

ONTOLOGY QUERY LANGUAGES FOR THE SEMANTIC WEB:

A PERFORMANCE EVALUATION

by

ZHIJUN ZHANG

(Under the Direction of John.A.Miller)

ABSTRACT

Ontology languages and corresponding query languages play key roles for representing and processing information about the real world for the emerging Semantic Web. Efforts have been made to develop various ontology languages. Each ontology language provides different expressive power and also computational complexity for reasoning. Ontology query languages were developed to query the information defined by these ontology languages and reasoning systems. We conduct a study to compare their expressive power, efficiency, scalability and best performing situation. We also introduce the OPS system which consists of two subsystems: the ROPS subsystem, an OWL reasoner built on Jena2 and the SWOPS subsystem, a First Order Logic (FOL) reasoning system based on Vampire. Each part can work separately and cooperate with each other for different tasks. In this paper, we will compare different ontology query languages together with query systems and give evaluations from the user's view point.

INDEX WORDS: OWL, SWRL, Performance Evaluation, Semantic Web, Ontology Query

Language

THE ONTOLOGY QUERY LANGUAGES FOR THE SEMANTIC WEB:

A PERFORMANCE EVALUATION

by

ZHIJUN ZHANG

M.S., The University of Georgia, 2005

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial

fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2005

© 2005

Zhijun Zhang

All Rights Reserved

ONTOLOGY QUERY LANGUAGES : A PERFORMANCE EVALUATION

By

ZHIJUN ZHANG

Approved.

Major Professor: John.A.Miller

Committee: Ismailcem Budak Arpinar

Liming Cai

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August, 2005

DEDICATION

To my wife Peng Xu

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. John. A. Miller for his guidance, and his assistance. Dr. Miller has been very generous to provide his suggestions and help. He also helped me to overcome the time limitation. I would also like to thank Dr. Ismailcem Budak Arpinar and Dr. Liming Cai for their valuable suggestions and being a part of my committee. I would also like to thank my friends Yanfeng Chen, Min Li, Yandong Su and Liyue Hou for their help during my research.

TABLE OF CONTENTS

	Page
CHAPTER	
1 INTRODUCTION.....	1
2 ONTOLOGY LANGUAGES FOR THE SEMANTIC WEB: A PERFORMANCE EVALUATION.....	3
1. Introduction.....	4
2. Ontology Languages for the Semantic Web.....	5
3. Query Languages and Implementations.....	8
4. ROPS (RDF-OWL Ontology Processing System).....	16
5. SWOPS (SWRL Ontology Processing System).....	19
6. Benchmarking Results.....	21
7. Related Work.....	31
8. Conclusions and Future Work.....	32
3 CONCLUSIONS.....	36
REFERENCES.....	38
APPENDIX	
A THE DESCRIPTION LOGIC FAMILIES.....	42
B USER GUIDE.....	44
C INSTALLATION GUIDE.....	50

CHAPTER 1

INTRODUCTION

Currently, there is tremendous effort going on to define languages for the Semantic Web. The goal of these languages is to represent information over the World Wide Web, so that it is understandable and accessible by machine. In addition, these languages should also guarantee that they have enough expressive power to represent rich semantics of real world information and also they should be efficient enough to be processed by machine.

Some of these languages are very successful, such as RDF(S) and OWL. They are recommended as standard by W3C. There are also some languages still under development. For example, SWRL is an extension of OWL DL (a sublanguage of OWL) with Horn like rules

To utilize the information represented by Semantic Web languages, ontology query languages are being developed to provide the ability for users to retrieve information from ontology. Several of these query languages are implemented in ontology query systems. For these query languages and implemented systems, there has been little comparison and performance evaluation done so far. In our research, we compare several ontology query languages giving their advantages and drawbacks. We also evaluate the performance of corresponding query systems using the LUBM benchmark. The systems tested include Jena2 with RDQL, RACER with nRQL, and Vampire with SWRL.

We introduce OPS, an ontology query system that consists of two subsystems: ROPS

built on Jena2 and SWOPS built on the Vampire theorem prover. Our intension to build OPS is to provide an ontology query system that can deal with both complex ontology and large ontology. The ROPS subsystem can handle ontology which is simple, but with a large data set. The SWOPS subsystem can process very complex ontology that is not quite so large.

CHAPTER 2

ONTOLOGY QUERY LANGUAGES FOR THE SEMANTIC WEB:

A PERFORMANCE EVALUATION¹

¹ Zhiun, Zhang, John A. Miller. Submitted to *Journal of Web Semantics*.
Corresponding author. *Address*: Department of Computer Science, University of Georgia, Athens, GA 30602. *E-Mail* – jam@cs.uga.edu. *Telephone*: 706-542-3440. *Fax*: 706-542-2966.

1 Introduction

The Semantic Web is considered by some as the next generation of the World Wide Web. The development of suitable languages for representing ontology and efficient reasoners are two key tasks to be solved for the Semantic Web. Currently, several ontology languages and ontology query languages have been developed. For these languages and querying systems, there does not exist a thorough comparison and evaluation among them. This paper evaluates those mainstream ontology languages and ontology query languages and provides a view about the advantages and disadvantages of each of them. After that, it provides our evaluations on what query languages and what query systems to choose under a certain situation.

In this paper, we will focus on three ontology languages: RDF(S) [1], OWL [2], and SWRL [3]. The Resource Description Framework (RDF) is a framework developed by the W3C (World Wide Web Consortium) for representing information in the World Wide Web. RDF inherits XML syntax and exploits URI to identify resources. RDF Schema (RDFS) is used to specify the vocabularies in RDF. RDF(S) provides a foundation for other advanced languages for similar purposes. The Web Ontology Language (OWL) is a semantic markup language for ontology representation. It extends RDF syntax and is derived from DAML+OIL [4] and many other influencers. OWL provides three increasingly expressive sublanguages: OWL Lite, OWL DL (Description Logic) and OWL Full [5]. The Semantic Web Rule Language (SWRL) is a rule language combining OWL and RuleML [3]. It is still under development by the Joint Markup Language Committee². SWRL extends OWL DL with binary/unary first order Horn like rules. This rule extension makes SWRL more powerful and flexible than OWL DL. As a trade-off, the computational complexity of SWRL increased to semi-decidable [6].

Along with the development of these ontology languages, several query systems have been developed. Jena2 is a framework built by HP Labs. It provides multiple reasoners for RDF, RDFS, and OWL. It also provides a flexible query language, called RDQL [7]. Jena2 is sufficient for many sophisticated query tasks, it can support OWL Lite and partially OWL Full now. Another system is OWL-QL [8]. It introduces some very valuable features for ontology querying. The other system is RACER that uses nRQL as the query language. RACER supports reasoning for RDF, RDFS, and OWL ontology languages. Based on Jena2, we built ROPS (RDF-OWL Ontology Processing System), a sub-system of OPS (Ontology Processing System) supporting ontology reasoning in RDFS mode and OWL mode. To test the power of the SWRL language, we also built our own SWRL query tool -- SWOPS (SWRL Ontology Processing System) which is based on the Vampire Theorem Prover [9]. SWOPS is also a sub-system of OPS.

² The Joint United States / European Union ad hoc Agent Markup Language Committee was created in October 2000 by Jim Hendler of DARPA and Hans-Georg Stork of the European Union Information Society Technologies Programme (IST). Refer <http://www.daml.org/committee/> for detail.

