Large Abox Store (LAS): Database Support for Abox Queries

Cui Ming Chen

A Thesis
In
The Department Of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September 2005

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: Cui Ming Chen

Entitled: Large Abox Store (LAS): Database Support for Abox Queries

And submitted in partial fulfillment of the requirements of the degree of

Master of Computer Science

Complies with the regulations of the university and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ **Chair**
Hon Fung Li

_____ **Examiner**
Shiri Nematollaah

_____ **Examiner**
Mudur Sudhir

_____ **Supervisor**
Haarslev Volker

Approved by _____
Chair of Department of Graduate Program Director

_____ _____
Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

ABSTRACT

Large Abox Store (LAS):

Database Support for Abox Queries

Cui Ming CHEN

The semantic web has drawn the attention from both academic and industry. Description Logics (DLs), a family of formal languages for representing knowledge and supporting reasoning about it, is regarded as a suitable tool that supports the semantic web and enables its data to be both machine readable and machine understandable. Recently, several approaches on how to combine description logics with databases were proposed. In this thesis, we propose techniques for connecting databases with description logic reasoners effectively and completely, and describe the design and implementation of LAS (Large Abox Store), a DL application combining Aboxes reasoning and database query processing to perform efficient reasoning for Aboxes containing role assertions.

With the goal to provide a user-friendly, scalable, and complete ontology query processor, we designed our system as an additional layer for the description logic reasoner—RACER.

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my supervisor, Dr. Haarslev for his direction, assistance, and guidance. Without his guidance and persistent help, this thesis would not have been possible.

I would like to thank Concordia Community, who provides great environment for learning and researching.

Special thanks should be given to my parents, my sister and all my friends who constantly support and understand me.

Table of Content

Table of Figure

1. Introduction

Being envisioned as the next generation web, the semantic web has drawn considerable attention from researchers in academia and industry. Its application will significantly benefit from combining expressive description logics (DLs) and databases. DLs with decidable inference problems are useful in structuring and representing knowledge in terms of concepts and roles, but reasoning procedures are currently not adequate for answering complex queries based on large scale data sets. Databases, on the other hand, are known for efficient data management and features such as robustness, concurrency control, recovery and scalability. Therefore, the importance and usefulness of combining description logics with database has been recognized for quite sometime.

Thus, the features of description logics and databases, such as semantic representation and powerful reasoning services, efficient management and accessibility, have led to the research to make good use of their respective advantages. As early as in 1983, [7] already investigated the technical issues of enhancing expert systems with database management facilities. In 1993, the approach of loading data into description reasoners was investigated [8]. In 1994, the theoretical foundations of a DL approach to DBs have been established [38]. [9] extends the traditional DL Abox with a DBox so that users can transparently make queries without being concerned about whether a DB or KB has to be accessed. Pointed out by [10] that the previous approaches lack of an automated translation between DL and database schemes, Roger proposed an object oriented

model that contains concrete classes, virtual classes and abstract classes, and two translations so that the schema in this model can be translated into a description logics schema and into a database schema [39].

In recent years ontology design—explicit formal specifications of the terms in the domain and relations among them [4]—has focused on domains with massive data, where a large number of related individuals exist. Although RACER has strong reasoning abilities, there exists a great potential to combine it with databases in order to deal with large numbers of related individuals. Our mission is to build an extra layer combining RACER with databases so that reasoning in this domain is efficiently supported.

Before illustrating the techniques we used to combine DLs with DBs, and describing our system in detail, we will first give an introduction to the semantic web and description logics.

1.1 Semantic Web

1.1.1 History of the semantic web

Invented in 1989, web browsers and World Wide Web (WWW) servers were designed to build a place where one can find any information and reference [1]. In other words, the WWW realizes the dream of information sharing all around the world.

Despite its good information sharing capabilities, the current World Wide Web, which is based primarily on documents written in HTML, offers limited support to classify blocks of text on a page, hence the content of the web pages is mostly machine readable but not machine understandable. Focused on a visual presentation, HTML is used for describing and presenting interspersed text with multimedia objects. Lacking the facility to represent formal semantics of embedded data, the current web does not facilitate reasoning and answering intelligent queries. As a result, the idea of the semantic web was borne.

The semantic web, proposed also by Tim Berners-Lee, who wove the World Wide Web and created a mass medium for 21st century, is a project that intends to create a universal medium for information exchange by giving semantics, in a manner understandable by machines on the content of documents on the web. It addresses to overcome the shortcoming of the current WWW, using descriptive technologies such as RDF and OWL, and the data-centric, customizable markup Language XML.

The main idea of the Semantic Web is to combine the descriptive technologies in a way to supplement or replace the content of web documents by storing the descriptive data in web-accessible databases or as markup within documents. The machine-readable descriptions allow content managers to add meaning to the content, thereby enable machines to understand and consume the semantics of information in order to support automated reasoning.

1.1.2 Components of the semantic web

The Semantic Web uses the following relevant standards: XML, RDF, RDF Schema, OWL, URIs and HTTP. We will discuss each of them briefly.

URIs

URIs are Uniform Resource Identifiers, which are text strings that identify resources or concepts.

HTTP

HTTP is a protocol for web browsers and servers to communicate with each other. It has three main methods: GET—asks for information about a resource; POST—sends a request to a resource; PUT—sends updated information about a resource. By using HTTP, Semantic Web services get all this functionality for free, because it is already built into many HTTP Servers and clients that are available for lots of platforms in almost every programming language.

XML

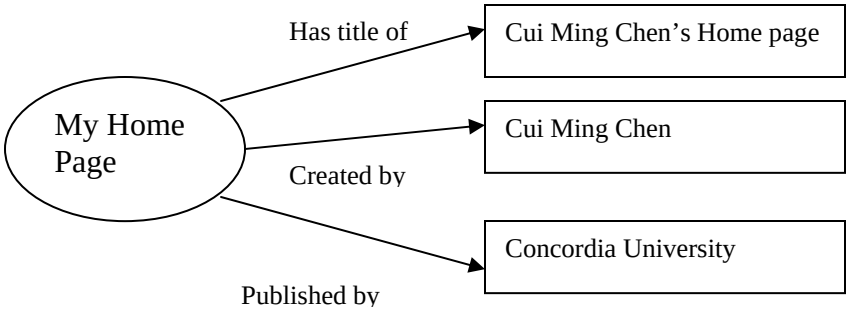
The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language for creating special-purpose markup languages. Its simultaneously human and machine readable format, its hierarchical structure and its strict syntax and parsing requirements offer document storage and processing, both online and offline. However, its syntax is fairly verbose and partially redundant, and there exist no facilities for randomly accessing or updating only

portions of a document. Moreover, mapping XML to the relational or object oriented paradigm is often cumbersome [48].

XML was originally designed for documents, not data. As a result, the features that are suitable for documents might cause problems when expressing data. For example, there can be many ways to express the same thing in a document. Because of lack of a standard expression, it is not easy to come up with algorithms for machines to extract the meaning. If one combines two XML documents, the resulting document may no longer be well-formed XML.

RDF

The resource description framework (RDF) is a family of specifications for a metadata model that is often implemented as an application of XML [2]. RDF explicitly makes statements about resources in the form of a subject-predicate-object expression, namely, a triple in RDF terminology. The subject is the resource—"things" being described. The predicate describes an aspect of the resource, and often expresses a relationship between the subject and the object. The object is the object of the relationship. The following shows an example of RDF forms. It has the subject as "My Home Page", three predicates "Has title of", "Created by" and "Published by" and three objects " Cui Ming Chen's home page", "Cui Ming Chen" and "Concordia University" respectively.




```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description
   rdf:about="http://www.cs.concordia.ca/~cui_chen/">
    <dc:title>Cui Ming Chen's Home Page</dc:title>
    <dc:creator>Cui Ming Chen </dc:creator>
    <dc:publisher>Concordia University </dc:publisher>
  </rdf:Description>
</rdf:RDF>
```

Figure 1.1 Example of RDF (Upper part: graphic explanation, lower part : RDF format)

RDFS

RDFS (The Resource Description Framework Schema) is an extension to RDF that allows one to define RDF vocabularies using RDF itself. It includes the relationship between things such as rdfs:subClassOf and rdfs:subPropertyOf where "rdfs" is an abbreviation for http://www.w3.org/2000/01/rdf-schema#. For example:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="Teacher">
     <rdfs:comment>Teacher Class</rdfs:comment>
     <rdfs:subClassOf rdf:resource="#Person"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Course">
     <rdfs:comment>Course Class</rdfs:comment>
     <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-
rdf-syntax-ns#Resource"/>
</rdfs:Class>
```

