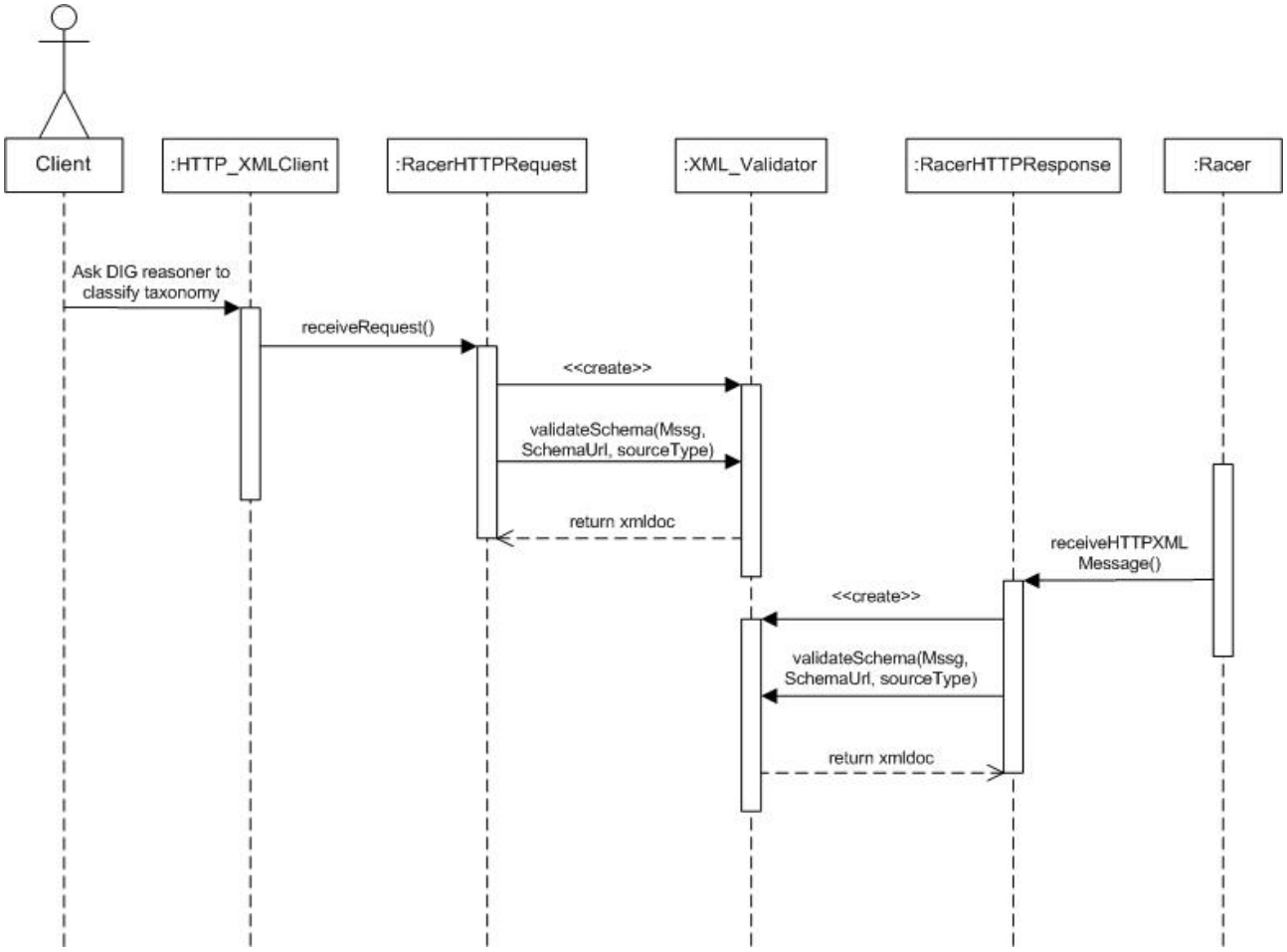

Figure 5.1 Sequence diagram of the usage of XML_Validator in Racer Proxy

This implementation of XML validator in Racer Proxy has been tested by writing HTTP POST requests from a temporary test class, and by using applications such as Protégé 3.1 beta and OilEd 3.4.7.

6. Conclusion and Outlook

Description logics are currently of much interest in the Semantic Web community, and the need of sophisticated DL reasoners is also increasing. For developers of semantic web applications, having a convenient means to access different standard DL reasoners would be extremely beneficial. The DIG DL reasoner interface specification from the Description Logic Implementation Group provides this means.

RACER is a DL reasoner that implements the DIG Interface in its HTTP-XML connection with clients through an application called Racer Proxy. Before this, Racer Proxy parses message from client or server and determines whether that message makes a well-formed XML document, but unfortunately it is only a parsing process without validating the document based on DIG Schemas. Meanwhile, in reality, both client's request and server's response do not always conform to DIG.

This Student Project presented an additional class in Racer Proxy packages which handles both parsing and validation of XML documents. By using this validator class named XML_Validator, DL reasoning process via HTTP between client and RACER will always be validated based on DIG interface schema. This class can be used to validate XML documents based on the other schema sources as well.

XML_Validator would inform any violation against DIG Schemas that may happen with both client's request and RACER Server's response. On the other hand, it would give a proof when RACER implements the DIG Interface correctly.

XML_Validator also provides logging mechanism using log4j which enables every log result to be written in a file instead of merely outputting it to the console window. This creates a localized logging archive to accommodate users or RACER developers in examination of any parsing or validation error that has happened.

For the future, a more user-friendly interface might be developed for Racer Proxy, such as generating more intelligible and helpful message for clients if any error is encountered. The present XML validator can also be improved to let users view the XML document if either parsing or validation error occurs, in order to help them understand the cause of error better. Usage of log4j for logging is also recommended for other parts of Racer Proxy.

REFERENCE

- **DIG**
  - The DIG Description Logic Interface: DIG/1.1, Sean Bechhofer
    http://www.sts.tu-harburg.de/~r.f.moeller/racer/interface1.1.pdf
  - Implementation experience with the DIG 1.1 specification, Ian Dickinson
    http://www.hpl.hp.com/techreports/2004/HPL-2004-85.pdf
  - *http://dl.kr.org/dig/*

- **RACER**
  - RACER System Description Homepage
    http://www.sts.tu-harburg.de/~r.f.moeller/racer/

- **Racer Proxy**
  - Racer-Proxy-Dokumentation Version 1, Christian Finckler

- **JAXP**
  - http://java.sun.com/xml/jaxp/index.jsp

- **Log4j**
  - http://logging.apache.org/log4j/docs/

- **XML Schema**
  - *http://www.brics.dk/~amoeller/XML/schemas/*
  - *http://xml.coverpages.org/schemas.html*

- **Description Logics**
  - *http://www.ida.liu.se/labs/iislab/people/patla/DL/*
  - *http://en.wikipedia.org/wiki/Description_logic*