This paper is organized as follows: Section 2 reviews the relevant ontology languages. Section 3 discusses several query languages supporting RDF, RDFS, OWL and SWRL, respectively. We focus on languages/systems with some degree of OWL support. In Section 4, we present ROPS, built for OWL querying and then describe SWOPS, built on the Vampire prover for SWRL querying in Section 5. The comparison of the performance for these query languages along with other Semantic Web query languages is discussed in Section 6. Section 7 describes the related work and section 8 gives the conclusions and future work.

2 Ontology Languages for the Semantic Web

This section briefly discusses various ontology languages for the Semantic Web (technically, these languages are in different layers of the Semantic Web layer cake [10], but are all used for representing ontology). The focus is on the differences and correspondences among different languages. Since all of these languages are influenced by RDF and RDFS, we first briefly describe RDF and RDFS. DAML+OIL is very similar to OWL and is also very briefly described. In this section, we will focus a bit more on the higher end of the Semantic Web layer cake, OWL and SWRL.

2.1 RDF and RDFS

The goal of the Semantic Web is to make information on the Web both accessible and understandable by not only humans, but also computers. RDF(S) was thus developed to represent information in the Web. In RDF(S) the resources in the Web are identified by Web identifiers (Uniform Resource Identifier or URI) [11]. To make it machine processible, RDF inherits XML-based syntax. After years of development, RDF(S) now has formal syntax, formal semantics and XML Schema datatypes. It is a W3C recommended information representation standard for the Semantic Web. The RDF abstract syntax has a graph pattern, where the statements are represented as N-triples [12] (format: Subject – Predicate – Object or node-arc-node link, hence the term ‘graph’) [13].

RDF can express resource properties and their values. RDFS extends RDF by providing the ability to define RDF vocabularies such as classes, properties, types, ranges, domains, etc. However, RDF and RDFS only have very limited expressive powers [11] [14]:

- RDF(S) cannot express equality and inequality;
- RDF(S) cannot define enumeration of property values;
- RDF(S) cannot apply cardinality and existence constraints;
- RDF(S) cannot describe unique, symmetric, transitive, inverse relationships among properties;
- RDF(S) cannot describe union, intersection and complement;
- The domain and range in RDF(S) can only be specified globally [14].

As a result, several more sophisticated languages have been developed to meet these

requirements.

2.2 DAML+OIL

DAML+OIL [4] is a combination of DAML (DARPA Agent Markup Language) and OIL (Ontology Inference Layer) [15]. It has an RDF/XML syntax based on the frame paradigm [16] and describes the structure of a domain (schema) in an object-oriented style. DAML+OIL consists of a set of axioms asserting the relationships between classes and properties.

DAML+OIL uses a Description Logic style model theory to formalize the meaning of the language [16]. This is a very important feature to reduce arguments and confusions, thus giving the language the ability to precisely represent the meanings of information. This ability is crucial for automatic reasoning, which is the goal of the Semantic Web.

The new features DAML+OIL supports are the following:

- Constraints (restrictions) on properties (existential/universal and cardinality),
- Boolean combinations of classes and restrictions, e.g., union, complement and intersection,
- Equivalence and disjointness,
- Necessary and sufficient conditions, and
- Constraints on properties.

However, DAML+OIL tries to be compatible with RDF syntax, but this raises some serious syntactic and semantic problems [16]. Another problem is that DAML+OIL datatypes are not compatible with RDF, since RDF did not provide datatype definition ability when DAML+OIL was being developed.

2.3 OWL

OWL [5] was developed on top of RDF and borrowed from DAML+OIL. Like RDF, OWL is the standard recommended by W3C for Semantic Web. OWL is powerful in expression, but complex for computation. To compromise between expressive power and acceptable computational complexity, OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full. Among them, OWL Lite is a subset of OWL DL, and OWL DL is a subset of OWL Full.

OWL Full contains all the OWL language constructs and provides the free, unconstrained use of RDF constructs [2]. In OWL Full, *owl:Class* is equivalent to *rdfs:Class*. OWL Full also permits classes to be individuals. A class can even be a property of itself. In OWL Full, *owl:Things* and *rdfs:Resource* are equivalent too. This means that object properties and datatype properties are not disjoint.

The advantage of this jointness is that it provides high expressive power. Unfortunately, the drawback is that it is computationally undecidable [16]. As the result, it is very difficult to build a reasoning tool for OWL Full. Although theoretically, OWL Full can be processed via some FOL engine, it can not guarantee quick and complete answers.

As a sublanguage of OWL Full, OWL DL introduces several restrictions on the usage of OWL constructs. These restrictions are carefully chosen to make sure that OWL DL is decidable. OWL DL does not support all of RDF(S) [2]. In OWL DL, classes, datatypes, individuals, and properties are all pairwise disjoint. Datatype properties and object properties are also disjoint. As a result, inverse, transitive and symmetric relationships can not be applied to datatype properties. Cardinality constraints are also forbidden on transitive properties.

These restrictions guarantee that OWL DL is computationally decidable. It is equivalent to DL **SHOIN(D)** [16] whose worst case is non-deterministic exponential time (NEXPTIME). Reasoning for OWL DL can be supported via DL or FOL reasoners without losing accuracy. It is the best choice for users who require accurate results with maximal expressive power.

OWL Lite can be considered as a simplified version of OWL DL. It supports simple classification hierarchies and simple qualification restriction. Constructs such as *one of*, *unionOf*, *complementOf*, *hasValue*, *disjointWith* and *DataRange* are not allowed in OWL Lite. Furthermore, some constructs also restrain the use of certain resources. The cardinality restrictions in OWL Lite can only have value of 0 or 1.

The computational complexity of OWL Lite is equivalent to that of DL **SHIF(D)**, which is exponential time (EXPTIME) in the worst case [16]. The purpose of OWL Lite is to provide a minimal useful subset of OWL with an efficient complete reasoner. Building a DL reasoner for OWL Lite is relatively straightforward. Several current DL reasoning systems perform very well on OWL Lite repositories.

2.4 SWRL

The Semantic Web Rule Language (SWRL) [3] is under development as a combination of the OWL DL with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language [17]. SWRL can be considered as DL + function-free FO Horn rules [18].

SWRL adds rules to OWL DL. The reason is that these rules provide more expressive power to Description Logic. For example, you can use FOL to define (represented in the human readable syntax of SWRL) the concept *Aunt*.

$$\text{Parent}(\text{?x}, \text{?y}) \wedge \text{Sister}(\text{?y}, \text{?z}) \Rightarrow \text{Aunt}(\text{?x}, \text{?z})$$

Here ‘Parent(?x, ?y) \wedge Sister(?y, ?z)’ is called the antecedent (body) and ‘Aunt (?x, ?z)’ is called the consequent (head) of the rule. Whenever the antecedent holds, the consequent holds. OWL can not define relationships like this. Applying rules one can also extend OWL with composition capability.

Although SWRL is relatively new, FOL has already been thoroughly studied. Also, the combination with FOL allows SWRL to easily communicate with traditional relational database systems. This feature is very attractive since most information in the real world is still stored in relational databases.

The disadvantage of SWRL is its computational complexity. Although it is still under construction, developers have agreed that it will support most of the features of Description Logic and partial Horn Logic Programs. It is sure to be undecidable [3]. However, since FOL has been studied for more than a century, it will still benefit from the precedent model theory and FOL engines. A good solution for this is to borrow the idea of OWL – developing multiple sublanguages of SWRL with decreasing complexities, from full to simple versions and decreasing the complexities from being undecidable to being decidable. This task has not formally started yet, but has already been considered [19].