```
<rdf:Property rdf:ID="teacher">
      <rdfs:comment>Teacher of a course</rdfs:comment>
      <rdfs:domain rdf:resource="#Course"/>
      <rdfs:range rdf:resource="#Teacher"/>
</rdf:Property>
```

Figure 1.2 Example of RDFS

OWL

As described above, RDF Schema is a vocabulary for describing properties and classes of RDF resources, with semantics for generalization hierarchies of such properties and classes. However, it is not expressive enough. Therefore, OWL (Ontology Web Language) adds more vocabulary to describe properties and classes. For example, it adds facilities to express the relationship between classes, the cardinality, equality characteristics of properties and enumeration of classes.

There are three subclasses of OWL [43] defined as follows:

- OWL Lite supports users who primarily need classification hierarchies and simple constraints.
- OWL DL supports users who want the maximum expressiveness while retaining computational completeness and decidability. OWL DL is named according to its correspondence with description logic, a research field that has studied the logics that form the formal foundation of OWL, which we will introduce later in this chapter.
- OWL Full is for the users who want maximum expressiveness and syntactic freedom of RDF with no computational guarantees.

In conclusion, the ontology vocabulary for the Semantic Web is illustrated in Figure 1.3.

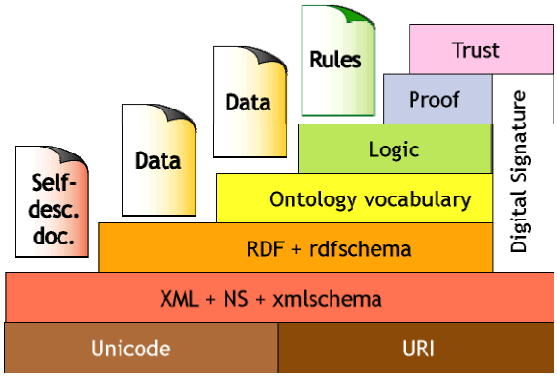


Figure 1.3 Ontology Semantic Layer [Tim Berners-Lee, the semantic web and challenge]

1.2 Description Logics

1.2.1 Introduction to description logics

As discussed before, the goal of the semantic web is to facilitate machines to extract semantics from human words or to process natural language. In order to achieve this, everything has to be written out in structured languages. Because the ultimate goal is ambitious, and it is hard to achieve within a few years, nowadays, researchers have made more reasonable and short-term achievable objectives—collecting data in a useful way, like a large database.

In order to represent data in a way that machines can more easily do the automatic reasoning based on the reasoning facility supplied, more support for reasoning mechanisms within the semantic web are needed. Description Logics, a family of knowledge representation languages, are a possible solution for this. DLs can deal with the expressive knowledge about concepts and concept hierarchies. Given a Tarski style declarative semantics, description logics are regarded as an important

formalism based on the traditions of frame-based systems, and can provide object-oriented representations, semantic data models and type systems.

Description logics are comprised of three basic building blocks: concepts, roles and individuals. Concepts, interpreted as sets of instances, are used to describe the common properties of a collection of instances and can be considered as unary predicates in first order logic. Roles are interpreted as binary relations between objects, while individuals are defined as the instances of concepts.

In description logics, there exist some language constructors to define new concepts and roles, such as intersection, union, role quantification, etc.

The main inference services supported by description logic reasoners are classification, satisfiability checking and Abox realization. Classification is the computation of a concept hierarchy based on subsumption. Satisfiability is to check whether there exists at least a model. Abox realization is to compute the most-specific concepts for each individual in the Abox. We will describe the inference services of description logic in detail in a later subsection.

1.2.2 Components of description logics knowledge bases

A DL knowledge base is comprised of two components—a “Tbox” and an “Abox.” The Tbox contains a terminology and is built through declarations that describe general properties of concepts. Due to the subsumption relationships between concepts, the hierarchy structure of the Tbox is analogous to a lattice.

The Abox contains extensional knowledge—also called assertional knowledge, which is specific to the individuals of the domain of discourse. The Tbox, as internal knowledge, is often thought to be unchangeable, while the Abox, which represents the extensional knowledge, is changeable [3].

1.2.3 Description languages

1.2.3.1 The basic description languages AL

As introduced in [40], the language AL (Attributive language) is regarded as a minimal language that is of interest. The other languages of this family are extensions of AL .

The basic description language AL is formed according to the following syntax rule [6]:

$C, D \rightarrow A \mid$ (atomic concept)

$\mathrm{T} \mid$ (universal concept)

$\bot \mid$ (bottom concept)

$\neg A \mid$ (atomic negation)

$C \cap D \mid$ (intersection)

$\forall R.C \mid$ (value restriction, universal quantification)

$\exists R.\mathrm{T} \mid$ (limited existential quantification)

1.2.3.2 The description logic ALC

In the area of Description logics, *ALC* is considered as a basic description logic with structures related to first order predicate logic: conjunction, disjunction, full negation and existential and universal role quantification.

ALC concepts are built using a set of concept names (NC) and role names (RN). The meaning of concepts is given by a Tarski style model theoretic semantics using an interpretation $I = (\Delta^I, \cdot^I)$, where Δ^I is the domain and $\cdot^I$ is the interpretation function. The function $\cdot^I$ maps each concept name in NC to a subset of Δ^I and each role name in RN to a binary relation over Δ^I (a subset of $\Delta^I \times \Delta^I$) such that the following conditions are observed:

$\mathrm{T}^I = \Delta^I$	top
$\perp^I = \phi$	bottom
$\neg A^I = \Delta^I \setminus A^I$	complement [C]
$(C_1 \cap C_2)^I = C^I{}_1 \cap C^I{}_2$	conjunction
$(C_1 \cup C_2)^I = C^I{}_1 \cup C^I{}_2$	disjunction [u]
$(\forall R.C)^I = \{i \in \Delta^I \mid \forall j.(i, j) \in R^I \Rightarrow j \in C^I\}$	universal quant.
$(\exists R.C)^I = \{i \in \Delta^I \mid \exists j.(i, j) \in R^I \wedge j \in C^I\}$	existential quant. [ε]

Note that in AL, negation can only be applied to atomic concepts, and only the top concept is allowed in the scope of an existential quantification over a role.

However, *ALC* supports full negation and full existential quantification which means negation can be applied to complex concepts as well.

1.2.3.3 The family of AL language

The *AL* -family can be obtained by adding further constructors to *AL* . For example, number restrictions $[N]$ are written as $\geq n\ R$ (at-least restriction) and as $\leq n\ R$ (at-most-restriction). They are interpreted as

$(\geq n\ R)^I = \{a \in \Delta^I \mid |\{b \mid (a,b) \in R^I\}| \geq n\}$

$(\geq n\ R)^I = \{a \in \Delta^I \mid |\{b \mid (a,b) \in R^I\}| \leq n\}$

Transitive roles (R_+) can be defined as $\{< x, y > \mid < x, z > \in R^I \cap < z, y > \in R^I\}$.

Inverse roles (R^-) can be defined as $\{< x, y > \mid < y, x > \in R^I\}$

Functional roles (F) is defined as iff $\{(a,b), (a,c)\} \subseteq R^I$ implies b=c.

Extending *AL* by any subset of the above constructors yields a particular *AL* -language. An *AL* -language is named by a string of the form $AL[u][\varepsilon][N][C]$ where a letter in the name stands for the presence of the corresponding constructor. For example, $AL\varepsilon N$ is the extension of *AL* by full existential quantification and number restrictions. In DL terminology, *S* is often used to stand for *ALC* with transitive role (R_+) , *H* for role hierarchy (e.g. $hasDaughter \subseteq hasChild$), *O* for nominals/singleton classes (e.g.,{Canada}), I for inverse roles (e.g. $isChildOf \equiv hasChild^-$), N for number restrictions (e.g. $\geq 2 hasChild$) and Q for qualified number restrictions (e.g. $\geq 2 hasChild.Doctor$).

1.2.3.4 Inference services

In description logics, concept satisfiability, subsumption, instance checking, retrieval and realization are the interesting inference services. Let us assume a knowledge base $\Sigma =< Tbox, Abox >$:

Concept satisfiability is the problem of checking whether conecept C is satisfiable, w.r.t Σ, i.e. whether there exists a model I of Σ such that $C^I \neq \phi$.

Subsumption is the problem of checking whether concept C is subsumed by D w.r.t. Σ, i.e. whether $C^I \subseteq D^I$ in every model I of Σ

Instance checking is the problem of checking whether the assertion $C(a)$ is satisfied in every model of Σ.

Abox realization is to compute for all individuals in an Abox their most-specific concept names w.r.t. Tbox T.

All inference services can be reduced to the satisfiability test. For example, the subsumption test $C \subseteq D$ can be reduced to test whether $\Sigma \cup \{(C \cap \neg D(x)\}$ has no model. And instance checking $C(a)$ can be reduced to test whether $\Sigma \cup \{\neg C(a)\}$ has no model.

1.3 Motivation

As evolved from a combination of frame based systems and predicate logic, description logics were designed to overcome some problems of the frame based

systems and to provide a clean and efficient formalism to represent knowledge. Therefore, the semantic organization of data and the powerful deductive capabilities are main characteristics of description logics.

1.3.1 Current description logics reasoners

There are a few Description Logics reasoners available these days, such as FaCT (**Fa**st **C**lassification of **T**erminologies), RACER (**R**enamed **Abox** and **C**oncept **E**xpression **R**easoner), Pellet, KAON2.

FaCT is a description logic classifier that can also be used for modal logic satisfiability testing. It can deal with the *SHF* logic (*ALC* augmented with transitive roles, functional roles and a role hierarchy), or with the logic *SHIQ* [5] (*SHF* plus inverse roles and qualified number restrictions), both of which use sound and complete tableau algorithms. FACT++ is a new generation of FACT reasoner. It used the established FACT algorithms, but with a different internal architecture. However, neither of them supports individuals.

RACER is a knowledge representation system, which implements a highly optimized tableau calculus for a very expressive description logic [44]. It offers the services for both multiple Tboxes and Aboxes. It can deal with the description logic $ALCQHI_{R^+}$ also known as *SHIQ*. Moreover, RACER provides reasoning with concrete domains.

Pellet is an OWL DL reasoner based on the tableaux algorithms developed for expressive Description Logics. It supports all the OWL DL constructs including the ones about nominals, namely owl:oneOf and owl:hasValue. Pellet provides reasoning that is sound and complete for OWL DL without nominals (SHIN(D) in DL terminology) and OWL DL without inverse properties (SHON(D) in DL terminology). It is provably sound but incomplete with respect to all OWL DL constructs (SHION(D) in DL terminology).

KAON2 is a OWL-DL reasoner and it supports SHIQ(D). KAON2 supports answering conjunctive queries whose variables in a query are bound to individuals explicitly occurring in the knowledge base, even if they are not returned as part of the query answer [45].

1.3.2 Problems for DL reasoners while dealing with large Aboxes

However, although some description logic reasoners can deal with a very expressive description logic, they are often not very efficient in reasoning with large Aboxes. One reason is that they compute the complex tableau algorithms mainly in the main memory, which means they are constrained by the limitation of the memory. Therefore, if a reasoner terminates, it will not store any results permanently. It will lose all the information, and has to recompute it again. Another reason is that some of the tableau computations might require a lot of CPU running time. Instead, due to the facility of nowadays DBMS, it would be a good choice to use SQL to answer some DL queries.

The recent DL research was stimulated on merging DL reasoners with databases, which are well known for their efficient management of huge data sets. In order to address this problem, we use so-called Pseudo Model techniques to combine databases and the description logics reasoner RACER. We proposed to use databases as reasoning filters in order to reduce the number of individuals which have to be reasoned by RACER. In particular, most of the inferences made by RACER are converted into a collection of SQL queries, with the goal to gain efficiency by relying on the optimization facilities of existing DBMS. We also implemented an algorithm to generate the whole Tbox hierarchy based on the taxonomy retrieved from RACER. Therefore, in order to answer queries about the Tbox hierarchy, one can just rely on database SQL queries. As for the Abox reasoning, LAS first stores pseudo models of all individuals and concept names into the database. Then, it uses these pseudo models to do so-called mergable tests, and choose the possible candidates to let RACER do the final checks. After LAS gets the answers from RACER, it stores all the information into the database for future queries. This approach is especially efficient for the registered users of LAS, who log off the system and will log on again in the future. As for RACER, since all the reasoning is performed in main memory, it will lose all the information whenever the server is shut down or is required to load another new file. Therefore, it does not keep any information permanently at all. One of the goals of LAS is to keep all information for reuse in the future.

LAS system is developed co-operately with my colleague Jiao Yue Wang. This project is a result of two master theses. There exist common parts in both thesis: The techniques for combining database and RACER, the initial design of the system and the database schema. This thesis covers mainly on Abox queries part, while the Tbox part is discussed in Jiao Yue Wang's thesis

In the following chapters, we will introduce the LAS (Large Abox Store) system which extends the DL reasoner RACER with a relational database used to store and query Tbox and Abox information.

2. Techniques for combining databases and RACER

Currently, there already exists a system that can deal with large Aboxes, which combines a DL reasoner with a database, Instance Store [46]. However, it supports only a restricted form of Aboxes, namely role-free Aboxes, which means there are no role assertions in Aboxes allowed. With the role assertions, the individuals are no longer isolated. They are linked to different groups by different role assertions. Therefore, it is more complicated to deal with large Aboxes with role assertions, because it will increase the expressiveness due to role hierarchies and transitive roles.

Recently, the research on description logics has shifted to the development of algorithms and optimized implementations for increasingly expressive DLs, which support transitive roles and general inclusion axioms or even more expressive languages [11]. In this chapter, we will discuss two techniques that can be used to reason about an Abox containing role assertions, the so-called precompletion and psudo model techniques.

2.1 Precompletion

The main idea of the precompletion techniques is to eliminate Abox axioms specifying relationships between individuals, and transform a general Abox into an equivalent role-free Abox [12, 13]. After eliminating those role assertions between individuals, the individuals in the Abox become isolated, and are no

longer linked with each other by a role. Therefore, the individuals become independent and can be verified using a standard Tbox reasoner. Thus, role assertions can be transformed into concept assertions, and the remaining relevant elements are only concept assertions, which means Abox reasoning can be reduced to Tbox reasoning.

2.1.1 The language scope of precompletion— $ALCFHR^+$

As we already introduced ALC, the description logics basic language, $ALCFHR^+$ is the extension of ALC by adding functional roles, transitively closed roles, and role hierarchies. It has a standard Tarski-style semantics based on an interpretation $I = (\Delta^I, \cdot^I)$. Let us assume A, C, D are concepts names, R, S are roles names. T is a set of transitive roles and F is a set of functional roles, and a, b are individuals names. The syntax and semantics of $ALCFHR^+$ is as follows:

Concepts:

Syntax	Semantics
A	$A^I \subseteq \Delta^I$
$\neg C$	$\Delta^I \setminus C^I$
$C \cup D$	$C^I \cup D^I$
$C \cap D$	$C^I \cap D^I$
$\exists R.C$	$\{ a \in \Delta^I \mid \exists b \in \Delta^I : (a,b) \in R^I, b \in C^I \}$
$\forall R.C$	$\{ a \in \Delta^I \mid \forall b : (a,b) \in R^I \Rightarrow b \in C^I \}$

Terminology Axioms:

Syntax	Semantics
$R \in T$	$R^I = (R^I)^+$
$R \in F$	$\Delta^I \subseteq (\exists_{\leq 1} R)^I$
$R \subseteq S$	$R^I \subseteq S^I$
$C \subseteq D$	$C^I \subseteq D^I$

Assertions

Syntax	Semantics
$a : C$	$a^I \in C^I$
$(a, b) : R$	$(a^I, b^I) \in R^I$

2.1.2 Technical definitions for $ALCFHR^+$

We define the concept name Top (T) and Bottom ($\perp$), by denoting T an abbreviation for $C \cup \neg C$, and $\perp$ as for $C \cap \neg C$.

(1) Generalized concept inclusions:

If C and D are concept terms, then the generalized concept inclusion $C \subseteq D$ is a terminological axiom. It can be transformed into the equivalent assertion $T \subseteq \neg C \cup D$. Therefore, the whole Tbox can be transformed into the required generalized concept inclusion form.

(2) Negation normal form:

The concept expressions are all transformed into negation normal form, which means the negation constructor $\neg$ can appear only in front of concept names.

(3) Label of individuals

Given a knowledge base Σ = <T, A> where T represents the Tbox and A represents the Abox. The label $\xi(\Sigma, o)$ of an individual o is the set of concept expressions occurring in the assertions on the individual itself.

$\xi(\Sigma, o) = \{C | o : C \in A\}$

While the individual concept expression $\bigcap \xi(\Sigma, o)$ is defined as the conjunction of all the concept expressions in $\xi(\Sigma, o) : C_1 \cap \ldots \cap C_n$.

(4) Functional Role Operator

The functional role operator is defined as $\overset{o}{\approx}_A$ (depending on the individual name o, and an Abox name A) , which holds between two functional roles R and S, and the Abox assertions force the R and S successors of the individual name o to be the same element. In other words, if (x, y) $\in R^I$, (x, z) $\in S^I$, and $R \subseteq F_1, R_i \subseteq F_i (1 \le i \le n), S \subseteq F_n$, where F_i^I is functional role, then y = z. Thus, $R \overset{o}{\approx}_A S$.

(5) Role Hierarchy Operator

We assume in the language $ALCFHR^+$ that exist no cyclical definitions among role names, which means if there exists $S \subseteq R$ and $R \subseteq S$, then we can conclude that $S = R$. Therefore, the role hierarchy can be defined using a partial order $\le$.

It is easy to prove that a subrole of a functional role is also functional.

2.1.3 Precompletion rules

A knowledge based is precompleted if all the information entailed by the presence of role assertions is exhibited in the form of concept assertions [11]. Therefore, in the precompleted knowledge base, role assertions become redundant and can be removed. The precompletion rules that break down the role assertions to concept assertions are as follows:

(1) $A \to \subseteq \{o : C\} \cup A$: if o is in O, T $\subseteq C$ is in Tbox and o:C is not in A. This is for global axioms.

(2) $A \to \cup\{o:D\} \cup A$: if $o:C_1 \cup C_2$ is in A, and D = C_1 or D = C_2, and neither $o:C_1$ nor $o:C_2$ in not in A. This rule deals with disjunctions.

(3) $A \to \exists\{o':C\} \cup A$: if $o:\exists R.C$ and $<o,o'>:S$ is in A, $R \overset{o}{\approx}_A S$, and $o':C$ is not in A. This rule deals with existential restrictions.

(4) $A \to \forall^+\{o':\forall R.C\} \cup A$: if $o:\forall T.C$ is in A, $<o,o'>:S$ is in A, and there is $R \in TRN$ (transitive roles) such that $S \le R \le T$, and $\{o':\forall R.C\}$ is not in A. This is a transitive role value restriction rule.

(5) $A \to \cap\{o:C_1, o:C_2\} \cup A$: if $o:C_1 \cap C_2$ is in A, neither $o:C_1$ nor $o:C_2$ is in A. This rule deals with conjunctions.

(6) $\forall^1\{o':C\} \cup A$: if $o:\forall R.C$ and $<o,o'>:S$ are in A, there is $R' \le R$ s.t. $R \overset{o}{\approx}_A S$ and $o':C$ is not in A.

(7) $\forall\{o':C\} \cup A$: if $o:\forall R.C$ is in A, and $<o,o'>:S$ is in A, and $S \le R$, and $o':C$ is not in A. This rule deals with universal quantifications.

As proved in [11], the precompletion algorithm will always terminate no matter in what order applicable rules are chosen. Although different strategies for the priority rules to be chosen can lead to a different computational complexity, as long as the disjunction of concept exists, the worst case of the computational complexity will be exponential. Now, we will give an example to explain how to apply these rules to transform an Abox with role assertions into a role-free Abox.