3 Query Languages and Implementations

Based on the ontology languages described above, several query languages and systems have been already developed. RDQL [7] is the query language for Jena2. Vampire [9] is a FOL theorem prover using TPTP³ format for problem input and query input. nRQL is the query language for RACER. Finally, OWL-QL [8] is a query language for the OWL-QL system.

SPARQL [20] is a Server-Client-based RDF query language. It has SQL syntax and is influenced by RDQL and SquishQL⁴. SPARQL supports disjunction in the query and thus can process more complex query than RDQL. SPARQL also provides optional variable binding and result size control mechanisms for real world usage. However, we did not choose SPARQL as a query language for testing, since there was no supporting system available at the time.

Because Jena2 is a development package and did not provide an executable interface, we build ROPS system over Jena2 for OWL ontology querying. Also, since Vampire is a prover, we build SWOPS over Vampire as a FOL ontology reasoner.

In rest of this section, we briefly describe each of these query languages and the systems developed for them.

3.1 nRQL and RACER

RACER[21] is a description logic ontology reasoning system supporting DL *ALCQHI+(D-)*. RACER extends basic Description Logic *ALC* by adding role hierarchies, transitive roles, inverse roles, and qualifying number restrictions.

³ TPTP stands for “Thousands of Problems for Theorem Provers”. Related information can be found at <http://www.cs.miami.edu/~tptp> by Geoff Sutcliffe and Christian Suttner

⁴ <http://swordfish.rdfweb.org/rdfquery/>

nRQL (new RACER Query Language, an extension of RQL) [21] is an extended query language for RACER. nRQL was constructed based on Description Logic model theory [22].

In Description Logic, the knowledge base is represented in a tuple (*T-Box*, *A-Box*). *A-Box* contains assertions about individuals and *T-Box* defines the concepts (classes or types of instances), roles (predicates), and features (attributes/properties) of these instances.

In *A-Box*, the set of individual (instance) names I is the signature of *A-Box*. The individual set must be disjoint with both the concept (class) set and the role (property) set. In *A-Box* there are four types of assertions: asserting an individual IN_1 to be of a concept C ; asserting role filler for a role R to an individual IN_2 that is, individual IN_1 is related to individual IN_2 via role R ; assigning an attribute to an individual; or asserting a restriction on an individual. RACER uses an optimized tableau algorithm to calculate the satisfiability problem [23]. Tableau algorithms are the dominate algorithms for DL reasoning currently. The basic idea of this algorithm is to apply transformation rules (these rules preserve the consistency of original *A-Box*) to the *A-Box* until no rules are applicable. If there is no contradiction in the *A-Box*, it is called consistent. Given a concept description C , C is called satisfiable *iff* there exists an interpretation I so that $C^I \neq \emptyset$. The subsumption problem in RACER is reduced to the satisfiability problem [24].

The RACER *T-Box* includes the conceptual model of concepts and roles. The model consists of a set of concept names C and a set of role names R . By exploiting several operators (constructs), one can build complex concepts and roles. RACER supports *Negation*, *Conjunction*, and *Disjunction* constructs. RACER also provides *Existential* and *Universal* quantified restrictions to ensure that certain roles filler have to be of a specific concept. RACER provides three types of number restrictions: *At-most*, *At-least*, and *Exactly*. The restrictions can be applied to roles. However, only non-transitive roles (also no transitive sub-roles) can apply cardinality restrictions to attributes.

For datatypes, RACER provides a *Concrete Domain* that describes concrete predicate restrictions for attribute fillers. The types includes cardinal, integer, real, complex, and string. The restrictions are all mapped to this concrete domain.

The concept axioms that RACER supports include *concept inclusion* that states the subsumption relationship between two concepts, *concept equation* that states equivalence between two concepts, and *concept disjointness* that states the disjointness relationship among concepts. RACER can also define the concept name as a special type of a concept term. In RACER, concept axioms can be cyclic or even several axioms for just one concept.

Role declarations in RACER are unique. Only one declaration can be done to one role name. This restriction also applies to attributes in *T-Box* and individual names in *A-Box*. Role declarations can declare features (attributes) of a given role, declaring a role to be transitive, and declare hierarchy relationships among roles. In the current version of RACER,

the sub-role relationship can not be cyclic.

To suit variant purposes of reasoning, RACER provides two inference modes. Given a query, RACER can minimize the computation time in the lazy inference mode. If the lazy inference mode is enabled, only the individuals involved in a “direct types” query are realized. However, when the query involves much classifying, another mode – the eager inference mode provides better performance. The other way to save processing time is to save A-Box and T-Box in separate files. Thus, classifying one of them does not need to affect another.

As an effort to be a universal query framework for the Semantic Web, RACER can read RDF, RDFS, DAML+OIL, and OWL documents as inputs⁵. RACER can process OWL Lite knowledge bases, as well as OWL DL.

nRQL provides a powerful query language for RACER. It is an A-Box query language. The variables in the queries are to be bound to A-Box individuals.

An example query with complex predicate is like:

```
(retrieve (?x ?y) (?x ?y (:constraint (age) (age) >))),
```

The example query retrieves the individual pairs that who are older than whom. In this query, nRQL first assigns the *age* attribute with individuals bound to *?x* and *?y*. Here variables *x* and *y* are not restricted to specific classes. Any instance having a property *age* is possible to participate in the answer.

nRQL also supports compound queries using the *and* operator:

```
(retrieve (?x ?y) (and (?x mother) (?y man) (?x ?y has-child))),
```

This query asks about a mother who has whom as a son. Here variable *x* is bound to class *mother* and variable *y* is bound to class *man*.

nRQL also supports *negation as failure* by providing a *not* operator:

```
(retrieve (?x) (?x (not grandmother))),
```

This query does not return any female individual unless she was explicitly defined in the ontology as not having any grandchild.

With the features mentioned above, RACER is a very powerful and formalized description logic inference engine with a flexible query language – nRQL, for the Semantic Web. It provides many attractive features, especially for when expressive power and performance are both critical. Even for those who want the maximum expressive power, it is still a good choice.

⁵ RACER applies consistency checking during ontology loading process. It can not load test cases directly in our research. We uses OilEd [25]. to load test ontology into RACER.

3.2 OWL-QL

The OWL Query Language (OWL-QL) is a well designed language for querying over knowledge represented in a repository [8]. OWL-QL is an updated version of DAML Query Language (DQL).⁶ It is intended to be a candidate for query-answering dialogues among answer agents and query agents. Then information receivers and information providers can transfer queries and answers via the Internet.

OWL-QL provides a formal description of the semantic relationships among queries, answers, and knowledge bases used to produce answers. An OWL-QL query contains a collection of OWL sentences in which some URI references are considered to be variables. This collection is called *query pattern*.

The answers to a query provide bindings to the query pattern so that the binding result of the query pattern is existentially quantified. It may not necessarily entail a binding for every variable in the query. OWL-QL extends this by enabling clients to designate which variable or set of variables must be bound to the query pattern. Each variable occurring in the answer pattern has one of the three binding types, i.e., *must-bind variable*, *may-bind variable*, and *don't-bind variable*. A quantified answer must provide all the bindings for all the *must-bind variables*, may provide bindings for any of the *may-bind variables* and must not provide bindings for any of the *don't-bind variables*. All the lists are disjoint with each other. By adjusting the variable for binding, OWL-QL can answer the questions such as “What resources make the query pattern true” or “Is the query pattern true”. This is quite flexible and allows for sophisticated queries. It also provides users the ability to specify an answer pattern, so that the server knows in what format the answers should be returned.

Since OWL-QL provides a large range of query-answer services, the size of the answers returned for a query may be very large. In addition, if the query is over a large KB, the process may take an unexpected long time. The solution is to permit the answering server to return part of query results. In addition, there needs to be a communication channel between clients and server. OWL-QL introduces the mechanisms called *answer bundle* and *process handle* to deal with this.

Unfortunately, because an executable package of OWL-QL is not available right now, we could not compare it with the other query systems in section 6.

3.3 RDQL and Jena2

RDQL is a query language for RDF in the Jena framework. The development of RDQL is to provide a data-oriented query model. This means that RDQL only retrieves information stored in the model which contains a set of N-Triple [12] statements. RDQL provides no reasoning mechanisms. The reasoning is provided by user selected reasoners bound to the model containing the original ontology information. Provided with a proper reasoner, RDQL

⁶ Developed by the Joint United States/European Union ad hoc Agent Markup Language Committee

can process ontology in various languages including OWL. Unlike OWL-QL, all variables in the input query are *must-bind* variables.

An RDQL query has the following form:

```
SELECT ?x
WHERE (?x,<ns0:father>,<?y>) (?y,<ns0:cars>,<?z>) (?x,<ns0:carType>,<truck>)
USING ns0 FOR <http://www.somewhere.com/driverregistration#>
```

This query asks for the retrieval of all the instances of person whose fathers have trucks.

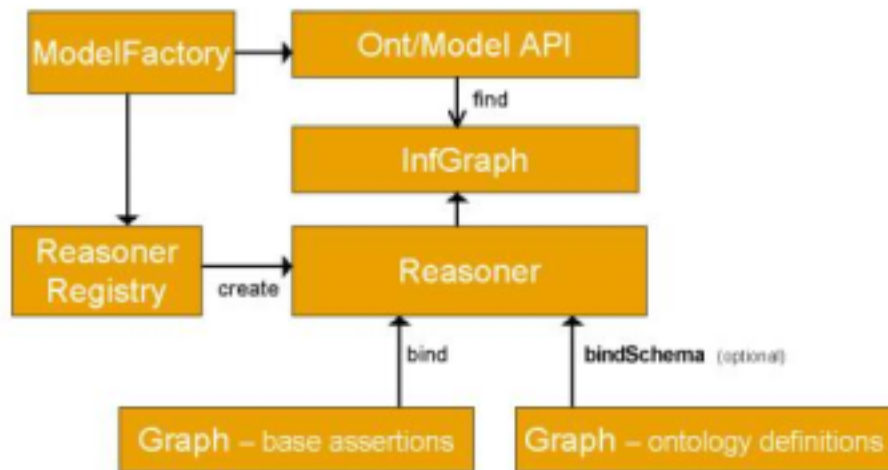


Figure 1: Structure of the inference machinery of Jena2

The notation of ?x represents a variable. In the *WHERE* clause, a set of N-Triples define the query pattern of a query. The *USING* clause defines an alias for the prefix of a URIs to simplify the URI. RDQL can query about predicates or objects too. The limitation of RDQL is that it does not support disjunction in a query. Though RDQL is relatively simple in syntax, it is efficient for most of the ontology queries. The simple syntax also makes RDQL very flexible.

The machinery structure of Jena2 is shown in Fig. 1. The asserted statements are held in the base RDF graph. The selected reasoner uses information in the base graph to generate additional entailments from the original set of statements. RDQL queries are processed by Ont/Model.

To optimize ontology reasoning, Jena2 provides a variety of reasoners for ontology with different formats and complexity. These reasoners include the following:

Transitive Reasoner:

The transitive reasoner provides storage and traverses classes and properties. It processes the *transitive* and *symmetric* properties of *rdfs:subPropertyOf* and

rdfs:subClassOf. This reasoner is included in RDFS rule reasoner by default. It is a basic reasoner that can be embedded in other hybrid reasoners.

RDFS Rule Reasoner:

The RDFS rule reasoner processes a configurable subset of the RDFS entailments. There are three subsets of them – Full, Default, and Simple. The RDFS rule reasoner is a hybrid implementation. It combines other basic reasoners to do reasoning tasks to reduce redundancy and flexibility. The hierarchical relationships are calculated by the transitive reasoner embedded into the RDFS rule reasoner by default. The rest of the RDFS operations are executed by the generic hybrid rule reasoner.

OWL Rule Reasoner:

The OWL rule reasoner intends to be the smooth extension of RDF Reasoner. Compared to more sophisticated Description Logic reasoners, this reasoner is less efficient. The OWL rule reasoner supports OWL Lite plus some of the constructs of OWL Full.

Generic Rule Reasoner:

The Generic rule reasoner is a rule-based reasoner that supports user defined rules. It supports forward chaining, tabled backward chaining, and hybrid execution strategies. The forward chaining engine is based on the standard RETE algorithm [26]. The backward chaining engine is a logic programming (LP) engine supporting tabling, which is the technique used in XSB⁷. This technique can successfully resolve recursive problems, such as transitive closure to avoid infinite loops.

The current version of Jena is 2.2 and it supports most of the major ontology languages including RDF(S), DAML+OIL, and OWL. However, Jena2 does not fully support OWL yet. It can understand all the syntax of OWL, but it do not support reasoning in OWL Full. Jena2 now supports OWL Lite plus some constructs of OWL DL and OWL Full such as *hasValue*. Some of the important constructs that are not supported in Jena2 include *unionOf*, *complementOf*, and *oneOf*. The constructs Jena2 supports are listed in Table 1. Jena supports cardinality restrictions on literal valued properties. Notice that for the cardinality restrictions, Jena2 only supports the values of 0 and 1 (the same as in OWL Lite).

Jena2 provides a model interface that can store N-triples persistently in a database. When a query is processed on a stored ontology again, Jena2 can just load the statements from the database without having to redo the reasoning process.

Jena2 also provides a generic rule engine and an interface for users to enter rules as an extension to the original rule set. This feature theoretically makes Jena2 support SWRL reasoning (Jena2 only supports a subset of features of SWRL right now).

⁷ XSB is developed by Stony Brook University, in collaboration with Katholieke Universiteit Leuven, Universidade Nova de Lisboa, Uppsala Universitet and XSB, Inc.

Table 1: The construct list supported by Jena2

Constructs supported by Jena2
rdfs:subClassOf, rdfs:subPropertyOf, rdf:type
rdfs:domain, rdfs:range
owl:someValuesFrom, owl:allValuesFrom
owl:minCardinality, owl:maxCardinality, owl:cardinality
owl:intersectionOf
owl:equivalentClass, owl:disjointWith
owl:sameAs, owl:differentFrom, owl:distinctMembers
owl:Thing
owl:equivalentProperty, owl:inverseOf
owl:FunctionalProperty, owl:InverseFunctionalProperty
owl:SymmetricProperty, owl:TransitiveProperty
owl:hasValue

In our research, we built the ROPS system based on Jena2 for evaluating Jena2 and RDQL. The ROPS system will be introduced in detail in section 4.

3.4 SWRL as a Query Language

As we described above, SWRL is a combination of OWL DL and first order Horn like rules. OWL DL is a subset of DL (OWL DL is equivalent to DL *SHOIN (D)*) and DL is a subset of FOL [18]. We can represent SWRL in FOL without losing information. Theoretically, we can start from a universal FOL engine to implement a SWRL query engine. Currently, there is no complete implementation for SWRL. However, we can use a human readable syntax (Datalog-like) to formulate queries to simulate SWRL queries and process them using a FOL theorem prover.