Tbox: $T \subseteq \neg D_1 \cup X \cup Y$

Abox: $(a,b_1):R_1$ $\quad$ $b_1:C_1$ $\quad$ $a:\forall R_1(A \cup B)$

$(a,b_2):R_2$ $\quad b_2:C_2$ $\quad a:\forall F_1.D_1$

$(a,b_3):R_3$ $\quad b_3:C_3$ $\quad a:\forall F_2.D_2$

$R_1 \le F_1$ $\quad R_2 \le F_1$ $\quad R_2 \le F_2$

$R_3 \le F_2$ $\quad$ (F_i is functional roles)

Step1: Precompletion rule (1)

$a:\neg D_1 \cup X \cup Y$

$b_1:\neg D_1 \cup X \cup Y$

$b_2:\neg D_1 \cup X \cup Y$

$b_3:\neg D_1 \cup X \cup Y$

Step2: Completion Rule

$(a,b_1):R_1$

$a:\forall R_1.(A \cup B) \quad \Rightarrow b_1:A \cup B$

Step3: Precompletion rule (7)

$(a,b_1):R_1$

$a:\forall F_1.D_1 \quad \Rightarrow b_1:D_1$

$R_1 \le F_1$

Step4: Precompletion Rule (6)

$(a,b_2):R_2$

$a:\forall F_1.D_1 \quad \Rightarrow b_2:D_1$

$R_1 \le F_1, R_2 \overset{a}{\approx}_A R_1$

Step5: Precompletion Rule (7)

$(a,b_2):R_2$

$a:\forall F_2.D_2 \quad \Rightarrow b_2:D_2$

$R_2 \le F_2$

Step6: Precompletion Rule (7)

$(a,b_3):R_3$

$a:\forall F_2.D_2 \quad \Rightarrow b_3:D_2$

$R_3 \le F_2$

Step7: Precompletion Rule (2)

$b_1:A \cup B \quad \Rightarrow b_1:A$

So finally, the new Abox is as follows:

$(a,b_1):R_1$	$b_1:C_1$	$a:\forall R_1(A\cup B)$
$(a,b_2):R_2$	$b_2:C_2$	$a:\forall F_1.D_1$
$(a,b_3):R_3$	$b_3:C_3$	$a:\forall F_2.D_2$
$R_1\le F_1$	$R_2\le F_1$	$R_2\le F_2$
$R_3\le F_2$	$b_1:A$	$b_1:D_1$
$b_2:D_1$	$b_2:D_2$	$b_3:D_2$

Note that in the step 7, we can choose either $b_1:A$ or $b_1:B$, Thus, if there are n individuals and m disjunctions for each individual, represented as $o_i:C_1\cup...\cup C_m(1\le i\le m)$, there would be m^n choices. Therefore, although the precompletion algorithm can guarantee to generate a finite number of precompletions, but it has an exponential computational complexity in the worst case.

2.2 Pseudo model techniques:

In this section, we will discuss another approach to combine databases with description logic reasoners, called Pseudo Model Technique. The main advantage of this technique is to avoid the expensive satisfiability test, which is due to the undeterministic completion rules.

As for description logics reasoning, classifying Tbox and realizing Abox are the two most expensive computations. Therefore, it is critical to speed up the subsumption test during the classification of a Tbox and realization of an Abox. The pseudo model technique introduces a very effective mergable test which re-uses the information computed for previous satisfiability tests by the tableau algorithms [14].

Each subsumption test $(C \subseteq D)$ can be transformed into a satisfiability test as $(unsat?(C \cap \neg D))$. If $(C \cap \neg D)$ is satisfied, then $(C \subseteq D)$ does not hold, which means C is not subsumed by D. In order to test whether the conjunction $(C \cap \neg D)$ is satisfied or not, a mergable test between C's pseudo model and $\neg D$'s pseudo model can be applied. If the pseudo models of C and $\neg D$ can be merged by the mergable test, in other words, there is no interaction between the pseudo models of C and $\neg D$, then $(C \cap \neg D)$ is satisfied. Therefore, it can be inferred that C is not subsumed by D.

The pseudo model technique is sound but incomplete. If the respective pseudo models of C and $\neg D$ cannot be merged, the tableau algorithms must be applied later on to test the satisability of the conjunction $(C \cap \neg D)$. However, as one can see, using the mergable test can effectively reduce the burden of invoking the tableau algorithms for subsumption queries. In this section, we will explain the definition of a pseudo model and the algorithm of mergable test [15].

2.2.1 Flat pseudo model for Abox reasoning

As mentioned in section 1.2.3.4, to realize an individual a, in other words, to compute the direct types of the individual a for a given subsumption lattice of the concepts $D_1,...,D_n$, it is required to perform a sequence of Abox consistency tests for $A_{D_i} = A \cup \{a : \neg D_i\}$. Therefore, supplying a cheap but sound mergable test for

the focused individual a and sets of concept terms $\neg D_i$ is the main purpose of the flat pseudo model technique.

(1) Pseudo model for an individual a.

We assume that Abox A is consistent, and there exists a non-empty set of completions C. A completion $A' \in C$. An individual pseudo model M for an individual a in A is defined as the tuple $< M^A, M^{\neg A}, M^{\exists}, M^{\forall} >$ w.r.t. A' and A using the following definitions:

$M^A = \{A \mid a : A \in A'\}$

$M^{\neg A} = \{A \mid a : \neg A \in A'\}$

$M^{\exists} = \{R \mid a : \exists R.C \in A'\}$

$M^{\forall} = \{R \mid a : \forall R.C \in A'\}$

(2) Pseudo model for a concept D

First, we define a label set $L_A(a)$ for an individual a. $L_A(a)$ is a set of all concept terms from all concept assertions for a in a completed Abox A'. Let D be a concept and A be the Abox : $A = \{a : D\}$, the flat pseudo model M for D consist of the following cach sets:

$M_D^A = \{D \mid D \in L_A(a)\}$

$M_D^{\neg A} = \{D \mid \neg D \in L_A(a)\}$

$M_D^{\exists R} = \{R \mid \exists R.C \in L_A(a)\}$

$M_D^{\forall R} = \{R \mid \forall R.C \in L_A(a)\}$

The mergable test for flat pseudo models M_1 and M_2 is to check whether there are some interactions:

For atomic concepts: $((M_D^{A_1} \cap M_{\neg D}^{A_2} \neq \phi) \vee (M_D^{A_2} \cap M_{\neg D}^{A_1} \neq \phi))$

For role successors: $((M_{\exists R}^{A_1} \cap M_{\forall R}^{A_2} \neq \phi) \vee (M_{\exists R}^{A_2} \cap M_{\forall R}^{A_1} \neq \phi))$

As discussed before, Tbox subsumption test can be preceded by the mergable test, and the realization of an Abox can also be preceded by the mergable test for the individual and the concept. For example, to test whether D is the type of individual a, it is sufficient to test whether a's pseudo model is mergable with $\neg D$'s. If they are mergable, then D is not the type of individual a.

However, the mergable test is only a sound but incomplete algorithm. In order to have a better understanding of the pseudo model technique, we will discuss the soundness and completeness problem in description logics.

2.2.2 Soundness and completeness

In descriptions logics, a decision procedure solves a problem with YES or NO answers to: $KB \mid -_i \alpha$, whether sentence α can be derived from the set of sentences KB by procedure i [16]. Thus, we can assume that the mergable test is regarded as a decision procedure which solves a problem with YES or NO answers, since it can be treated as to decide whether the focused pseudo models are mergable or not.

Soundness: Procedure i is sound if whenever procedure i proves that a sentence α can be derived from a set of sentences KB ($KB \mid -_i \alpha$), then it is also true that KB entails α ($KB \mid = \alpha$). A procedure is sound means no wrong inference is drawn from the knowledge base using this procedure. However, sometimes, a sound procedure may fail to find the solution in some cases, even though there indeed exists one. Therefore, it leads the notion of completeness.

Completeness: The procedure i is complete if whenever a set of sentences KB entails a sentence α ($KB \mid = \alpha$), then procedure i proves that α will be derived from KB ($KB \mid -_i \alpha$). In another word, the complete procedure can draw all the correct inferences from the knowledge base. However, a complete procedure may claim to have found a solution in some cases, when there is actually no solution. Therefore, we can say that for the YES/NO questions, if the procedure is sounds, and the answer we get is YES, then we can trust that. On the other hand, if the procedure is complete, and the answer we get is NO, we can trust it.

However, it is not ensured that we can find sound and complete algorithms to solve problems in any case. In fact, there exist many sound and incomplete algorithms which are considered as good approximations of problem solving, because they can simplify the procedure to find the solution by reducing the computational complexity. In our case, the pseudo model techniques we used are also sound and incomplete. Therefore, for the mergable test for the subsumption

question $A_1 \subseteq A_2$, where we assume that A_1 and A_2 are both complex concepts, we consider the following four conditions:

$(M_D^{A_1} \cap M_{\neg D}^{\neg A_2} = \phi)$
$(M_D^{\neg A_2} \cap M_{\neg D}^{A_1} = \phi)$
$(M_{\exists R}^{A_1} \cap M_{\forall R}^{\neg A_2} = \phi)$
$(M_{\exists R}^{\neg A_2} \cap M_{\forall R}^{A_1} = \phi)$

If all of them are true, which means there is no interaction between the pseudo models, they can be merged and the conjunction of concept terms $A_1 \cap \neg A_2$ is satisfiable, which means A_1 is not subsumed by A_2. If one of the condicitons is false, which means that the pseudo models of A_1 and $\neg A_2$ have an interaction, the tableaux prover must be used to test the satisfiablility of the conjunction. Therefore, it is easy to see that for this sound and incomplete algorithm, if there is no interaction between the pseudo models, we do not need to pursue the expensive tableaux algorithm to test the satisfiability problem, which largely reduces the computational complexity.

For example:

(1)Tbox: $C = A$ $\qquad\qquad$ $D = A \cap B \cap E$

Query:? $C \supseteq D$

Mergable test: ? $C \supseteq D$ is equivalent to test whether $D \cap \neg C$ is unsatisfiable.

So we should test whether the pseudo model of $\neg C$ and D are mergable.

$$M_D = \begin{matrix} M_D^A = \{A, B, E\} \\ M_D^{\neg A} = \phi \\ M_D^{\exists} = \phi \\ M_D^{\forall} = \phi \end{matrix} \qquad\qquad M_{\neg C} = \begin{matrix} M_{\neg C}^A = \phi \\ M_{\neg C}^{\neg A} = \{A\} \\ M_{\neg C}^{\exists} = \phi \\ M_{\neg C}^{\forall} = \phi \end{matrix}$$

Test : Whether $(M_D^A \cap M_{\neg C}^{\neg A})$, $(M_D^{\neg A} \cap M_{\neg C}^A)$, $(M_D^{\exists} \cap M_{\neg C}^{\forall})$ and $(M_D^{\forall} \cap M_{\neg C}^{\exists})$ $= \phi$.

One of them has an interaction, because $M_D^A \cap M_{\neg C}^{\neg A} = \{A\}$. Generally speaking, we can not get the conclusion that $C \supseteq D$ because the mergable test is incomplete. However, each of the concepts D and $\neg C$ has only one possible pseudo model as shown above, so we can say that $C \supseteq D$.

Now, we will see another example to illustrate why the mergable test is incomplete.

(2) Tbox: $C = A \cap B$ $\qquad\qquad$ $D = A \cup B \cup C$

Query:? $C \supseteq D$

Mergable test: ? $C \supseteq D$ is equivalent to test whether $D \cap \neg C$ is unsatisfiable.

The pseudo models for D and $\neg C$ are as follows:

$$M_{D_1} = \begin{matrix} M_{D_1}^A = \{A\} \\ M_{D_1}^{\neg A} = \phi \\ M_{D_1}^{\exists} = \phi \\ M_{D_1}^{\forall} = \phi \end{matrix} \qquad M_{D_2} = \begin{matrix} M_{D_2}^A = \{B\} \\ M_{D_2}^{\neg A} = \phi \\ M_{D_2}^{\exists} = \phi \\ M_{D_2}^{\forall} = \phi \end{matrix} \qquad M_{D_3} = \begin{matrix} M_{D_3}^A = \{C\} \\ M_{D_3}^{\neg A} = \phi \\ M_{D_3}^{\exists} = \phi \\ M_{D_3}^{\forall} = \phi \end{matrix}$$

$$M_{\neg C_1} = \begin{matrix} M_{\neg C_1}^A = \phi \\ M_{\neg C_1}^{\neg A} = \{A\} \\ M_{\neg C_1}^{\exists} = \phi \\ M_{\neg C_1}^{\forall} = \phi \end{matrix} \qquad M_{\neg C_2} = \begin{matrix} M_{\neg C_2}^A = \phi \\ M_{\neg C_2}^{\neg A} = \{B\} \\ M_{\neg C_2}^{\exists} = \phi \\ M_{\neg C_2}^{\forall} = \phi \end{matrix}$$

If we choose M_{D_1} to take the mergable test with $M_{\neg C_1}$, there is an interaction between these two pseudo models, as we already discussed above, since it is incomplete, we can not trust it. The reason is obvious that as long as there exists a disjunction in a concept's definition, it will lead incompleteness, because during

the satisfiability test, RACER generates only one pseudo model for each concept and each individual no matter whether they have disjunctions in their definition or not. As to the above's example, due to the disjunction in D, there exist three pseudo models for D. However, for the mergable test, only the one generated during the satisfiability test will be used. Therefore, it does not contain complete information about D.

We now discuss the individual realization [15]. Let *a* be an individual mentioned in a consistent Abox A w.r.t. a Tbox T, $\neg C$ be a satisfiable concept, M_a and $M_{\neg C}$ donote the pseudo model for the individual *a* and the concept $\neg C$ respectively. If the mergable test returns true, which means it does not have an interaction at all, the Abox $A \cup \{a : \neg C\}$ is consistent, so *a* is not an instance of C.

In general, individuals in an Abox only belong to a few named concepts, therefore, the proof of contradiction will not derive a clash, which means in most of the cases, *a* is not an instance of C. As for the mergable test, if *a* is not an instance of C, there is no interaction between those two pseudo models. Since the mergable test is sound and incomplete, we can trust the "YES" result. We do not need to perform the expensive tableaux algorithm most of the time.

Now, let us see an example about the individual realization.

Tbox: $C \equiv \exists R.Y$

Abox: $(a,b):R, \quad b:X$

Query: ? $a:C$

We build the pseudo model for a and $\neg C$

$$M_a = \begin{matrix} M_a^A = \phi \\ M_a^{\neg A} = \phi \\ M_a^{\exists} = \{R\} \\ M_a^{\forall} = \phi \end{matrix} \qquad M_{\neg C} = \begin{matrix} M_{\neg C}^A = \phi \\ M_{\neg C}^{\neg A} = \{C\} \\ M_{\neg C}^{\exists} = \phi \\ M_{\neg C}^{\forall} = \{R\} \end{matrix}$$

So there is an interaction between M_a and $M_{\neg C}$, the two pseudo models are unmergable. Due to the incompleteness of the mergable test, we need to do the tableaux test. However, if we continue recursively, we can see that we can still succeed with the mergable test. From $(a,b):R$, $b:X$ and $C \equiv \exists R.Y$ we can perform the mergable test of M_b and $M_{\neg Y}$.

$$M_b = \begin{matrix} M_b^A = \{X\} \\ M_b^{\neg A} = \phi \\ M_b^{\exists} = \phi \\ M_b^{\forall} = \phi \end{matrix} \qquad M_{\neg Y} = \begin{matrix} M_{\neg Y}^A = \phi \\ M_{\neg Y}^{\neg A} = \{Y\} \\ M_{\neg Y}^{\exists} = \phi \\ M_{\neg Y}^{\forall} = \phi \end{matrix}$$

So there is no interaction between M_b and $M_{\neg Y}$, they are mergable which can lead to the conclusion that *a* is not an instance of C. However, this kind of deep pseudo model technique is based on a tree instead of a graph. Because it recursively applies the completion rule based on the previous one, it would have problems with graphs that are not trees.

2.3 Our choice—pseudo model techniques

After illustrating the basic concept of the two techniques to combine the description logics reasoners with databases, it is easy to draw the conclusion that we prefer to apply the pseudo model techniques.

As for the precompletion technique, by applying the precompletion rules, it removes the role assertions from the Abox and keeps only concept assertions. The definition of a precompletion for a knowledge base $< T, A >$ is given in a procedural way, as a new KB $< T, A_{pc} >$ is created where the Abox is extended using the syntactic precompletion rules as long as they are applicable. Because of the nondeterminism of the rules ($R_{\exists}, R_{\leq n}$ rules), several precompletions can be generated. It is sound and complete. However, as discussed in section 2.1.3, it may generate an exponential number of precompletions, which means if the computational complexity is a critical concerned for a system, it is not a very good choice to take.

Compared to the precompletion technique, instead of being based on a whole set of precompletion rules, the pseudo model technique is only based on one completion while generating pseudo models for individuals and concepts. Just because of that, it is sound but incomplete. However, its minimal computational overhead and the avoidance of any indeterminism overweigh its incompleteness, especially when efficiency is the main concern for a system.

Our objectives for building the LAS (Large Abox Store) are to make good use of both reasoning services of description logic systems and efficient data management of database systems. Dealing with a large Abox, one of the difficulties is to retrieve individuals given a query concept. If we do the instance-checking test one by one, it will involve a lot of work. By observing the characteristics of an Abox, to perform the mergable test between pseudo models for all individuals in an Abox and pseudo model for the given concept can largely avoid the expensive tableau test.

On the other hand, the description logics reasoner RACER generates all the pseudo models for the concepts and individuals while loading the file into itself. To be more precise, pseudo models for concepts are generated while RACER performs the Tbox satisfiability tests, and pseudo models for individuals are generated while RACER test the Abox satisfiability. Therefore, no matter which technique we use, RACER will have the pseudo models computed in its initialization phase. Hence, it is a good reason to apply the pseudo model technique.

3. The Large Abox Store (LAS)

LAS is a Java application that combines databases with the description logics reasoner RACER.

In general, we designed our system into two modes— lazy and eager mode. Both of the two modes will first connect to RACER, and get the basic information for initialization, then store the information into the database. In the lazy mode, LAS does reasoning on demand, which means whenever a query is posed, it will ask RACER to get the result, and then return the answer to users. Afterwards, it will store the results in the database for future use. For the eager mode, LAS will try to convert description logics reasoning into equivalent DBMS SQL queries, so that the reasoning efforts by RACER can be reduced.

For the initialization, no matter what mode has been chosen, LAS will first load the file into RACER, and let RACER check the satisfiability of Tbox and Abox. Then we store the Tbox taxonomy and Abox assertions into the database. Afterwards, when the queries are posed, we check the database, and apply SQL to extract results. If the information about the specific incoming query is not complete, we will send the query to RACER. After RACER has finished its reasoning, the result is sent to both the user and its corresponding database, and the completeness of information is recorded in the database for the future use. As

we rely on RACER to perform the final checking, in total, our system can answer queries correctly and completely.

3.1 System Architecture

3.1.1 Four Main Components of LAS

LAS consists of four components (see also Figure 3.1):

- a database such as

  Orcle, JdataStore Explorer or MySQL, which can be accessed through JDBC.

- a resoner: RACER

  Because LAS applies the pseudo model technique, a technique which is only compatible with RACER, we only chose RACER as the description logics reasoner.

- an ontology :

  A set of objects and axioms in the form of either OWL or LISP

- User Interface:

  User portal to initialize and execute queries.

3.1.2 Three Interfaces of LAS

These four components can be divided into three main groups according to their different functional responsibility:

LASUser:

Mainly responsible for the display and interaction with users including user initialization and queries. For the user initialization, it is partitioned into two

different parts according to the status of users. For the new users who use LAS for the first time, we create the accounts for them and build the database and create all the tables. As for old users, who had already logged in to LAS before and have their records in the corresponding database, we do not need to generate a new database for them. LAS will open the database connection using JDBC. Afterwards, users can pursue their queries based on the existing information in their database.

LASRACER:

The LASRACER interface is responsible for communicating with RACER. It consists of the TCP-based client for RACER, and all classes are based on or related to Racer reasoning results. On the other word, it passes the results to LAS from RACER and vise versa.

LASDatabase:

LASDatabase interacts with the database, and generates the queries for the database.

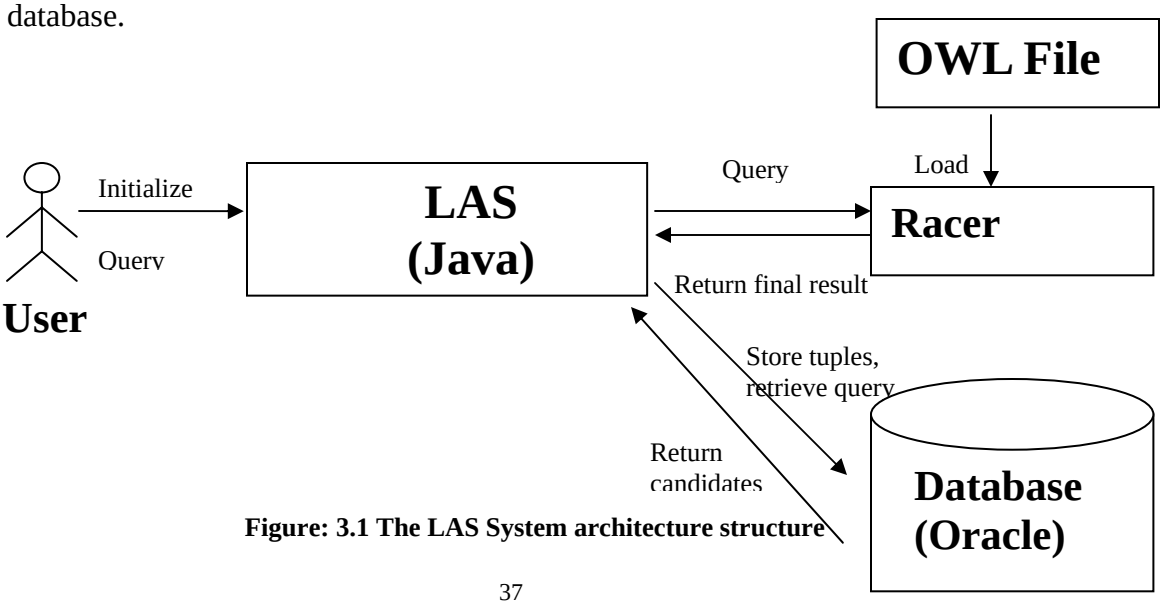


Figure: 3.1 The LAS System architecture structure

3.2 Database management

One of the main features of the LAS System is that it can reuse the information retrieved from RACER no matter whether the RACER server has been shut down or not. Therefore, for different ontology files, even though they share the same schema, we will create different databases for them so that they are independent and will not overwrite the other ontology files.

3.3 Database schema

In order to convert the whole ontology into the database tables, we define a database schema as follows:

1. Individual (individualname String,
   inpseudomodelid Integer,
   indcomplete Boolean)