To process OWL via a FOL engine, we need to translate Description Logic statements to FOL rules [27]. We give some example translations below:

DL statement $C_A \subseteq C_B$, where C_A is a subclass of C_B , can be represented in FOL as:

$$C_A(?x) \Rightarrow C_B(?x).$$

$P_A \subseteq P_B$, P_A is a *subProperty* of P_B can be represented as:

$$P_A(?x, ?y) \Rightarrow P_B(?x, ?y).$$

$dom(P) : C$, *Domain of Property P is Class C*, can be represented as:

$P(?x, ?y) \Rightarrow C(?x).$

$range(P) : C$, Range of Property P is Class C , can be represented as:

$P(?x, ?y) \Rightarrow C(?y).$

$a \in C$, a is a instance of Class C , can be simply represented as:

$C(a).$

$(x, b) \in P$, (a, b) is a instance of Property P , can be represented as:

$P(a, b).$

The other constructs can also be represented in the same way as discussed in [27] [28]. All of these constructs in OWL DL can be translated into FOL rules with consistent meanings.

The FOL engine we adopted is Vampire⁸, an automated theorem prover (ATP) developed for first order classical logic by Andrei Voronkov and Alexandre Riazanov in the Computer Science Department, University of Manchester. Vampire exploits several techniques to improve its performance such as optimized algorithms for backward and forward subsumption, indexing and discrimination trees. These optimizations make Vampire very efficient in theorem proving.

Vampire is a saturation-based theorem prover. The problem given to the kernel of Vampire is a set of clauses. Each clause is a disjunction of literals. The prover then tries to derive new clauses from the initial clause set until an empty clause is found (proved). Otherwise, the prover goes on until no new clause can be generated (unproved) and the problem set is called saturated [29]. As a result, Vampire is very efficient for unsatisfiable problem like proving a subsumption (a subsumption holds if the corresponding problem is unsatisfiable), since it can often derive an empty clause before consuming all clauses to stop. For satisfiable problems like non-subsumption, it will be less efficient since it have to exhaust all the clauses until it can not generate any new clause.

Vampire is a theorem prover, which makes it less efficient for acting as a real time FOL query engine. Vampire can only prove a set of clauses that contains a query problem one at a time and with only three possible outputs: satisfiable (unproved), unsatisfiable (proved) and timeout. It can not return a list of answers like a common query engine. As a result, for queries potentially having multiple answers, Vampire can not return all the answers at once. In addition, Vampire needs to reload the problem data (ontology information in our test) for each query, since Vampire can only take a file as input. All these limitations compromise the performance of Vampire as an ontology reasoner.

As a sophisticated prover, Vampire provides a list of options to control the behavior of the kernel to improve performance. However, most of these options can not improve the performance of ontology reasoning for query processing.

To test the performance of a FOL ontology reasoner, we built the SWOPS system by modifying Vampire to improve its performance and created a GUI for it too. We will

⁸ The website of Vampire is <http://www.cs.man.ac.uk/~riazanoa/Vampire/>

describe SWOPS in section 5.

4 ROPS (RDF-OWL Ontology Processing System)

ROPS is a sub-system of our Ontology Processing System (OPS). It is the RDF/OWL ontology reasoning system we developed using Jena2. Taking advantage of the friendly Java interfaces of Jena2, we developed the ontology reasoning kernel and visual user interface for ROPS. In this section, we first illustrate the architecture of the OPS system. After that, we explain the related features in Jena2 and then describe the implementation of ROPS.

4.1 Architecture of OPS System

OPS is a flexible ontology reasoning system for both OWL and SWRL. It is composed by two subsystems: RDF-OWL Ontology Processing System (ROPS) based on Jena2 and SWRL Ontology Processing System (SWOPS) based on Vampire. This ROPS subsystem will be discussed later in this section and the SWOPS subsystem will be illustrated in section 5. Fig. 2 illustrates the architecture of OPS.

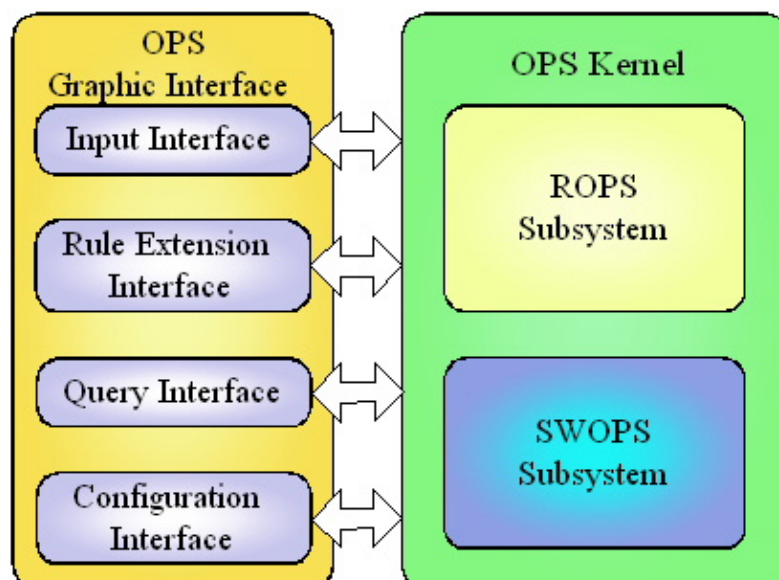


Figure 2: Architecture of OPS

The OPS system provides a graphic user interface through which users can control its subsystems and perform ontology reasoning process. The GUI provides universal API for both of OPS and SWOPS subsystems. Each subsystem provides a plug-in to communicate with OPS interface. The Input interface provides I/O control for reading input ontology and output the results. The configuration interface provides controls on the performance of ROPS and SWOPS subsystems. The rule extension interface provides the mechanism to extend current ontology with SWRL rules for the queries beyond OWL. The queries are entered via a query interface to communicate with the selected subsystem. A user guide of

OPS is in Appendix B.

Users can choose ROPS or SWOPS as the ontology reasoner based on the input ontology and reasoning tasks. Currently, OPS does not support direct switching between ROPS and SWOPS during the reasoning process.

4.2 Related Features in Jena2

Jena2 provides several choices for the input interface. The input could be from a file on a local disk, over a network, or any resolvable URL. This makes OPS very flexible to get input resources among different location.

Jena2 supports most of the current ontology languages as input. ROPS supports RDF(S), DAML and OWL (we only use OWL in our test). A user can define the type of input language to improve performance. OWL Full is set as the input language by default. Users can also define whether to perform reasoning in memory or in a database. The default mode is to process in memory.

When creating the ontology model, users can specify the type of the input by providing the model factory the URI of the specific language. Currently, Jena2 supports RDFS, DAML+OIL, OWL Lite, OWL DL, and OWL Full. For each type of input language, Jena2 provides several reasoners optimized for that ontology language. Users can also select what reasoner should be bound to the ontology model.

Since selecting a proper reasoner can affect performance, to generate the best performance, the users of ROPS need to have sufficient knowledge of the ontology content and queries. ROPS selects the OWL Rule Reasoner by default, which is the most powerful and least efficient reasoner.

4.3 Architecture of ROPS Subsystem

ROPS is embedded in OPS and provides compatible API to OPS GUI. There are four main functions in ROPS to implement the four modules in the visual interface of OPS. The operations selected via the GUI are processed by ROPS and then sent to Jena2 for corresponding actions. There is also code to answer queries about direct subsumption relationships. This function is done by directly calling Jena2 API. The architecture of ROPS is shown in Fig. 3.

Through the intermediate layer, users can use GUI of OPS system to control I/O process. Users can also configure the process of ROPS by selecting a proper reasoner based on the content of the input ontology and complexity of the query tasks. A proper reasoner can provide better performance. If users do not specify the reasoner, the system will choose the OWL Rule Reasoner automatically.

Users can extend the ontology by adding new rules via the rule extension interface of OPS. New rules can be input from an input box or be imported in a batch from a local file or

URL. However, this enhancement can only be used together with the generic rule reasoner. This limits the usage of rule extension mechanism of ROPS.

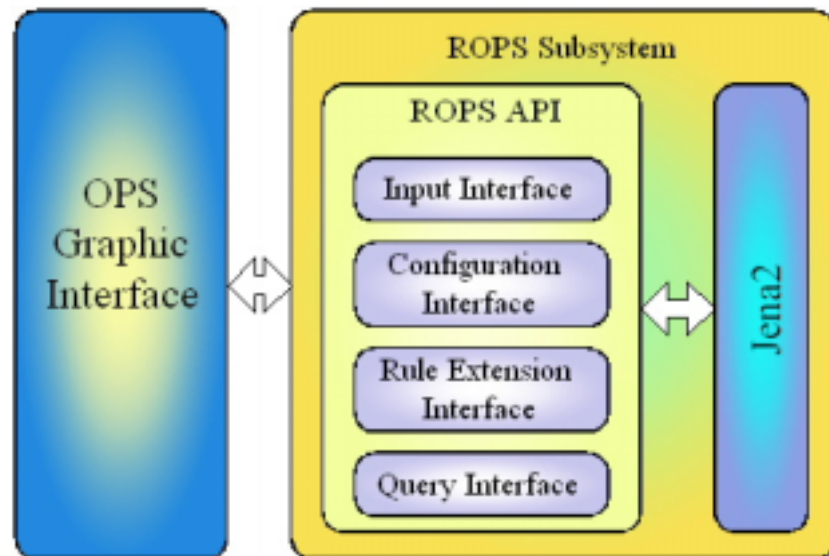


Figure 3: Architecture of ROPS

Users can also add and remove statements like adding rules to the existing ontology together with reasoners other than the generic rule reasoner. This is also processed through the rule extension interface of OPS. However, both these activities will trigger the reasoner to restart the reasoning process instantly. To improve the efficiency, users can choose to set the reasoners in non-incremental mode to notify the reasoner not to re-infer until a new query is submitted. Users can also trigger the reasoner to restart reasoning after all modifications are done.

Users can input RDQL queries directly via the query interface. However, RDQL has limited support for queries on the schema. ROPS extends RDQL to support this type of queries (e.g., queries about *subsumption* relationships, such as *superclassOf*). These extensions queries are processed by directly calling and combining the functions provided by Jena2.

ROPS supports the following types of queries:

- *subClass* or *superClass* of a given class type,
- *subProperty* or *superProperty* of a given property type,
- what type of class (all or direct superclasses) a given instance is,
- whether two given instances or two types are the same or different,
- what is the value of a specified property of a instance,
- all the instances of a class,
- all the properties of a given instance (currently only can get direct properties), and
- combinations of these queries together for more query power.

In the query interface, users can also choose to generate derivation history for the input queries. The derivation history can give the detail about the inferred statements. However, since this process needs to create additional data for each step, it is very expensive both in time and in memory. The default setting of this option is off.

ROPS is still an incomplete query system for RDF and OWL. The main purpose of ROPS is for academic research and test usage. The performance of ROPS will be discussed in section 7.

5 SWOPS (SWRL Ontology Processing System)

SWOPS is another sub-system of our OPS system. It is a FOL ontology query system based on the Vampire theorem prover. SWOPS is a prototype query system for SWRL. Since OWL can be translated into SWRL without losing information, SWOPS supports both OWL and SWRL.

The Vampire prover is not optimized for ontology reasoning and processing of queries. As a result, the performance of Vampire is compromised and can be improved. In the SWOPS system, we modified Vampire to solve some of the problems mentioned in section 3.4. These improvements discussed below make SWOPS more efficient for processing queries.

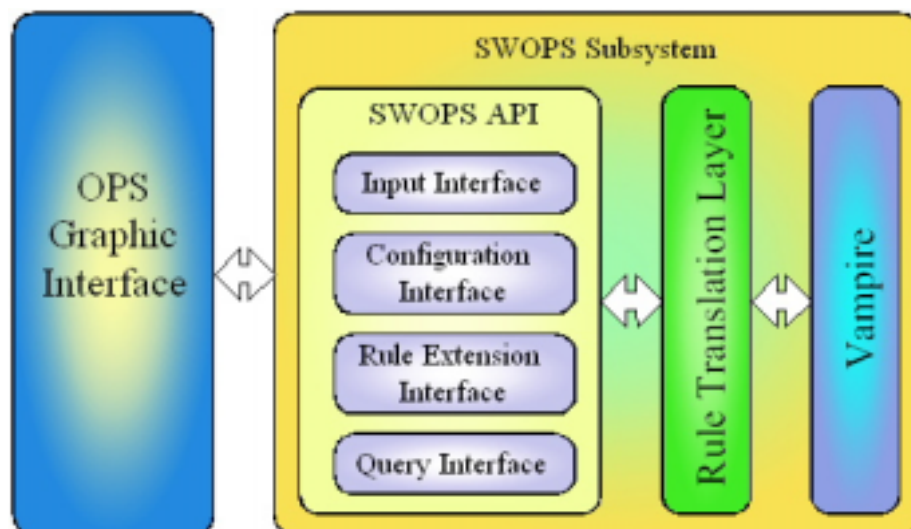


Figure 4: Architecture of SWOPS

First, Vampire accepts TPTP format files as input. The file-input-only interface makes it awkward for semantic queries since for each new query (even for the same query over the same ontology) a new problem file is required before processing the new query. This is very expensive if the input file is very large. Even for a small input file, if the number of queries is very large, the overall cost of input is still unacceptable. We partially improve this by refactoring portions of Vampire and wrapping it in the interface we developed so that our

Java based SWOPS can communicate with Vampire directly without using an intermediate problem file as input and screen print out as output (see Fig.4).

The second problem with Vampire is that it can only answer whether the input problem is *proved*, *unproved* and *unknown*. This means that it can only return one answer for each execution. For those queries that have multiple answers, Vampire must execute as many times as the number of answers. We modified Vampire by rearranging its proving procedure so that it can generate all the answers in one execution. In the improved version, Vampire tries to find the next answer for the query recursively until no further answer can be found. This modification guarantees that all the answers can be generated. These two modifications improve the performance of SWOPS for ontology reasoning and make it possible to compare SWOPS with DL reasoning systems.

Vampire was developed in C++. To communicate with Vampire from Java, we developed an adapter via Java Native Interface (JNI). This adapter substitutes the command line interface Vampire originally provides. As shown in Fig. 4, the JNI adapter permits the rule translation layer to communicate with Vampire directly. The translation layer interacts with the JNI adapter to control the behavior of Vampire prover and also provide the I/O method.

The graphic user interface plug-in of SWOPS is similar to that of ROPS. However, the interface of SWOPS is simpler, since Vampire does not provide as much choice as Jena2. The functions provided by these sub-interfaces are shown below.