2. IndividualPseudoModel (inpseudomodelid Integer,
   ina String,
   innota String,
   inexistence String,
   inuniversial String,
   unique Boolean)

3. Description (descriptionname String,
   despseudomodelid Integer,
   negationdespseudomodelid Integer,
   descomplete Boolean)

4. DescriptionPseudoModel (despseudomodelid Integer,
   desa String,
   desnota String,
   desexistence String,
   desuniversial String,
   unique Boolean)

5. DesParent (desparent String,
   Deschildren String)

6. DesAncestors (desancestors String,
   dessendants String)
7. Tmp_Desparent (desparent String,
   deschildren String)

8. Tmp_Desancestors (tmp_desancestors String,
   tmp_desendants String)

9. TmpDesPM (des String,
   desa String,
   desnota String,
   desexistence String,
   desuniversial String,
   uniquename Boolean)

10. Inassertion (individualname String,
    descriptionname String,
    mostspecific Boolean)

11. RoleType (rname String)

12. Transitiverole (rname String)

13. Rsynonyms (rname1 String,
    rname2 String)

14. RAncestors (rancestors String,
    rdecendants String)

15. RParents (RParents String,
    RChildren String)

16. RoleAssertion (individual1name String,
    individual2name String,
    rolename String,
    complete1 Boolean,
    complete2 Boolean,
    rolecomplete Boolean)

17. Synonyms (descriptionname1 String,
    descriptionname2 String)

18. SymmetryRole (rname String)

19. Tmp_RAncestors (RAncestors String
    RDescendants String)

These nineteen tables are partitioned into three groups: those that represent the ontology, those that deal with the pseudo models, and those that represent the taxonomy of the ontology. Each group deal with different RACER queries. Please refer to Appendix A for the specification of the used RACER query commands.

The first group consists of the tables Individual, Description, RoleType, TransitiveRole and Rsynonyms, which store the Tbox information and tables InAssertion and RoleAssertion, which store the Abox information.

Individual (individualname, inpseudomodelid, indcomplete)

The attribute individualname stores the individual's namespace which is obtained from RACER, inpseudomodelid stores the pseudo model ID of the corresponding individual, while indcomplete is a flag which indicates the completeness status of the information about this individual. If the information about the individual is complete, which means all the atomic concepts of which the individual is an instance are known and already stored in the table InAssertion, then indcomplete is set to T. Therefore, when the query "individual-types" [Appendix A] is excuted, LAS first checks this column and then decides whether to ask RACER for further reasoning or just extract the information from the database.

Description (descriptionname, despseudomodelid, descomplete)

The descriptionname column stores the namespace of the description and despseudomodelid stores the pseudo model ID of the corresponding description.

The descomplete is the status flag for information completeness about the description. For example, for the retrieve query, which means to retrieve all the individuals from an Abox that are instances of the specified concept, LAS checks the descomplete flag first. If it is True, which means the information is complete, LAS just executes a SQL query and retrieves the instances as result. If the status flag is False, it will first perform the pseudo model mergable test in the database, and then forward the known candidates to RACER and let RACER perform the final check. It is easy to see the advantages. For the non-complete information, the database acts as a filter for RACER. Due to the pseudo model technique, all the pseudo models information is already stored in database; the mergable test can be performed by the database. Due to the efficiency of SQL queries, the mergable test is fast, and reduces the number of possible candidates. It only sends the most possible candidates to RACER. The advantage is especially obvious for a large Abox. In general, the number of the instances of a specific concept will not reach the total number of the individuals in the Abox. Conversely, most of the individuals do not belong to a specific named query concept. Therefore, as a filter, LAS reduces the workload of RACER so that RACER does not need to check all the individuals whether they are instances of that specific concept. For the complete information, we do not need to interact with RACER, we can just execute a simple SQL query and get the correct answers efficiently.

RoleType (rname)

TransitiveRole(rname)

Rsynonyms (rname1, rname2)

The RoleType table stores all the roles defined in a Tbox, and TransitiveRole stores the corresponding transitive roles. A tuple from the Rsynonyms is a pair of equivalent role names. This tuple refer to the same role by using different names. At the first glance, this design is a bit odd, because according to the database three normalization theory, we can design the table RoleType as RoleTypev2(rolename, rtransitive, rsymmetry,rinversename) (v2 stands for another verstion of RoleType table), so that to mark the feature of the specific role. For example, if the given role is transitive, the value for the attribute 'rtransitive' can be set to True. The data type of rtransitive and rsymmetry would be Boolean, while the data type of rinversename is String, which stores the role name of the inverse role for roles. However, according to the characteristics and implementation of our system, the information about the properties of a role, such as transitive, symmetric and the corresponding inverse role has to be retrieved from RACER by separate queries. In other words, we have to use different RACER queries to extract all the information about roles and then parse the result and store it into the database. For example, we first use the query "all-roles" to get all roles and features from the specified Tbox. Afterwards, LAS calls other queries such as "all-transitive-roles" and "all-symmetry-roles" which return all the transitive roles and symmetric roles respectively. As we already discussed, if we store all this information into one table, we have to combine all the results of these queries about roles: "all-roles",

“all-transitive-roles” and “all-symmetry-roles”. On the other hand, the information about transitive roles and symmetric roles is only used when we need to query the relationship between individuals; we need to propagate the RoleAssertion table according to the focused role’s attribute (e.g. Rtransitive, Rsymmetry). Therefore, instead of checking the RoleTypev2 table, we can check whether the focused role exists in the tables Rtransitive or Rsymmetry, and decide which action to perform in order to propagate the RoleAssertion. As for this design, it is not only more direct to store the information of the attribute of roles because we get these results separately from RACER, but also it will reduce the use of database space. In general, there are actually not that many roles that are transitive or symmetric, so if RoleTypev2 were used in the schema, there would be a lot of tuples whose value for the attributes rtransitive or rsymmetry were False. Therefore, we finally chose to store this information separately using different tables. The table Rsynonyms stores the synonyms of a role including the role itself.

In the first group, the other two tables—InAssertion and RoleAssertion are used to store the information about the Abox assertions.

InAssertion (individualname, descriptionname, mostspecific)

It stores all concept assertions from a specified Abox. The individualname stores the name space of the individual of an assertion. The descriptionname stores the name space of the description of an assertion. Here, we only store atomic concept

assertions. The mostspecific attribute is a status flag which represents whether the value of description name is the most direct type of the individual. At the beginning, the value of ‘mostspecific’ is always set to False. If a query asks for the direct types of a specific individual, first, we will check this attribute to see whether the mostspecific attribute of the focused individual is T, if it is true, then we can just use the SQL query to extract the descriptionname from the InAssertion table. Otherwise, we will send the “individual-direct-types” query to RACER, and store the result of the direct types of this specific individual into the database, and at last, set the focused individual’s ‘mostspecific’ flag to T. Therefore, when a similar query is processed, we just need to query the database, instead of asking RACER to process the query.

RoleAssertion (individual1name, individual2name, rolename, complete1, complete2, rolecomplete)

It stores all role assertions from a specific Abox. The attributes individual1name, individual2name and rolename store the name space of individuals, description and role respectively, and those three together represents the structure of a role assertion. The attributes ‘complete1’, ‘complete2’ and ‘rolecomplete’ are status flags which indicate the completeness of the information about the individual1, individual2 and the role respectively. At first, all these three flag are set to be False.

Before explaining the usage of each status flag, we will first give some definitions about the terms in role assertion. A role assertion consists of three parts:

individual1, individual2, role, which means individual1 and individual2 are in the relationship of role. Given a role assertion (individual1, individual2, role), we have the following terms:

Individual fillers: individual2 is individual1's fillers with respect to the specified role.

Individual direct predecessors: individual1 is individual2's direct predecessors with respect to the specific role.

Related individuals: individual1 and individual2 are the related individuals with respect to the specific role.

Individual filled roles: The role is the individual filled role of individual1 and individual2.

Here, we discuss in detail the usage of the status flags. The 'complete1' flag records the completeness of information about individual fillers of a specific individual—individual1, and a specific role. For example, when a user wants to query all the individuals that are fillers of a role for a specified individual, we will first check complete1 of the focused individual and the given role. If its value is T, which means the individual fillers information for individual1 is already complete, we can just answer the query based on the database, otherwise, we will take other actions which depend on which mode the user has chosen. For the lazy mode, we will send the query "individual-fillers IN1 R" to RACER, and store all the fillers we get back from RACER into the database. At last, we set the 'complete1' flag of the given individual1 and role to T. For the eager mode, we

will propagate the RoleAssertion table based on the information in tables TransitiveRole and SymmetryRole, Therefore, we do not need to query RACER about the result.

The status flag complete2 records the completeness of the information about direct predecessors of a role for a specified individual. When querying the individual's direct predecessors, we first check the table RoleAssertion to see whether the tuple's 'complete2' value is T or F. If the 'complete2' value is T, which means the information about the direct predecessors of the given individual and specific role is complete, we can simply perform the SQL query and return the complete results to user. However, if 'complete2' is F, we have two ways to compute the final results. One way is to completely depend on RACER by sending the query "retrieve-direct-predecessors R IN". After we stored the result, the complete2 flag is set to be T to indicate that the information about the individual's direct predecessors is complete. The other way is based on the propagation of roles to compute the results in the database. The details of this query will be explained in the next section.

The status of 'rolecomplete' is to record the completeness of information about related individuals of a specific role. The basic idea is the same as for the first two status flags—complete1 and complete2. The details of the query will be discussed in the next section.

The second group of our schema, which deals with pseudo model techniques, consists of the tables IndividualPusedoModel, DescriptionPseuddModel and table TmpDesPM. The definition of pseudo model is introduced in Section 2.2.1.The attribute “unique” indicates whether this pseudo model is the only one pseudo model for the given individual (description). This information is provided by RACER.

For example, let us assume that M_{ind} =<{A B}, {C D E}, {R1 R2}, {R3}> and M_{des} =<{D B}, {E F}, {R4 R5 R6 R7}, {R1}>, where M_{ind}^{A} = {A, B}, while $M_{des}^{\neg A}$ = {B, D}. Therefore, we have to design a table that not only can store this information, but is also suitable for executing queries. In our system, we designed our pseudo model table in a way that we can use pure SQL to do the mergable test. We used pseudomodelid to identify pseudo models. Each pseudo model has only a unique ID. And then we store the value of M_{ind}^{A} into different rows, where each row contains the same ID, but each element of sets in M_{ind} (i.e. $M^{A}, M^{\neg A}, M^{\exists}, M^{\forall}$) requires one row.

According to the characteristics of the pseudo model structure and mergable test, the pseudo model can be divided into two groups: ($M^{A}, M^{\neg A}$) and ($M^{\exists}, M^{\forall}$), because the mergable test has to do the checking within these two groups separately, M^{A} will not be checked with $M^{\exists}$. A specific pseudo model has a unique ID and four sets: $M^{A}, M^{\neg A}, M^{\exists}, M^{\forall}$, each element of these sets occupies

one row. Among these four sets, there will be at least one set which contains the most number of elements. Let us assume the maximum number is n, which means the set has n elements. Hence, this specific pseudo model has n rows, each row contains the same ID, and for the sets whose number of the elements is less than n, we fill the value of its corresponding columns with NULL.

For the example given above, M_{ind} =<{A B}, {C D E}, {R1 R2}, {R3}> and M_{des} =<{D B}, {E F}, {R4 R5 R6 R7}, {R1}>, the pseudo models for M_{ind} and M_{des} are as follows, where the assigned pseudo model ID for M_{ind} is 1 and that for M_{des} is 2.

inpseudomodelid	Ina	innota	Inexistence	inuniversial	unique
1	A	C	R1	R3	T
1	B	D	R2	NULL	T
1	NULL	E	NULL	NULL	T

Figure 3.1: Pseudo model for M_{ind}

inpseudomodelid	Ina	innota	Inexistence	inuniversial	unique
	D	E	R4	R1	T
2	B	F	R5	NULL	T
2	NULL	NULL	R6	NULL	T
2	NULL	NULL	R7	NULL	T

Figure 3.2 Pseudo model for M_{des}

The information about pseudo models of individuals and descriptions can be retrieved through the RACER query “get-individual-pmodel <individual-name> <Abox-name>” [Appendix A], where <Abox-name> is optional and “get-description-pmodel <description-name> <Tbox-name>”, where <Tbox-name> is

optional. Currently, RACER will compute only one pseudo model for a particular individual (description). In other words, for a complex description with disjunctions or an individual which is the instance of a complex description with disjunctions, there exist more than one pseudo model, but RACER will compute only one pseudo model and return it. Therefore, after the mergable test, due to the disjunction, we still have to send the candidates to perform the final check. However, even though RACER returns only one chosen pseudo model, we propose another solution in case RACER could return all the pseudo models, so that we do not need to forward the candidates in case the description's definition does not contain a role. The results on database SQL query are the final results. If the definition of the description contains roles, then we will send the candidates to RACER to do the deep pseudo model mergable test.

As discussed in the previous chapter, the non-subsumption checking might be replaced by pseudo model mergable tests. As long as there is no interaction between the pseudo models of two selected concepts, we can conclude safely that there is no subsumption relationship between these two concepts. On the other hand, if all pseudo models of two selected concepts do not have role parts (existential and universal parts), and if and only if all these pseudo models have an interaction with one another, then these two selected concepts have a subsumption relationship. Herein, we get another idea for the mergable test which is based on the condition that if RACER can return the information about all the pseudo models of a specific concept or individual. The following is the schema of

the pseudo model table according to this method. We named it as IndPseudoModel_v2 and DesPseudoModel_v2.

IndPseudoModel_v2 (*inpseudomodelid* *Integer*,
indversion Integer,
ina *String*,
innota *String*,
inexistence *String*,
inuniversial *String*)

DesPseudoModel_v2(*despseudomodelid* *Integer*,
desversion Integer,
desa *String*,
desnota *String*,
desexistence *String*,
desuniversial *String*)

The other part (*italic type*) is the same as in the tables IndPseudoModel and DesPseudoModel we defined before, except that we add one more column indversion (desversion) to indicate the disjunction. Those who contain the same indpseudomodelid (despseudomodelid) but different indversion (desversion) are the pseudo models for the same individual (description) but are based on different disjuncts. Those who contain the same pseudo model id and same indversion, but a different value in other columns means they together represent one pseudo model, and the relationship among different rows is conjunction.

Therefore, to check whether an individual is a given concept's instance could be replaced by checking whether all the pseudo models of this individual have interaction with all the pseudo models of the given concept.

For example, *a*: $(A \cap B \cap C \cap D) \cup (A \cap F) \cup (D \cap Q)$, we want to check whether *a* is an instance of $A \cap D$. We store the pseudo models of *a* and the complex description $\neg A \cup \neg D$ as follows. The details of the query are discussed in the next section.

Inpseudomodelid	indverstion	Ina	innota	Inexistence	Inuniversial
1	1	A	NULL	NULL	NULL
1	1	B	NULL	NULL	NULL
1	1	C	NULL	NULL	NULL
1	1	D	NULL	NULL	NULL
1	2	A	NULL	NULL	NULL
1	2	F	NULL	NULL	NULL
1	3	D	NULL	NULL	NULL
1	3	Q	NULL	NULL	NULL

Figure 3.3 InPseudoModel_v2 for individual a

Despseudomodelid	desversti on	desa	desnota	desexistence	Desuniversi al
1	1	NULL	A	NULL	NULL
1	2	NULL	D	NULL	NULL

Figure 3.4 DesPseudoModel_v2 for complex description: $\neg A \cup \neg D$

Table TmpDesPM is a temporary table. It stores the pseudo model of a specific complex concept when the query about a complex concept is processed. After the query has finished, information about the pseudo model from TmpDesPM table is deleted.

We like to point out that although it is a solution if all pseudo models are given by RACER, it is unrealistic to compute all the pseudo models in advance.

The third group of the database schema, which mainly deals with the taxonomy of the ontology, consists of DesParent, DesAncestores, Synonyms, Tmp_Desparent,

Tmp_Desancestors which represents the taxonomy of concepts in the Tbox, and RParents, RAncestors and Rsynonyms which describe the taxonomy of roles in the Tbox. Table Tmp_Desparent, Tmp_Desancestors are the temporary tables for generating the table DesAncestores from table DesParent. Please refer to the other thesis [41] for the details.

3.4 Database query--Abox query

Our system can deal with various queries. It covers most of the query types supported by RACER [47]. Abox queries focus on answering queries concerned on concept assertions and role assertions. Compared with Tbox queries, they base on different database tables, and different RACER commands to communicate with RACER.

3.4.1 Individual queries

The Abox queries that are supported by LAS are query_indivdiual_types, query_individual_direct_types, query_retrieve, which are about the concept assertions of the Abox, and query_indivdiual_fillers, query_direct_predecessors, query_individual_filled_roles, query_related_individual which are mainly concerned with the role assertions of the Abox. In the following, we will discuss each of these queries in detail.

3.4.1.1 query_individual_types

To query individual types is to get all atomic concepts of which the individual is an instance. In our system, we first check the ‘indcomplete’ flag of the focused

individual to see whether it is true or false. If it is true, then we simply use a SQL query to retrieve the existing information from the database. Otherwise, we will ask RACER to compute the result and store it in the database and set the indcomplete status flag to be True for the next time's use. The pseudo code of this query is as follows:

```
Query_individual_types (String ind, Reasoning reasoner,Connection c)
{

    indcomplete =  "Select Individual.indcomplete
                    from Individual
                    where Individual.indivdiualname = 'ind'";

    if (indcomplete== True){
        individual_types = "select distinct descriptionname
                            from InAssertion
                            where indivdiualname = 'ind'";

    }else {
        individual_types = reasoner.get_individual_types (ind);
        store (individual_types, Connection c, Table InAssertion); store the results of
                                                                   ; individual_types into database
     update (Table Individual, String indcomplete, String ind); update the indcomplete flag of the
                                                              ; specific individual 'ind' in the
                                                              ; table Individual.
    }
    return individual_types;
}
```

3.4.1.2 query_individual_direct_types

To query the individual direct types is to get the most-specific atomic concepts of which an individual is an instance. It is a subset of the previous query. Due to our database schema, we designed the table InAssertion with a column to describe whether this concept is the most specific type of a given individual. This query is similar to query_individual_types, just this query is based on the table InAssertion, while query_individual_types combines table Individual and InAssertion. The detail algorithm of this query is as follows.

```
Query_individual_direct_types (String ind, Reasoning reasoner,Connection c)
{
    specific  = "Select InAssertion.specific
                        from InAssertion
                        where InAssertion.individualname = 'ind'";

    if (specific== True){
    individual_direct_types = "select distinct descriptionname
                                from InAssertion
                                where indivdiualname = 'ind' and mostspecific = 'T'";
    }else {
    individual_direct_types = reasoner.get_individual_direct_types (ind);
    store (individual_direct_types, Connection c, Table InAssertion);  store the results of
                                                                        ; individual_diret_types into
                                                                        ;  the database
    update (Table InAssertion, String mostspecific, String ind)   ; update the mostspecific flag of
                                                                   ; the specific individual 'ind'  in
                                                                   ; the table InAssertion
    }
    return individual_direct_types;
}
```

3.4.1.3 query_retrieve

The retrieve query is to get all individuals from an Abox that are instances of a specified concept. Here, the concept can be atomic or complex. Our system first checks the input concept. If it is atomic, we will check the table Description to see whether the information about the focused concept is complete. If 'descomplete' has the value 'True', which means all the individuals of this concept are already stored in the table, we can just execute the query based on the database. Otherwise, we have to perform the mergable test, then compute the possible candidates and send them to RACER to perform the final check. After getting the return results, we will store them into the database and set this description's descomplete status flag to be True. If the 'descomplete' is False, we first have to get the pseudo model of this complex description, and then do the mergable test

and exonerate the non-instances of the focused complex description. After that, the other steps are similar to those for atomic concepts.

```
Query_retrieve (String concept, String concept_status, Reasoning reasoner, Connection c)
{

    if (concept_status == “atomic”){
            descomplete  = “Select Description.descomplete
                           from Description
                           where Description.description = ‘concept’”;
            if (descomplete== True){
                    retrieved_indivdiuals = “select distinct individualname
                                            from InAssertion
                                            where descriptionname = ‘concept’”;
                    }else {
                             Vector individual_candidates = mergable_test (String concept);
    Retrieved_indivdiuals=  reasoner.concept_instances(concept, Abox, individual_cadidates)
    ; send the candidates to RACER and get the final results

    store (retrieved_individuals, Connection c, Table InAssertion); store the results of
                                                                    ; retrieved_indivdiuals into the
                                                                    ; database
    update(Table Description, String descomplete, String concept)    ;update the descomplete flag
                                                                ;of the specific description
                                                                ;‘concept’  in the table Description
                            }
    }else {
    ; for the complex concept
    ;negate the complex concept and get this negation’s pseudo model
    String neg_concept = "(not " +concept+ ")";
    Vector des_neg_psmodel = reasoner.get_cnp_psmodel(neg_concept);

    Vector individual_candidates = mergable_test (String concept);

    ; send the candidates to RACER and get the final results
    retrieved_indivdiuals= reasoner.concept_instances(concept, Abox, individual_cadidates)

    ; here we do not store back the retrieved_individuals of the complex description into the
    ;database before in the table InAssertion and table Description we only store the information
    ;about atomic concepts
    }
    return retrieved_individuals;
}
```

Now, we present the mergable test which is implemented in SQL.

```
 mergable_test (String concept)
{
    define candidates as ResultSet;
    define sql1 as
    “select distinct individual.individualname from individual
      where individual.inpseudomodelid in
```

```
      (select individualpseudomodel.inpseudomodelid
      from individualpseudomodel,description,descriptionpseudomodel
      where description.descriptionname = ' concept ' and
      description.negationdespseudomodelid=descriptionpseudomodel.despseudomodelid
       and ( (individualpseudomodel.ina=descriptionpseudomodel.desnota and
              individualpseudomodel.ina<>'NIL')
            or (individualpseudomodel.innota=descriptionpseudomodel.desa and
              individualpseudomodel.innota<>'NIL')
            or (individualpseudomodel.inexistence=descriptionpseudomodel.desuniversial and
               individualpseudomodel.inexistence<>'NIL')
            or (individualpseudomodel.inuniversial=descriptionpseudomodel.desexistence and
               individualpseudomodel.inuniversial<>'NIL')))";"

     candidates  = c.execute(sql1)"

     return candidates;
}
```

As discussed in the previous section, we also designed the pseudo model schema for the case that we can get all possible pseudo models of a given concept or individual. In this case, if all the pseudo models do not contain the existence part and universal part, which mean it is in propositional logic, then we can trust the answer and do not need to send the candidates to RACER. If they do have role related information in the pseudo models, we will send the candidates to RACER for further reasoning.

```
mergable_test_v2 (String concept)
{
     table1= " select individualpseudomodel.inpseudomodelid, indversion, desversion
                 from individualpseudomodel,description,descriptionpseudomodel
                 where description.descriptionname = ' concept ' and
                description.negationdespseudomodelid=descriptionpseudomodel.despseudomodelid
                and ((individualpseudomodel.ina=descriptionpseudomodel.desnota and
                       individualpseudomodel.ina<>'NIL')
                    or (individualpseudomodel.innota=descriptionpseudomodel.desa and
                       individualpseudomodel.innota<>'NIL')
                   or (individualpseudomodel.inexistence=descriptionpseudomodel.desuniversial and
                       individualpseudomodel.inexistence<>'NIL')
                   or (individualpseudomodel.inuniversial=descriptionpseudomodel.desexistence and
                       individualpseudomodel.inuniversial<>'NIL'))";"

                int sumiver = get_count (Table individualpseudomodel.indversion, String ind);
                int sumdver = get_count (Table descriptionpseudomodel.desversion, String concept);
```

```
        int row_number = get_row_number (table1);

        if row_number = sumiver* sumdiver;
        { retrieved_indivdiual =  "select distinct individual.individualname
                                    from individual, table1
          where individual.individualpseudomodelid = table1.indivdiualpseudomodelid"
        }
        return retrieved_individuals;
}
```

This algorithm is based on the assumption that if all of the pseudo models of an individual have an interaction with all possible pseudo models of a description, then this individual is an instance of the given description. If the definition of a description contains m disjunctions with n disjuncts, then this description has at most m^n pseudo models.

Thus, in the algorithm presented above, the column "sumiver" computes the number of the indversion, which means the number of the pseudo models of this focused individual, while sumdver calculates the number of the desversion, which means the number of the pseudo models of this focused description. Let us assume that sumiver is m, and sumdver is n. There are m*n possible combinations of the pseudo models of the individual and pseudo models of the description. If all these possible combinations clash, which means they all have an interaction, we can conclude that the subsumption relationship exists. Hence, the individual is an instance of the given description.

3.4.2 Role Assertion Queries

The following queries dealing with role assertions are implemented in two ways. One version is mainly relying on RACER, as long as we do not have complete information, we have to send the original query to RACER and ask for the results. The other version is mainly based on SQL to generate the complete information of the focused query and extract the results based on the newly generated RoleAssertion table.

3.4.2.1 query_individual_fillers

For the first method, we show the algorithm of query_individual_fillers below:

```
Query_individual_fillers (String ind, String role, Reasoning reasoner,Connection c)
{
     complete1  = "Select complete1
                  from RoleAssertion
                  where individual1name = 'ind' and rolename = 'role'"

if (complete1== True){
     individual_fillers = "select distinct individual2name
                          from RoleAssertion
                          where indivdiual1name = 'ind' and rolename = 'role'"

     }else {
               individual_fillers = reasoner.get_individual_fillers (ind, role);
               store (individual_fillers, Connection c, Table RoleAssertion); store the results of
                                                                              ;individual_fillers into
                                                                              ; the database
     update (Table RoleAssertion, String complete1, String ind); update the complete1 flag of the
                                                                 ; specific individual 'ind'
                                                                 ; and role 'role' in the table
                                                                 ; RoleAssertion
     }
     return individual_fillers;
}
```

The queries "query_direct_predecessors" and "query_individuals_filled_roles" are similar, they just check different status flags to verify whether the information of the posed query is complete. In detail, we check the status flag 'complete2' for

“query_direct_predecessors” and ‘rolecomplete’ for
“query_individuals_filled_roles”.

Now, we consider the second method to query these role assertions, which is mainly based on SQL. Below, we will illustrate this method for these queries concerning role assertions.

The procedure of dealing with the complete information is the same as in the first method. However, if the information about the individual’s fillers is not complete, we will not send the original query directly to RACER. Instead, we will first get the descendents of the focused role, and select the tuples whose individual1name is equal to the name of given individual. As we know, the assertions of a particular role’s descendents are also the assertions of this role. For example, let role has_son be the descendents of has_child. Therefore, if (a, b): has_son is known, we can also conclude that (a, b): has_child holds. Therefore, we can store these tuples into table RoleAssertion in order to finish the first step of propagating role assertions. Second, we will check whether the focused role is transitive. If it is transitive, we have to continue to propagate the role assertions. Here, we create a temporary table Tmp_RoleAssertions to generate the partial transitive closure of a specific role. It is called partial transitive closure because we generate this closure based on the focused individual1 and a specific role. In other words, the transitive closure of a specific role is complete with respect to both the given individual and role, but not complete with respect to a role itself. Hence, we call

it a partial transitive closure. At last, we insert these new generated tuples into RoleAssertion, and set the complete1 flag to “True”.

```
Query_individual_fillers (String ind, String role, Reasoning reasoner,Connection c)
{

    complete1  = “Select complete1
                    from RoleAssertion
                    where individual1name = ‘ind’ and rolename = ‘role’”;

    if (complete1== True){
        individual_fillers = “select distinct individual2name
                        from RoleAssertion
                        where indivdiual1name = ‘ind’ and rolename = ‘role’”;
        }else {
        roledecendants=get_roledecendants(role);
        descendants_assertion = “select * from RoleAssertion
                        Where rolename = ‘roledecendants’ and individual1name = ‘ind’”;
            ; store the assertion of descendants of the specific ‘role’ into RoleAssertion table.
        store_roleassertion (descendants_assertion, Connection c, Table RoleAssertion);
        if( is_transitive_role ( role)){
        define transitive_roleassertion as ResultSet;
        transitive_roleassertion = “select * from RoleAssertion
                                where individual1name = ‘ind’ and rolename = ‘role’”;
        create_table (Tmp_RoleAssertions);
        store_roleassertion(transitive_roleassertion, Connection, Table RoleAssertion);

        propagate_transitiveclosure (table Tmp_RoleAssertions, Connection c);
        ; store back the transitive closure into table RoleAssertion.
        store (Table Tmp_RoleAssetions, Connection c, Table RoleAssertion);
    }

    individual_fillers = “select individual2name from RoleAssertion
                    where individual1name = ‘ind’ and rolename= ‘role’”;
                    update (Table RoleAssertion, String complete1, String ind);
                    ;update the complete1 flag of the specific individual ‘ind’ and role ‘role’
                    ; in the table RoleAssertion
    }
    return individual_fillers;
}
```

It is known that dealing with the transitive closure is not possible in traditional SQL [17], and hence this is a problem in relational database management systems. However, in recent DBMS, some features are offered to overcome this problem. In our system, for propagating the transitive closure, we used the CONNECT BY

PRIOR and START WITH clauses provided by Oracle [18] in the SELECT statement to write the recursive query.

```
propagate_transitiveclosure (table Tmp_RoleAssertions, Connection c)
{
        "select distinct substr(paths,2,instr(paths,'/',1,2)-2)INDIVIDUAL1,
         substr(paths, instr(paths,'/', -1,1)+1,length(paths)-instr (paths,'/',-1,1))INDIVIDUAL2
         from (select sys_connect_by_path (INDIVIDUAL2,'/')paths
         from TMP_ROLEASSERTIONS
         connect by prior INDIVIDUAL2=INDIVIDUAL1) where instr(paths,'/',1,2)<>0"
    }
```

3.4.2.2 query_direct_predecessors

To query the direct predecessors is to get all individuals that are predecessors of a role for a specified individual. The implementation is similar to query_individual_fillers except we check the status flag "complete2" instead of "complete1" and select the individual1name of a role for a specific individual "individual2". For example, let R be a transitive role, and (a, b): R, (b, c): R, (c, d) : R. If there is a query asking for the direct predecessors of d, we can easily get the answer that {a, b, c} are d's direct predecessors. As we can see from the example, in order to propagate the transitive closure parts, we considered the focused individual2 in the bottom part of the transitive closure path in Oracle's SQL query. Therefore, the SQL code in propagation_roleassertion is as follows:

```
select distinct substr(paths,2,instr(paths,'/',1,2)-2)INDIVIDUAL1,
substr(paths, instr(paths,'/', -1,1)+1,length(paths)-instr (paths,'/',-1,1))INDIVIDUAL2
from (select sys_connect_by_path (INDIVIDUAL1,'/')paths
from TMP_ROLEASSERTIONS
connect by prior INDIVIDUAL2=INDIVIDUAL1) where instr(paths,'/',1,2)<>0"
```

3.4.2.3 query_related_individuals

To query the related individuals is to get all pairs of individuals that are related via the specific relation. In this query, we will check the 'rolecomplete' flag first. If it is True, we select all the pairs of individual1name and individual2name of the given role. Otherwise, we will propagate the RoleAssertion table and execute a query based on this new table.