The configure sub-interface of SWOPS is very simple. It is actually combined with the query interface. Though Vampire provides several options to optimize the performance, only a few of them are useful for improving the performance, primarily *running time limitation* and *memory limitation*.

The *running time limitation* is the most important option to control Vampire. Although Vampire is outstanding as a FOL prover, it is not guaranteed to always terminate, so setting a maximum running time is necessary. The *memory limitation* is very similar to time limitation. Users can adjust this value to optimize the usage of system memory.

The queries of SWOPS are in Datalog-like syntax. An example of the Datalog-like query similar to the one shown in Section 3.3 looks like the following:

```
?x := father(?x, ?y), cars(?y, ?z), carType(?z,truck).
```

The SWOPS query parser we built will parse the Datalog-like query into TPTP format query. The TPTP format is required by the Vampire prover. The queries in TPTP format look like the following:

```
input_formula(axiom_query,axiom,(subClassOf(male,X))).
```

This query asks about the *subclass* of class *male*. In the TPTP format, notations starting with

uppercase represent variables and notations starting with lowercase represent instances. In SWOPS, we can query about the following:

- instances of a *class*,
- *subSumption* relationships,
- satisfaction of a statement, and
- *equivalence* or *inEquivalence* of two instances.

6 Benchmarking Results

The LUBM (Lehigh University Benchmark) [30] was chosen to compare the performance of these ontology query systems. The benchmark test ontology generated by LUBM is in OWL (or DAML+OIL) syntax. LUBM can generate testing ontology automatically based on the initial setting provided by users. These ontology test cases are based on the real information from Lehigh University. All test cases generated by LUBM have a similar schema, but have various numbers of instances. LUBM also provides a set of 14 queries with different complexity levels for the benchmark test.

Our tests were conducted on a WinXP system running on 1.6GHz Pentium 4 processor with 512MB of memory. In order to reduce inaccuracy, only necessary applications are loaded during the testing. In our tests, the ROPS subsystem was built on Jena2.2. The tested version of RACER system is 1.7.24. The SWOPS subsystem was built on Vampire 7.0.

Table 2: Brief Information about five Test Cases

* The size of this file is measured before modification to be compatible with all the systems

	Case 1	Case 2	Case 3	Case 4	Case 5
Class #	41	41	41	41	41
Property #	24	24	24	24	24
Class Instance #	112	401	1084	6973	20659
Property Instance #	383	1470	3844	28039	82415
Size (KB) *	81.3	257	659	4360	8050

In our test, we created five test cases with increasing number of instances using LUBM. The details are shown in Table 2. The schemas of the five test cases are all the same. The schema consists of hierarchical relationships, equivalent classes, existential restrictions, intersections, and inverse relationships. There are no datatype restrictions, transitive properties, or cardinality restrictions. Since the OWL-QL system is not available, we did not compare it in the benchmark test.

The queries provided by LUBM do not require the power of SWRL. To illustrate the power of SWRL, we switched the order of query 13 with query 14, and added two more that could only be processed by the SWOPS subsystem. The new query set is shown in Table 3.

These queries were translated into different query languages for each system tested. There are three of them: RDQL, nRQL, and SWRL in a Datalog-like human readable syntax. In addition, queries may be specified in TPTP which can be directly given to Vampire. In table 3, each of these languages expresses four queries in group order (e.g., 1 to 4 are in RDQL).

Table 3: Modified Benchmark Query Set

In this table, the name space “NS” = “http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl”

	Query
Q1	SELECT ?x WHERE (?x, <NS#takesCourse>, <Dept0/GraduateCourse0>) (?x, <rdf: type>, <NS#GraduateStudent>)
Q2	SELECT ?x, ?y, ?z WHERE (?x, <NS#memberOf>, ?z) (?z, <NS#subOrganizationOf>, ?y) (?x, <NS#undergraduateDegreeFrom>, ?y) (?x, <rdf: type>, <NS#GraduateStudent>) (?y, <rdf: type>, <NS#University>) (?z, <rdf: type>, <NS#Department>)
Q3	SELECT ?x WHERE (?x, <NS#publicationAuthor>, <Dept0/AssistantProfessor0>) (?x, <rdf: type>, <NS#Publication>)
Q4	SELECT ?x, ?y, ?z, ?w WHERE (?x, <NS#worksFor>, <Dept0>) (?x, <NS#name>, ?y) (?x, <NS#emailAddress>, ?z) (?x, <NS#telephone>, ?w) (?x, <rdf: type>, <NS#Professor>)
Q5	(retrieve (?x) (and (?x NS#Person) (?x Dept0 NS#memberOf)))
Q6	(retrieve (?x) (?x NS#Student))
Q7	(retrieve (?x ?y) (and (?x NS#Student) (?y NS#Course) (Dept0/AssociateProfessor0 ?y NS#teacherOf) (?x ?y NS#takesCourse)))
Q8	(retrieve (?x ?y ?z) (and (?x NS#Student) (?y NS#Department) (?x ?y NS#memberOf) (?y University0 NS#subOrganizationOf) (?x ?z NS#emailAddress)))
Q9	?- teacherOf(Y, Z), takesCourse(X, Z), advisor(X, Y), Student(X), Faculty(Y), Course(Z)
Q10	? - Student(X), takesCourse(X, Dept0_GraduateCourse0)
Q11	?- ResearchGroup(X), subOrganizationOf(X, University0))
Q12	?- Chair(X), Department(Y), worksFor(X, Y), subOrganizationOf(Y, University0)
Q13	input_formula(axiom_query, axiom, (~(UndergraduateStudent(X)))).
Q14	input_formula(axiom_query, axiom, (~(Person(X) & hasAlumnus(wwwXUnivOxedu, X)))).
Q15	input_formula(axiom_query, axiom, (~(TAandRA(X)))).
Q16	input_formula(axiom_xxx, axiom, (![X, Y]: ((UndergraduateStudent(X) & GraduateCourse(Y) & takesCourse(X, Y)) => (busyOn(X, Y)))). input_formula(axiom_query, axiom, (~(busyOn(X, Y)))).

ROPS can process in both RDFS reasoning mode (RDFS-mode) and OWL reasoning mode (OWL-mode). However, in the RDFS-mode, the reasoner can correctly entail RDFS recognizable statements. As a result, the queries that can be correctly performed in RDFS-mode are query 1, 2, 3, 4, 5, 7, 11, 12 and 13.

Query 6, 8, 9, and 10 cannot be directly processed by ROPS using the RDFS reasoner. These queries all consist of axioms about class *Student*. In the testing ontology, implicitly defined class *Graduate Student* (defined via an equivalent class) should be recognized as a subclass of class *Student*. However, the reasoner in the RDFS-mode can not recognize this. It can only return answers from instances that belong to class *Undergraduate Student*, which is explicitly defined as a subclass of class *Student*.

However, these problems can be solved in this test case by adding a statement that explicitly defines the class *Graduate Student* to be a subclass of class *Student*. With this modification, these four queries can be processed correctly by ROPS in RDFS-mode. Note, this solution may not be applicable in other case. The running times for query 6, 8, 9, and 10 of ROPS in RDFS-mode shown in table 4 to 8 are the running times on the modified ontology.

ROPS in RDFS-mode could not execute query 14 since inverse properties can not be recognized by the RDFS mode reasoner. To fix this problem, we had to generate a new query by replacing the predicate in the original query with the corresponding inverse predicate. The combination of sets of answers generated from these two queries is the final correct answer set. However, since RDQL does not support disjunction, so we can not combine the new query with the original one. This means ROPS can not process query 14 in one step. As a result, we did not compare the running time of ROPS in RDFS-mode for query 14. The running times shown in the tables in parenthesis are those queries can not correctly answered by the query systems.