```
Query_related_individuals (String role, Reasoning reasoner,Connection c)
{

    rolecomplete  = "Select rolecomplete
                        from RoleAssertion
                        where rolename = 'role'";

    if (rolecomplete== True){
            related_individuals = "select distinct individual1name, individual2name
                                    from RoleAssertion
                                    where rolename = 'role'";
            }else {
            roledecendants=get_roledecendants(role);
            descendants_assertion = "select * from RoleAssertion
                                Where rolename = 'roledecendants'";
                ; store the assertion of descendants of the specific 'role' into RoleAssertion table.
            store_roleassertion (descendants_assertion, Connection c, Table RoleAssertion);
            if( is_transitive_role ( role)){
            define transitive_roleassertion as ResultSet;
            transitive_roleassertion = "select * from RoleAssertion
                                        where rolename = 'role'";
            create_table (Tmp_RoleAssertions);
            store_roleassertion(transitive_roleassertion, Connection, Table RoleAssertion);

            propagate_transitiveclosure (table Tmp_RoleAssertions, Connection c);
            ; store back the transitive closure into table RoleAssertion.
            store (Table Tmp_RoleAssetions, Connection c, Table RoleAssertion);
    }

    related_individuals = "select individual1name, individual2name from RoleAssertion
                            where rolename= 'role'";
                        update (Table RoleAssertion, String rolecomplete, String role);
                        ;update the rolecomplete flag of the specific role 'role'
                        ; in the table RoleAssertion
    }
    return related_individuals;
}
```

4. Huge Aboxes

In order to test the performance of our LAS system, we have used the OWL benchmark [19]—university ontology—as an experiment for the evaluation of LAS.

4.1 Introduction to OWL benchmark

One of the test data we used is the Lehigh University Benchmark (LUBM) [42]. It consists of a university domain ontology, with a sufficient number of asserted role relationships. The ontology hierarchy is displayed as a tree, as shown in a popular ontology editor—Protégé.

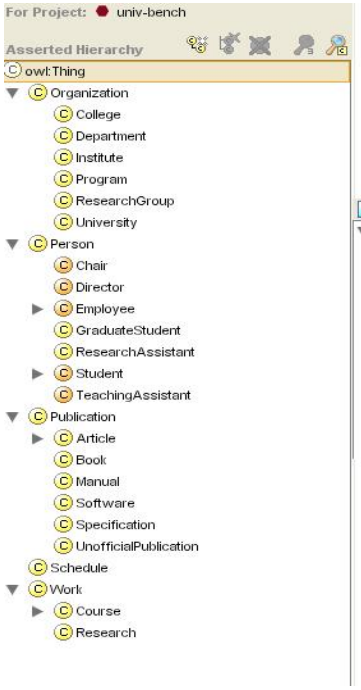


Figure 4.1 The hierarchy of Univ-bench

Figure 4.2 The properties of concept 'Employee'

The characteristics of the university benchmark are that one can generate a customizable size of data based on the same ontology. The university ontology is designed as realistic as possible, which reflects universities, departments, and activities that are related to this domain. It is based on the language ALC. As shown on Figure 4.1, concepts defined in the ontology are represented as a tree. However, it is not the case in the real world. The ontology can be actually considered as a directed acyclic graph, instead of a tree.

4.2 Test result

The university ontology has 43 concepts, and 32 roles. The Data Generator (UBA), generates arbitrary OWL data over the Univ-Bench of university. As in our test, we used the UBA to generate 10 individual universities, and then import the existing data to form a test data set as 1 university, 2 universities…10 universities. For each university, it usually contains around 1500 individuals, 1700 concept assertions, and 4000 role assertions. Also, each university consists of only one department. Therefore, with up to 10 universities we can get around 150000 individuals and 40000 role assertions. The experiment environment was as follows:

Microsoft Windows 2000 Operating System;
2.40GHz Pentium 4 CPU;
1 GHz of RAM; 80 GB of Hard Disk;
Oracle Enterprise 9i 9.2.0.1; JDBC/ODBC

Numbers of Universities	Number of instances	Concept assertions	Role assertions	LAS LoadTime (CPU second)
1	1554	1656	4114	72.1
2	2920	3171	7935	179.6
3	4079	4498	11128	276.2
4	5184	5773	14157	388.4
5	6325	7126	17243	502.0
6	7447	8487	20573	701.7
7	8620	9881	23976	931.9
8	9589	11083	26738	1057.9
9	12950	15279	36793	1612.9
10	14089	16709	49222	1689.9

Figure 4.3 LAS's Loading Time and ontology scalability

As shown in Figure 4.3, one can see that it takes more time to load a file for the first time than just realizing the Abox in RACER. The reason for this is that besides loading the file into RACER, LAS also has to store the basic information about both the Tbox and Abox. RACER has to classify the Tbox, check the consistency of the Abox, and compute the most-specific concepts for each individual in the Abox and the pseudo models for all the atomic concepts in the Tbox and all the individuals in the Abox. In addition, LAS has to parse the data returned from RACER and store it in the database.

However, although it takes a bit more time for LAS to load the file for the first time, as long as the file is loaded and stored in the database, when queries based on an existed database are posted, LAS will just open the database connection and answer queries instantly.

Based on the university ontology, we designed a set of test queries. We divided the test into two groups, one is concerned with concept assertions: “query-individual-types”, “query-direct-individual-types” and “retrieve-individuals”. The other is concerned with role assertions: “query-individual-fillers”, “query-individual-direct-processors” and “query-related-individuals”. We use in the benchmark one university (1656 concept assertions), five universities (11083 concept assertions) and ten universities (16709 concept assertions) respectively. The test was performed according to different status of the knowledge base, while the result was compared with RACER.

Query	Number of universities	KB status	Answer size	RACER	LAS
Query-individual-types	1	New	5	17.64	18.09
		Saved		0.001	0.012
	5	New	5	487.9	643.7
		Saved		0.001	0.11
	10	New	5	898.1	922.5
		Saved		0.011	0.671
Query-Individual-direct-types	1	New	1	18.6	22.1
		Saved		0.001	0.083
	5	New	1	492.3	502.7
		Saved		0.001	0.028
	10	New	1	959.7	1104.1
		Saved		0.011	0.23
Retrieve-indivduals	1	New	28	19.3	17.4
		Saved		0.976	1.983
	5	New	214	526.2	433.3
		Saved		0.05	0.071
	10	New	298	843.6	726.6
		Saved		0.15	0.12

Figure 4.4 Concept Assertion Query time and answer size

As shown in Figure 4.4, one can see that, in general, for both RACER and LAS, queries on a saved KB can be answered much faster than those on a new KB. For the first query, RACER has to check the consistency of the Tbox and Abox, which will take some time. For the saved KB, before RACER is terminated, the

computed result is already stored in the database. Therefore, LAS just need to do a simply lookup to extract the result.

Query	Number of universities	KB status	Answer size	RACER	LAS-Lazy	LAS-enthusiastic
Query-individual-fillers	1	New	1	18.17	19.0	1.76
		Saved		0.001	0.06	0.07
	5	New	1	400.4	496.5	11.71
		Save		0.018	0.04	0.441
	10	New	1	946.5	1314.0	21.20
		Saved		0.012	0.501	0.551
Query-Individual-direct-processors	1	New	1	15.43	17.34	1.6
		Saved		0.015	0.16	0.1
	5	New	1	495.4	654.4	1.17
		Saved		0.024	0.531	0.401
	10	New	1	699.5	887.2	14.56
		Saved		0.012	0.541	0.351
Query-related-individuals	1	New	1878	18.72	24.93	1.72
		Saved		0.002	0.191	0.171
	5	New	2089	466.3	513.2	5.398
		Saved		0.035	0.172	0.254
	10	New	2929	687.2	1031.3	11.43
		Saved		0.025	0.201	0.119

Figure 4.5 role assertions query time and answer size

Another aspect we can notice from Figure 4.5 is that for “query-individual-types”and “query-individual-direct-types”, LAS is a bit slower than RACER, while for the retrieve-individuals, our system is faster than RACER. The reason is that for the first two queries, we need to query RACER directly to get the result, and then store it into the database and update the corresponding completeness status flag. Thus, it would probably take more time than RACER alone. As for the retrieve-individuals, we employ the pseudo model techniques, which return the possible candidates for RACER to do the final check. As one can imagine, for a huge Abox, there exists a great amount of individuals, but in most of the case, only a small part of those individuals are the instances of a specific concept. Therefore, LAS filters away the non-candidates, which make up the majority of

individuals in the Abox, and reduces the workload of RACER's reasoning. As a result, we can see from the table that retrieve-individuals takes less time for LAS than for RACER.

Besides the test about concept assertions, we also constructed some test queries to evaluate the performance with respect to role assertions. As introduced in Chapter 3, our system has implemented two modes: lazy mode and eager mode.

As we can see from Figure 4.5, the LAS-lazy mode is slower than RACER. The reason for that is similar to that of "query-individual-types" and "query-individual-direct-types". The LAS-eager mode takes much less time than RACER because instead of relying on RACER, LAS will propagate the role assertions table by itself, and then execute the queries based on the newly generated role assertion table.

5. Implementation

In this chapter, we will describe the implementation details of LAS. As discussed in chapter 3, LAS consists of the interface to RACER, the interface to the database, and the interface to users (Figure 5.1). The following subsections are divided according to our system's API modules. For the interface to RACER, we concentrate on describing how to parse the RDF/OWL/RACER file. For the interface to the database, we will illustrate the connection with Oracle, MySql and DB2, and an algorithm to compute the transitive closure from different types of databases respectively. We will not discuss the user interface in this thesis. Please refer to the other thesis [41] for details.

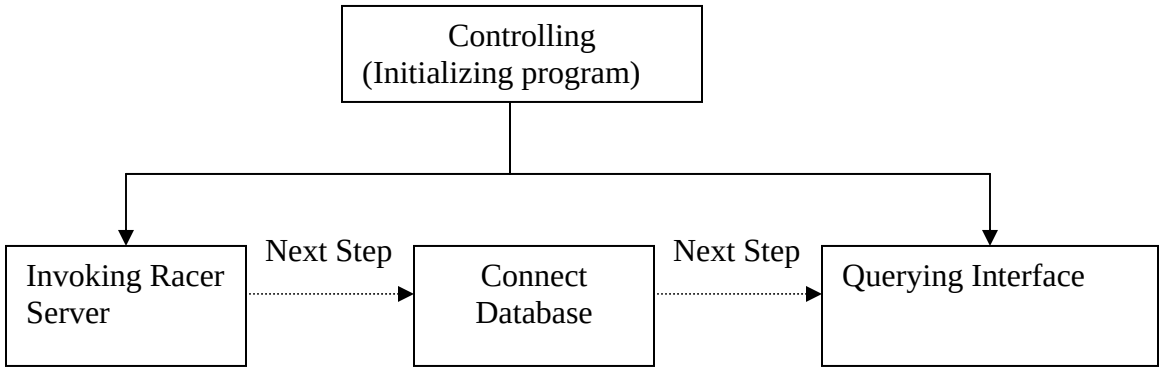


Figure 5.1 System Modules

5.1 Interface to RACER

In order to identify the definition of objects within the domain of a specific ontology, we need to parse the OWL file. Hence, we have two solutions to parse the OWL file.

One is to mainly rely on RACER, which means to use RACER to load the files and parse them automatically. The parsing task is transparent to our system. RACER uses the Wilbur [21] parser to parse the XML-based files, such as XML, RDF, RDFS, DAML, OWL. The other approach is to use the existent JAVA XML technology, such as JAVA SAXParser [22].

As considering the advantages of the first approach—simple, explicit, direct and unrepeated, we decided to rely on RACER and leave it to parse the file. Therefore, what LAS has to do is to parse the results RACER sends back to LAS.

This interface is responsible for communicating with RACER. It consists of JRacer—a java API—to connect to RACER. The main idea of JRacer is to open a socket stream, submit declarations and queries which are represented using strings, and to parse the answer string provided by RACER. Because JRacer provides a Java layer for accessing the services of RACER by calling methods, our system needs to parse RACER's answers.

The implementation of JRacer provides all the RACER commands as different functions. Therefore, in general, LAS parses the RACER result, and stores it in a Java Vector Data Type. Appendix B shows the UML diagram of the interface with RACER, and the UML of LAS's parsing part—reasoning class.

We use two ways to implement parsing RACER's result. One uses the java StringTokenizer class to divide RACER's result into several tokens, and then according to different content of the result, group them into related pairs. The other way treats the RACER result as regular-expression, and then uses the JAVA Pattern and Matcher class to split them into related pairs. The details of the implementation of parsing the RACER result are described in Appendix B.

5.2 Interface with the database

LAS currently uses Oracle 9i, connected through JDBC. The reason we chose Oracle 9i as a developer database is that besides its powerful DBMS, it partially supports the computation of transitive closures. In our system, after getting the taxonomy from RACER, which is represented as parents-children pairs, we have to propagate all descendants of a concept or role in the hierarchy graph. Although the computation of the transitive closure of a binary relation is a common requirement for many applications, the traditional SQL does not support it. [23] enumerates a number of possibilities to overcome this problem.

1. SQL 99

The last ISO standard for SQL, namely SQL 99, provides recursive queries by defining the recursive procedure as a view, then using the view name in an associated query expression [24].

```
WITH RECURSIVE
Q1 AS SELECT…FROM…WHERE…,
Q2 AS SELECT…FROM…WHERE…
SELECT…FROM Q1, Q2 WHERE…
```

2. Proprietary SQL Extensions

IBM's DB2 provides the WITH clause as a proprietary extension to SQL to allow recursive queries. In addition, Oracle also provides some similar facility to compute the transitive closure. Using the CONNECT BY PRIOR and START WITH clauses in the SELECT statement, it can partial support path enumeration [25]. Let us consider a scenario about the transitive closure of a manger-employee relationship. There must exist some employees that do not have any manager, who are at the top level of the hierarchy tree. The transitive closure of a given parent-child relationship is the set of all pairs of employees such that the first employee is a direct or indirect manager of a second employee, or, in graphical interpretation, a set of all pairs of vertices (v,w) for which there exists a path in the graph from v to w. In Oracle 9i or later version, there exist convenient tools for hierarchical queries: CONNECT BY

PRIOR operator and SYS_CONNECT_BY_PATH function.

```
SELECT employee_id, last_name, manager_id
FROM employees
CONNECT BY PRIOR employee_id = manager_id;
```

3. Nested Views

Another solution to deal with the transitive closure in relational database systems is to use nested views. This approach does not rely on any proprietary extensions. However, it has its limitation too, which is the depth of the graphs should be

known in advance. The following example shows the nested view of ancestors-descendant hierarchy computed from parents-children relationship.

```
CREAT VIEW partofprodtwo (ancestor, descendant) AS
SELECT p1.parentid, p2.childid
FROM    partof  p1, partof  p2
WHERE p1.childid = p2.parentid ;

CREAT VIEW partofprodthree (ancestor, descendant) AS
SELECT p1.parentid, p2.childid
FROM partof p1, partof p2, partof p3
WHERE p1.childid = p2.parentid AND p2.childid = p3.parentid;
```

However, although it is simple, this approach is quite limited. It has to know the depth of the graph and it is quite slow for a large amount of entries. Therefore, it is not a very good solution for our system, because LAS has to deal with a huge Aboxes.

After having compared the features of the different approaches, we decided that Oracle's proprietary extension query is more realistic and suitable for our system. However, for using the CONNECT BY PRIOR in Oracle, we have to know at least the top or the bottom of the hierarchy. For example, in LAS, when computing the descendants of the known taxonomy (parents-children pairs) from RACER, we first updates the 'TOP' to become NULL, and then apply the CONNECT BY PRIOR clause.

```
Statement s = c.createStatement();
s.execute("update tmp_desparent set desparent = NULL where desparent = 'TOP'");
String sql =" ";
sql = "insert into tmp_desancestors(tmp_desancestors,tmp_desendants) ";
sql = sql + "select substr(paths,2,instr(paths,'/',1,2)-2)desparent, ";
sql = sql + "substr(paths, instr(paths,'/', -1,1)+1,length(paths)-instr (paths,'/',-1,1))deschildren ";
sql = sql + "from (select sys_connect_by_path (deschildren,'/')paths from tmp_desparent ";
sql = sql + "connect by prior deschildren=desparent) where instr(paths,'/',1,2)<>0" ;
s.execute(sql);
s.close();
```

6. Related Work

In this chapter, we will introduce and evaluate some recent work on ontology management systems. The issue is considered on how to choose an appropriate KBS for a large OWL application. Currently, the prominent existing systems are Sesame system [26], OWLJessKB [28], DLDB-OWL [32], IBM's SnoBase [34], HP's Jena [35], KAON [36] and Instance Store [46]. We will briefly describe each system below.

6.1 Sesame System

The Sesame [27] system is a web-based architecture that provides persistent storage and efficient and expressive querying of large amounts of data in the format of RDF and RDF Schema as well as online querying. The architecture of the Sesame system is as follows:

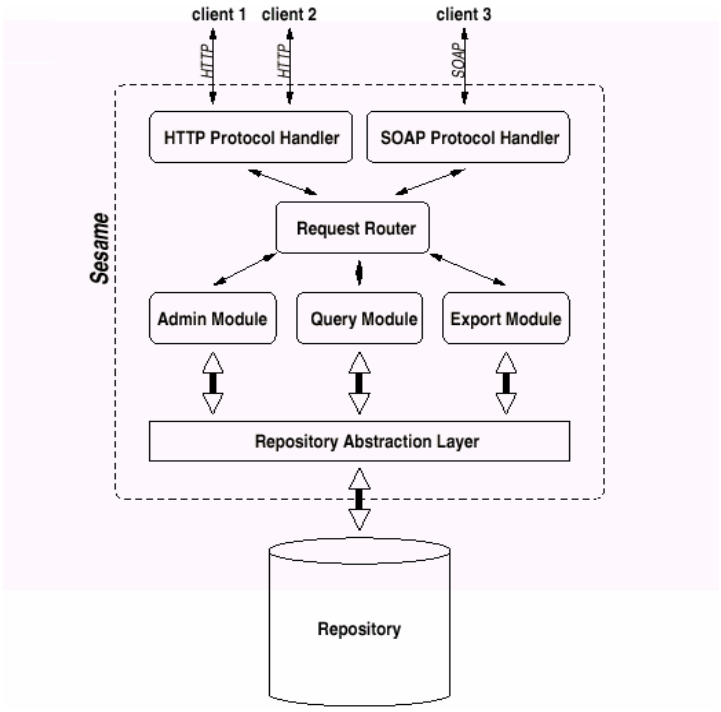


Figure 6.1 Architecture of Sesame System [26]

In order to store RDF data persistently, the Sesame system needs a scalable repository. As we can see from Figure 6.1, Sesame remains DBMS independent, which enables the Sesame system to be implemented on top of a wide variety of repositories without changing any other core component inside the system. It has three main functional modules: (1) the RQL (RDF Query Language) Module, which performs the RQL queries posed by the users; (2) the RDF Administration Module, which controls how to insert as well as delete RDF data and RDF Schema information from a repository; (3) RDF Export Module, which allows for extraction of the complete schema and/or data from a model in RDF format. The Sesame system supports RDF/RDFS inference, but it is not complete reasoner for OWL Lite.

6.2 OWLJessKB

OWLJessKB is a reasoner for OWL [28], which is a successor to DAMLJessKB [29]. It uses the Java Expert System Shell (Jess) [30]. DAMLJessKB maps the assertions, which are represented by RDF triples, into facts in a production system and then applies rules implementing the relevant Semantic Web languages. OWLJessKB loads RDF or OWL files, and uses parts of the Jena toolkit to parse RDF documents. Once a document is parsed, OWLJessKB asserts the triples into a production system along with the rules derived from the OWL semantics. As a result, new facts are populated and entailed from the knowledge base [31]. However, OWLJessKB deals with a language close to OWL Lite. Figure 6.2 shows the process of DAMLJessKB.

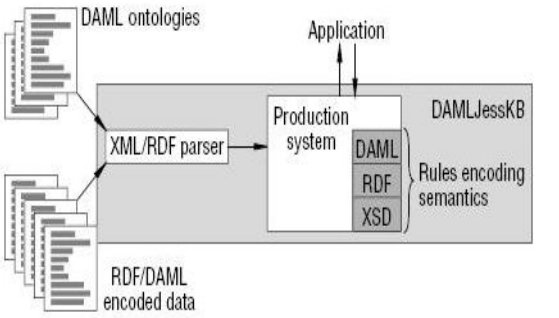


Figure 6.2 The process of DAMLJessKB [30]

6.3 DLDB-OWL

Implemented by Lehigh University, DLDB is a knowledge base system that extends a relational database management system with additional reasoning capabilities for OWL [32]. It uses the description logic reasoner FACT to precompute the subsumption hierarchy and stores it into a common RDBMS (MS Access).

It uses a 'ONTOLOGY-INDEX' table to manage the information about the loaded ontologies in the database [33]. After loading the file, OWL parsers parse the original source file, and translates it into a SHIQ equivalent knowledge base and generates a temporary XML file. The description logic reasoner FACT reads the file and checks the consistency of the classes, computes the taxonomy of the ontology and reports all the implicit relationships by writing back to that temporary XML file. After that, DLDB creates tables and views and stores the ontology hierarchy into the database. Therefore, as all the implicit information

about the ontology becomes explicit due to the reasoning of FACT, DLDB can be used to answer queries now.

DLDB is designed to suit the needs for personal or small business users who wish to take advantage of semantic web technology. Its scope of language is *ALC*. Due to its full dependence on description logic reasoner to compute the taxonomy, it can answer a large range of queries, but it is not very self supported, it has to highly rely on DL reasoner. It is also incomplete.

6.4 IBM's SNOBASE

IBM's ontology Management System, namely SNOBASE [34] for Semantic Network Ontology Base, is a system for loading ontologies from files or via the Internet and for locally creating, modifying, querying and storing ontologies. It consists of an ontology inference engine, a persistent data stores, an ontology directory and an ontology source code connector.

It does not rely on the current description logic reasoner, such as RACER or FACT, instead, it has its own inference reasoner. Internally, the system uses an inference engine, an ontology models and the inference engine deduces the answers and returns results sets similar to JDBC result sets.

6.5 HP's Jena

Jena [35] is a Java framework for building semantic web applications. It is a programming toolkit, which uses the java programming language and provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine.

It consists of a RDF API, a module for reading and writing RDF in RDF/XML, [1] N3 and N-Triples, an OWL API, an in-memory persistent data storage, and a user query interface—RDQL. The main component, the inference subsystem, is designed to support RDFS and OWL, which allows additional information to be derived from original facts. The architecture of the inference machinery is illustrated as below:

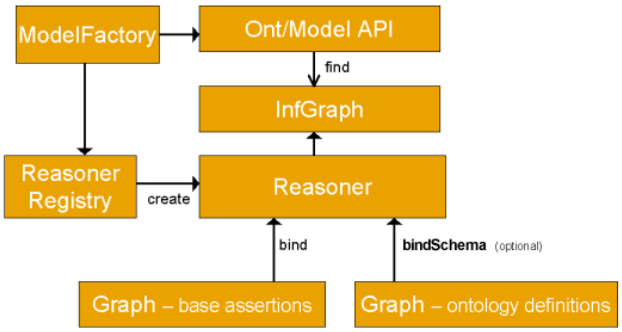


Figure 6.3 the structure of inference subsystem in Jena [35]

As described in Figure 6.3, the application accesses the inference machinery through the ModelFactory. With the help of the reasoner in the inference subsystem, the original facts with the additional statements, which were derived

[1] Language notation 3, which is basically equvalent to RDF in its XML syntax. It contains subject, verb and object. [42]

from the data using rules or other inference mechanisms implemented by the reasoner, are returned by the format of a newly created model.

6.6 KAON

KAON is an open-source ontology management infrastructure targeted for business applications [36]. It includes a comprehensive tool for building and managing onotologies and provides a framework for building ontology-based applications. KAON consists of a number of different modules providing different functionalities such as creation, storage, retrieval, maintenance and application of ontologies [37]. The architecture is described in Figure 6.4.

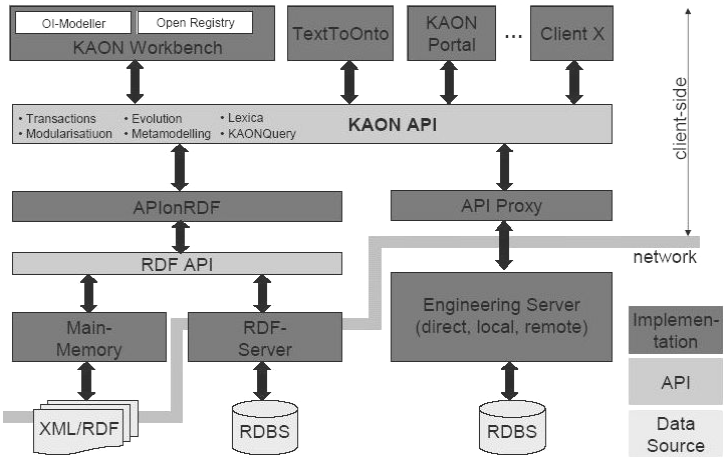


Figure 6.4 KAON Architecture Overview [36]

After loading a file, the KAON API accesses RDF-based data sources via the RDF API, for which two reference implementations exist: one is a simple main memory implementation including RDF parser and serializer; the other is a RDF Server which implements the RDF API remotely and allows RDF ontology

models to be stored in relational databases and hence enables transactional ontology modification.

KAON is based on RDF(S) with proprietary extensions for algebraic property characteristics, cardinality, modularization, meta-modeling, explicit representation of lexical information. However, it does not support any reasoning or Abox queries.

6.7 Instance Store

The Instance Store is a Java application and implements a form of simple A-Box reasoning by using a database to store asserted descriptions of large numbers of individuals. It is applied in the field of Gene description and Web-services discovery now.

The instance store is composed of a database, a reasoner and an ontology, and it has two operations: assert (individual, description) and retrieve (description).

Four strategies that cache the information are implemented: The basic strategy is that only the minimal information is stored in the database and the subsumption and hierarchy information is retrieved from a DL reasoner; the second strategy optimizes the basic one just through caching the classification hierarchy in the database; the third strategy is an alternative optimization of the second one through caching the transitive closure into primitives tables; the fourth and most

optimized approach is by caching the classification of each asserted and retrieved description.

Although efficient, it is severely restricted, because it can only deal with role-free Aboxes (an Abox without any role assertion between individuals).

7. Conclusion

After illustrating the techniques we used to develop LAS, and the architecture and detail implementation of LAS (Large Abox Store), we will provide a summary of our work in this chapter.

7.1 Conclusion

LAS is a description logics application which uses a combination of Abox reasoning and database queries to perform efficient Aboxes reasoning. By extending the DL reasoner RACER with a relational database, LAS stores the taxonomy and the Abox assertions of the given knowledge base in its database.

LAS can deal with the language $ALCH_{R^+}$. It does not support number restrictions or functional roles.

In conclusion, the most interesting features of LAS are as follows:

1. **Completeness:** The description logic it deals with is a logic-- $ALCH_{R^+}$, which extends *ALC* by adding role hierarchies and transitively closed roles. Moreover, it provides reasoning services on complete role-embedded Aboxes. LAS, as a second layer of RACER, is a sound and complete because RACER is a sound and complete description logics reasoning system.
2. **Speed:** As a filter for RACER, LAS largely reduced the reasoning time for queries, especially for retrieval queries. By employing the pseudo model mergable

test, which is implemented via SQL, LAS reduce the number of candidates for Abox queries by filtering out individuals that are proven to be not relevant for a particular query. Because it forwards to RACER only the reduced set of individuals relevant to answer this query, in the presence of thousands of individuals, the time savings can be significant.

3. **Reuse:** As we know that description logic reasoners do reasoning all in the main memory, if they are terminated, all the information they computed before will be lost. For the long-term usage, it is critical to store the information for future reuse. By combining them with the databases, all the information about the taxonomy and Abox assertions are stored into the database. Even though later on RACER will be shut down, the complete and computed information is still kept in the database.

4. **Flexibility:** LAS is not constrained by the **U**nique **N**ame **A**ssumption (UNA), it can deal with the situations where the same individual can have different names.

5. **Two mode implementation:** LAS system is implemented in two modes: Lazy and Eager. For the lazy mode, LAS relies completely on RACER, while for the eager mode it relies on SQL queries to generate the complete information for the posed queries. For the lazy mode, the system works on demand, although it takes slightly more time because it has to communicate with RACER, the information it obtains is complete. For the eager mode, it fulfills the task in minimal time. It is very beneficial for the future use.

7.2 Future Work

The future work of our system includes to explore more optimizations in order to support more complex queries. It can be summarized as follows:

1. To support nRQL (new RACER Query Language): Although nowadays, LAS can query most of the queries RACER provided, it needs to support more complicated queries. One solution to this problem can rely on combing nRQL from RACER and SQL from databases.

2. Adapt to more databases: So far we only used the Oracle database for developing and testing our system. We plan to update our system such that it can be used with more relational databases.

3. Multiple operating systems: So far, the operating system we used is Windows, it is very easy to extend it so that it can be compliable with Unix and Mac.

4. Extend the language scope: We also consider to extend our language scope from $ALCHR^+$ to $ALCFNHR^+$ which includes the functional roles and number restriction.

Reference

[1] Tim Berners-Lee, CERN: Information Management: A proposal May, 1990

[2] F. Manola, E. Miller : RDF Primer. W3C Recommendation Feb 10, 2004

[3] D. Nardi, R. J.Brachman An Introduction to Description Logics.The Description Logic Handbook. Cambridge University Press, 2003

[4] T. R.Gruber : Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, Volume 43.

[5] I. Horrocks, U. Sattler and S. Tobies: Reasoning with individuals for description logic SHIQ. *Proc.of the 17th Int. Conf. On Audomated Deduction(CADE 2000)*: 482-496

[6] F. Baader, W. Nutt: Basic Description Logics .The Description Logic Handbook. Cambridge University Press, 2003

[7] M.Jarke, Y.Vassiliou, J.Clifford: How does an expert system get its data? In *VLDB* 1983 pp.70-72.

[8] R.Brachman, A.Borgida: Loading data into description reasoners. Volume 22. *SIGMOD*, 1993

[9] P. Bresciani: Querying database from description logics. *In KRDB'95*, 1995.

[10] M.Simonet, M.Roger, A.Simonet: Bringing together description logics and database in an object oriented model. *In DEXA2002*, 2002.

[11] S. Tessaris, I. Horrocks : Abox Satisfiability Reduced to Terminological Reasoning in Expressive Description Logics. In Matthias Baaz and Andrei Voronkov, editors, *Proc. Of the 9th int. conf. On logic for programming and automated reasoning (lpar'02)*, volume 2514 of Lecture Notes in Computer Science. Springer, 2002

[12] B. Hollunder: Algorithmic Foundations of Terminological Knowledge Representation Systems. PhD thesis, University des Saarlandes. (1994)

[13] F.F.Donini, M. Lenzerini, D. Nardi, A. Schaerf: Deduction in Concept Languages: From Subsumption to instance checking. *J. of Logic and computation 4 (1994)*pp.423-452.

[14] V. Haarslev and R. Moeller and A.Y.Turhan: Exploiting Pseudo Models for Tbox and Abox Reasoning in Expressive Description Logics. *Proc. Of International*

Joint Conference on Automated Reasoning, IJCAR'2001, R.Goré, A.Leitsch, T.Nipkow (Eds.) pp.61-75.

[15] V. Haarslev and R. Moeller: Optimizing Tbox and Abox Reasoning with Pseudo Models. *Proc. of the international workshop in DL2000,* Aachen, Germany, 2000. pp. 153-162.

[16] E. Franconi : Propositional Description Logics. Course material for an introductory online course in Description Logics. http://www.inf.unibz.it/%7Efranconi/dl/course/slides/prop-DL/propositional-dl.pdf (last visited: 08-22-2005)

[17] A. Aho and J. Ullman. Universality of data retrieval languages. In *Proceedings 6th Symposium on Principles of Programming Languages, Texas*, pages 110-120, 1979

[18] D.C.Kreines: Oracle SQL: the Essential Reference. O'Reilly, 1 edition 2000.

[19] J. Heflin, Y. Guo and Z. Pan: Benchmarking DAML+OIL repositories. *In Second International Semantic Web Conference. ISWC 2003.*

[20] SWAT Projects-the Lehigh University Benchmark. http://swat.cse.lehigh.edu/projects/lubm/index.htm (last visited: 08-19-2005)

[21] Wilbur semantic web toolkit for CLOS. http://wilbur-rdf.sourceforge.net/ (last visited: 08-19-2005)

[22] Java Pattern Class and regular expression http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html (last visited: 08-19-2005)

[23] S.Wagner: A Data Warehouse for Cross-Species Anatomy. MSc Dissertation. Heriot-Watt University, 2002

[24] A.Eisenberg, J.Melton SQL:1999, formerly known as SQL3.

[25] S.S.B.Shi et al: An Enterprise Directory Solution with DB2. Technical report, IBM, 2000

[26] Sesame System: http://www.aidministrator.nl (last visited: 08-19-2005)

[27] J. Broekstra, A.Kampman and F.v Harmelen, Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. *In the Semantic Web (ISWC2002), Volume 2342* pp.54-68

[28] OWLJessKB: A semantic Web Reasoning Tool
http://edge.cs.drexel.edu/assemblies/software/owljesskb/ (last visited: 08-17-2005)

[29] KAMLJessKB http://edge.cs.drexel.edu/assemblies/software/damljesskb/ (last visited: 08-19-2005)

[30] Jess: the Rule Engine for the Java Platform http://herzberg.ca.sandia.gov/jess (last visited: 08-22-2005)
[31] J. Kopena and W.C. Regli DAMLJessKB : A Tool for Reasoning with the Semantic Web. IEEE Intelligent Systems, published by the IEEE Computer Society.

[32] Y.Guo, Z. Pan and J. Heflin. An Evaluation of Knowledge Base System for Large OWL Datasets. Third International Semantic Web Conference, Hiroshima, Japan, LNCS 3298, Spinger, 2004. pp. 274-288

[33] Z.X. Pan, J.Heflin DLDB: Extending Relational Databases to Support Semantic Web Queries. In Workshop on Practical and Scalable Semantic Systems. ISWC2003

[34] SnoBase : IBM Ontology Management System
http://alphaworks.imb.com/tech/snobase (last visited: 08-17-2005)

[35] Jena: A Semantic Web Framework http://jena.sourceforge.net/index.html (last visited: 08-19-2005)

[36] KAON: An Ontology Management Infrastructure For Business Applications.
http://kaon.semanticweb.org/ (last visited: 08-19-2005)

[37] T.Gabel, Y.Sure and J.Voelker Karlsruhe: Ontology Management Infrastructure. Insititute AIFB, University of Karlsruhe, technical report 2004

[38] M. Buchheit et al. : Refining the Structure of Terminological Systems : Terminology = Schema + Views. *AAAI 1994* pp.199-204

[39] M. Roger, A. Simonet, M. Simonet : Bringing together Description Logics and Databases in an Object Oriented Model. *DEXA 2002* pp.504-513

[40] R.A. Schmidt : Algebraic Terminological Representation, Master's thesis, 1991, Department of Mathematics, University of Cape Town, Cape Town, South Africa

[41] J.Y. Wang : Large Abox Store (LAS): Database Support for Tbox Queries, Master thesis, Department of Computer Science and Software Engineering, Concordia University 2005

[42] W3C: Primer: Getting into RDF & Semantic Web using N3. W3C tutorial 2005

[43] D.L. McGuinnes, F.V.Harmelen: OWL Web Ontology Language Overview. W3C recommendation 10 Feb, 2004.

[44] V. Haarslev, R. Moeller : Racer: A Core Inference Engine for the Semantic Web. *Proc. of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003)* Sanibel Island, Florida, USA. Oct 20, 2003 pp.27-36.

[45] C.Cumbo, W.Faber, G.Greco, N.Leone: Enhancing the magic-set method for disjunctive datalog programs. *(ICLP'04) pp.371-385*

[46] I. Horrocks, L.Li, D. Turi : The Instance Store: Description Logic Reasoning with Large Numbers of Individuals. *DL2004*

[47] C.M.Chen, V.Haarslev, J.Y.Wang : LAS: Extending RACER by a Large Abox Store. *Proc of the 2005 International Workshop on Description Logics (DL2005)* Edinburgh, Scotland, UK.

[48] Marc de Graauw, Using Topic Maps to Extend Relational Databases, March 2003. O'Reilly XML.com http://www.xml.com/pub/a/2003/03/05/tmrdb.html (last visited : 09-10-2005)

Appendix A

This appendix lists the specification of the RACER query commands used in the thesis. For the detail description about the commands, please refer to RACER Manual.

(1) individual-types
Semantics: Gets all atomic concepts of which the individual is an instance.
Syntax: (individual-type *IN* &optinal (*ABN* (abox-name * current-abox)))
Arguments: *IN* -individual name
ABN –Abox name
Values: List of name sets

(2) individual-direct-types
Semantics: Gets the most-specific atomic concepts of which an individual is an instance.
Syntax: (individual-direct-type *IN*
&optinal (*ABN* (abox-name * current-abox)))
Arguments: *IN* -individual name
ABN –Abox name
Values: List of name sets

(3) all-roles
Semantics: Returns all roles and features from the specified Tbox
Syntax: (all-roles &optinal (*tbox* * current-tbox))
Arguments: *tbox* -Tbox object
Values: List of name sets

(4) all-transitive-roles
Semantics: Returns all transitive roles the specified Tbox
Syntax: (all-transitive-roles &optinal (*tbox* * current-tbox))
Arguments: *tbox* -Tbox object
Values: List of name sets

(5) all-symmetry-roles
Semantics: Returns all roles that are symmetry from the specified Tbox
Syntax: (all-symmetry-roles &optinal (*tbox* * current-tbox))
Arguments: *tbox* -Tbox object
Values: List of name sets

(6) individual-fillers
Semantics: Gets all individuals that are fillers of a role for a specified individual.

Syntax: (individual-fillers *IN R* &optinal (*ABN* (abox-name* current-abox)))

Arguments: *IN* -individual name of the predecessor
R - role term
ABN - Abox name

Values: List of name sets

(7) retrieve-direct-predecessors

Semantics: Gets all individuals that are predecessors of a role for a specified individual.

Syntax: (retrieve-direct-predecessors *R IN abox*)

Arguments: *IN* -individual name of the role filler
R - role term
abox - Abox name

Values: List of individual names

(8) retrieve-related-indivdiuals

Semantics: Gets pairs of individuals that are related via the specified relation.

Syntax: (retrieve-related-indivdiuals *R abox*)

Arguments: *R* - role term
abox - Abox name

Values: List of paris of individual names

(9) retrieve-individual-pmodel

Semantics: Gets pseudo models of the specific individual from a specific Abox.

Syntax: (retrieve-individual-pmodel *IN* &optinal (*ABN* (abox-name* current-abox)))

Arguments: *IN* -individual name of the predecessor
ABN - Abox name

Values: List of pseudo model sets

(10) retrieve-description-pmodel

Semantics: Gets pseudo models of the specific description from a specific Tbox.

Syntax: (retrieve-description-pmodel *des* &optinal (*TBN* (tbox-name* current-tbox)))

Arguments: *IN* -individual name of the predecessor
TBN - Abox name

Values: List of pseudo model sets

Appendix B

In this appendix, we will describe the detail of implementation of LAS's interface with RACER. First, the UML diagram of interface with RACER and UML of LAS's parsing part—"reasoning" class will be shown below:

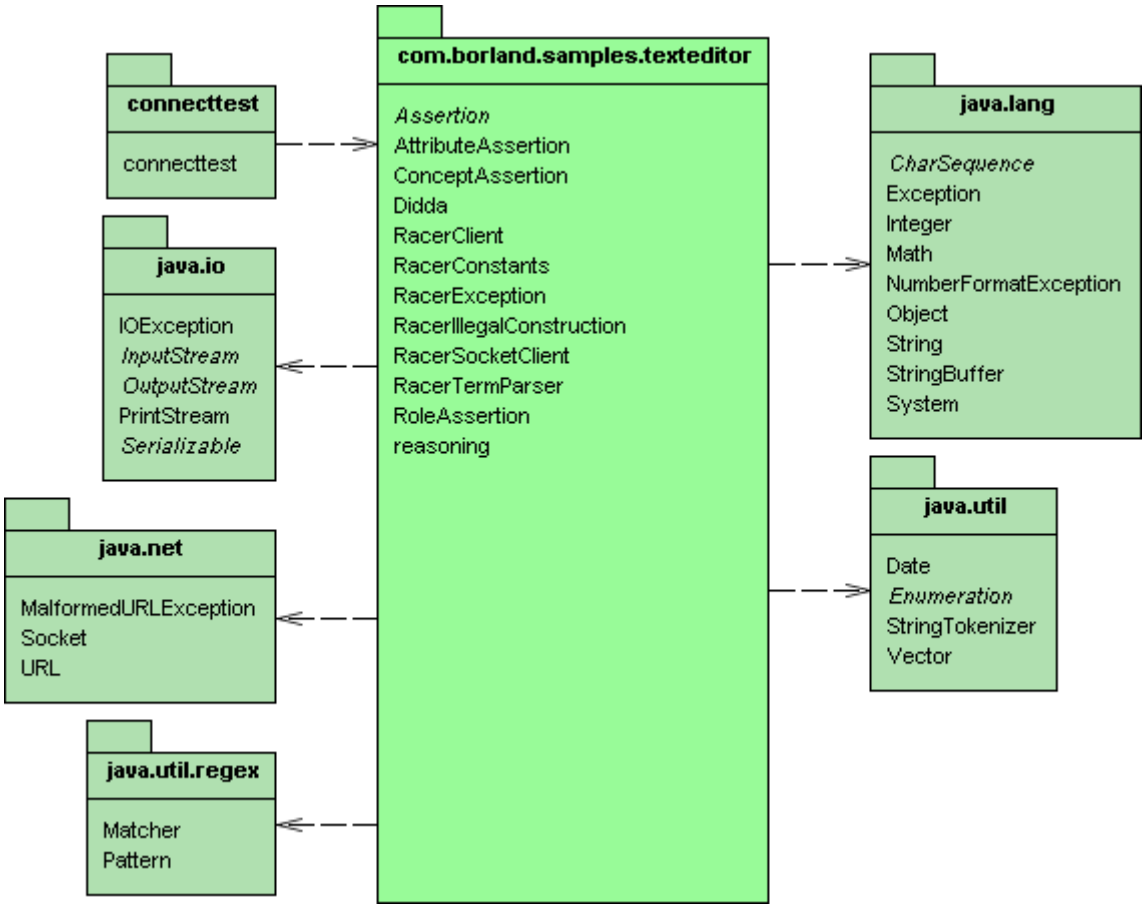


Figure B.1 UML diagram of the interface with RACER

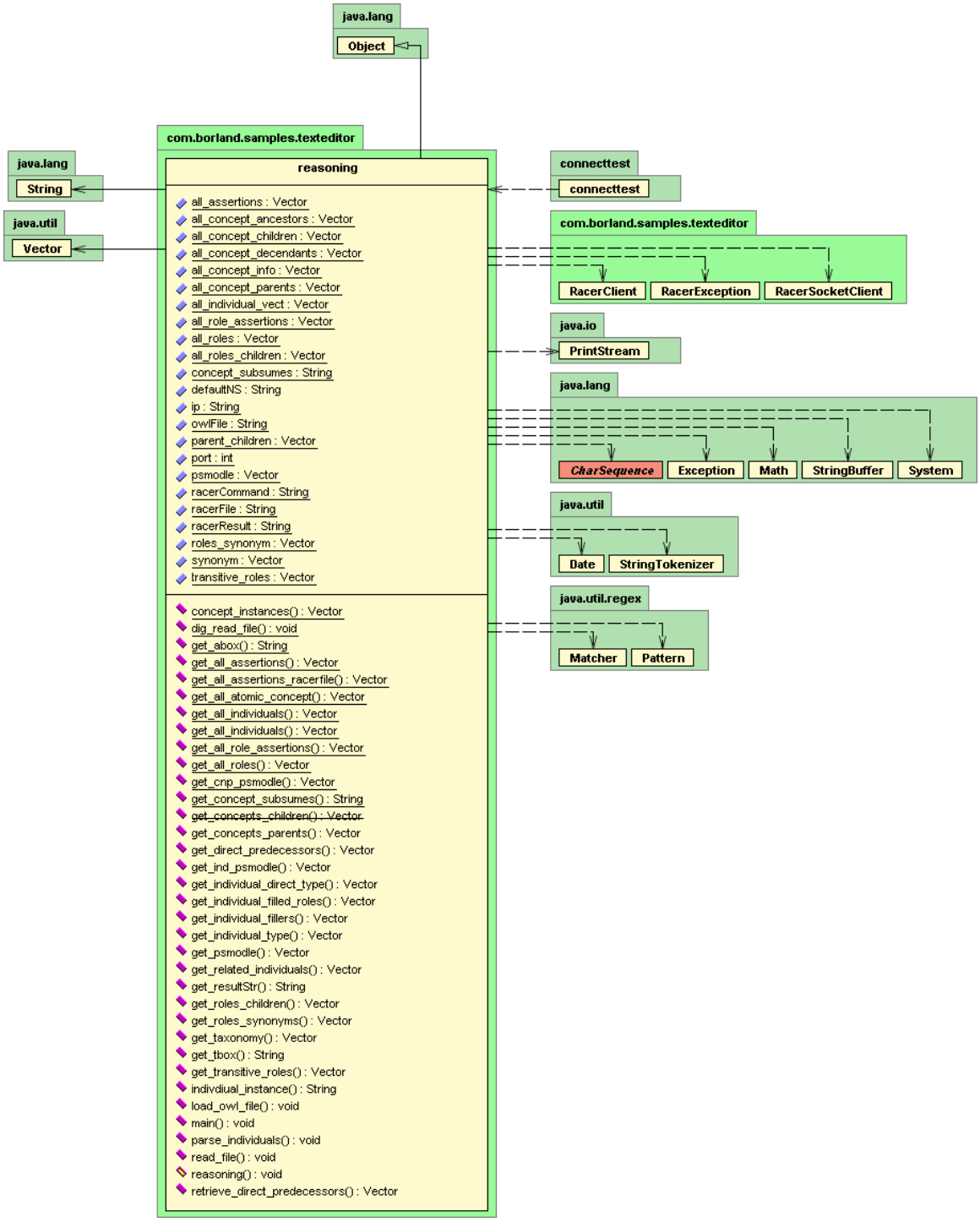


Figure B.2 UML diagram of reasoning class

Before describing the details, we first analyze the characteristic of the RACER result. For example, in order to get all the role assertions from RACER, part of the result is as follows:

```
(ALL-ROLE-ASSERTIONS) -->
(((|http://a.com/ontology#Toodles||http://a.com/ontology#Tom|)
  |http://a.com/ontology#In_same_series|)
((|http://a.com/ontology#Toodes||http://a.com/ontology#Zoot_Cat|)
   |http://a.com/ontology#First_Appearance|)
((|http://a.com/ontology#Toodles||http://a.com/ontology#Tom|)
   |http://a.com/ontology#In_same_cartoon_series|))
```

Observing the RACER result of “all-role-assertion”, we can see that the role assertion is a list comprising three elements, the format is: ((individual1 individual2) role). As a result, we transfer the RACER result to tokens. One token stands for one complete term—individual, role, concept. After we add all these tokens into a vector, we will organize them according to different content of the RACER results. The following shows how to transfer racerResult into tokens.

```
racerResult=(racerResult.replace('(',' '));
racerResult=(racerResult.replace(')',' '));
racerResult=(racerResult.replaceAll("    ",","));
racerResult=(racerResult.replaceAll("   ",","));
racerResult=(racerResult.replaceAll("  ",","));
racerResult=(racerResult.replaceAll(" ",","));
client.closeConnection();
StringTokenizer tokens = new StringTokenizer(racerResult, ",");
```

Another way to parse the RACER result is using the JAVA Pattern and Matcher class [22]. We treat the result string as a regular expression, and then build different patterns and a parser matching the corresponding expression. For example, to get the taxonomy from RACER, the racerResult is shown as follows:

```
(TAXONOMY) -->
(TOP NIL (|http://a.com/ontology#Animated_cartoon|
            (|http://a.com/ontology#Animation_star| |http://a.com/ontology#Cartoon_star|)
```

|http://a.com/ontology#Feature||http://a.com/ontology#Human|
|http://a.com/ontology#Place| |http://a.com/ontology#Sayings|))

The syntax of taxonomy result is defined as follows:

<entry>: := (<node><parents> <children>)
<node> : :=<name>|<synonym_name>
<synonym_name> : :=(<name>+)
<parents> : :=(<node>+)|NIL

Therefore, for the regular expression, <node>, <parent> and <children> can be construced as :

Taxonomy:\\(([\\w|:.\\~\\-/#]+)\\s\\(([\\(\\)\\w|:.\\~\\-/#\\s]+)\\)\\s\\([\\(\\)\\w|:.\\~\\-/#\\s]+\\)
where:
<node> : ([\\w|:.\\~\\-/#]+)
<parents>: ([\\(\\)\\w|:.\\~\\-/#\\s]+)
<children>: ([\\(\\)\\w|:.\\~\\-/#\\s]+) where \\w represents a word character, \\s stands for whilespace character.

According to the definition of <node>, <parents> and <children>, we match our RACER Result with the pattern, and parse it into a parent-children vector.