We extended the query set with query 15 and query 16 to distinguish the power of tested systems. For query 15, we added a new class *TAandRA* to those ontology testing cases. The class *TAandRA* was defined by the *unionOf* class *TeachingAssistant* and *ResearchAssistant*. Query 15 asks about all the instances of class *TAandRA*. ROPS in both RDFS-mode and OWL-mode can not process this query correctly, since Jena2 does not support *unionOf*. Compare to ROPS, RACER can correctly answer this query. Query 16 involves composition of properties. This query first defines a new property *busyOn* via the rule extension mechanism of the SWOPS system. This query is only supported by SWRL. All the other systems can not process this query since OWL does not support composition of properties.

Tables 4 to 8 show the benchmarking results on the five test cases. The running time is recorded separately for each query. These queries are executed consecutively after the testing ontology was loaded. Since only SWOPS can process query 16, the total running time is the sum from queries 1 to 15.

Table 4: Testing results for Case 1 (in seconds)

*The running time marked in the column of ROPS with RDFS Reasoner is the running time on the modified ontology.

	ROPS with RDFS Reasoner (RDQL)	ROPS with OWL Reasoner (RDQL)	RACER (nRQL)	SWOPS (SWRL)
Loading <i>t</i>	1.132	1.212	0.641	0.9
Q1	0.04	11.977	0.341	1.2
Q2	0.09	0.04	0.04	1.2
Q3	0.01	<0.001	0.04	1.2
Q4	0.01	<0.001	0.03	1.2
Q5	0.03	0.01	0.14	1.5
Q6	0.03*	0.01	0.04	1.9
Q7	<0.001	<0.001	0.01	1.5
Q8	0.02*	0.02	0.02	1.8
Q9	0.02*	0.21	3.665	1.3
Q10	0.01*	0.01	0.03	1.2
Q11	<0.001	<0.001	0.02	1.2
Q12	<0.001	<0.001	4.998	1.2
Q13	<0.001	<0.001	4.977	1.2
Q14	(<0.001)	<0.001	5.017	1.2
Q15	(<0.001)	(<0.001)	0.03	1.2
Total	1.441	13.216	19.387	19.7
Q16	-	-	-	1.2

In test case 1, the size of the test ontology is the smallest. We can see from Table 4 that ROPS has the best performance. It was faster than the other systems in both the OWL-mode and RDFS-mode. Particularly, in RDFS-mode, we can see most of the time was actually spent on loading and creating model. The RDFS mode reasoning of ROPS is about ten times faster than the others in this test case. Even with OWL reasoner, ROPS is still faster than RACER and SWOPS.

RACER is slower than ROPS in this case. However, the difference is not big. SWOPS is barely slower than RACER. Considering that Vampire is not optimized as a query reasoner and must repeat reasoning process once for each query, the potential performance of SWOPS as a FOL query processor can be as fast as the others.

In Table 4, there is an interesting scenario that most of the time ROPS spent in OWL-mode was on the first query. This is because that the reasoning strategy of Jena2 is to derive all the additional statements from original ontology at once. It does not matter

whether the statements are related to current query. Based on this strategy, the following queries can generate answers without or with very little cost on additional reasoning. Taking advantage of the optimized indexing technique of Jena2, the following queries can be processed very fast. This reasoning strategy makes ROPS efficient when processing a large number of queries on the same ontology.

Table 5 shows the result of test case 2. In this case, the input ontology is about 3 times as large as that in test case 1. The running time of ROPS in OWL mode increased more than 6 times. Apparently, ROPS in OWL-mode is the slowest in test case 2. The running time of ROPS in RDFS-mode is still the fastest. Its running time almost did not change when the size of input ontology increased. This shows that the OWL-mode reasoner of ROPS is not very scalable and efficient. However, the RDFS-mode reasoner is very efficient for those queries that can be processed correctly.

Table 5: Testing results for Case 2 (in seconds)

	ROPS with RDFS Reasoner	ROPS with OWL Reasoner	RACER	SWOPS
Loading t	1.312	1.402	1.091	2.6
Q1	0.05	96.682	0.871	3.9
Q2	0.04	0.05	4.997	3.8
Q3	0.06	0.02	0.04	4
Q4	<0.001	<0.001	0.21	4
Q5	0.02	0.01	0.621	6.9
Q6	0.06*	0.02	1.272	6.7
Q7	0.01	<0.001	4.987	4.3
Q8	0.01*	0.01	0.09	6.2
Q9	0.03*	0.942	0.101	4.1
Q10	<0.001*	<0.001	0.02	3.9
Q11	0.01	<0.001	4.736	3.9
Q12	<0.001	0.01	4.988	3.8
Q13	<0.001	<0.001	0.02	3.8
Q14	(<0.001)	<0.001	5.037	4.2
Q15	(<0.001)	(<0.001)	0.22	4.1
Total	1.713	99.13	28	66.1
Q16	-	-	-	3.9

The running time of RACER increases slowly compared to the increasing input size. It ran much faster than ROPS in OWL-mode, but not as fast as ROPS in RDFS-mode. However, ROPS in RDFS-mode can only correctly process 9 out of 14 queries (those queries that can only be correctly processed on the modified ontology are considered as

failed; SWRL only queries are not counted) compared to that of all 14 queries that can be processed by RACER. In this case, RACER can be considered as the winner.

SWOPS also demonstrated decent performance in this test case. Although it is slower than RACER, it is faster than ROPS in OWL-mode. The ratio between the increments of running time and input size is linear from test case 1. We can say that the performance of SWOPS is satisfying in this test case.

Table 6 shows the results of test case 3. The table shows that ROPS in OWL-mode is the slowest and ROPS in RDFS-mode is the fastest in this case. However, in OWL-mode, after finishing in the first query, all the following queries are very fast. Obviously, the query strategy of Jena2 is to provide best performance for large amount of queries about the same stable ontology.

Table 6: Testing results for Case 3 (in seconds)

	ROPS with RDFS Reasoner	ROPS with OWL Reasoner	RACER	SWOPS
Loading t	1.632	1.753	3.155	6.8
Q1	0.06	229.56	4.095	10.0
Q2	0.05	0.05	0.541	10.0
Q3	0.09	0.04	1.983	10.2
Q4	0.01	0.01	0.27	10.1
Q5	0.01	<0.001	0.581	10.3
Q6	0.131*	0.02	0.201	10.6
Q7	0.03	0.01	4.897	10.7
Q8	<0.001*	0.02	0.921	14.8
Q9	0.16*	0.291	3.124	10.6
Q10	0.01*	0.01	4.988	10.0
Q11	<0.001	<0.001	4.987	10.0
Q12	<0.001	0.01	5.037	10.1
Q13	<0.001	0.01	0.261	10.0
Q14	(<0.001)	<0.001	4.977	13.9
Q15	(<0.001)	(<0.001)	0.11	9.0
Total	2.193	231.784	36.893	158.1
Q16	-	-	-	8.9

RACER has a default lazy reasoning strategy that it provides good performance for a single query. The processing time of each query depends on the complexity of the query and whether the necessary information has already generated. We can tell from the running times of queries 9 and 10. The information to answer the query 10 has already been used in

query 9. There is no reasoning needed for query 10. In this test case, RACER spent some time on this query. Although in test case 2, this time is short. For ROPS, none of queries (except the first) takes more than half a second.

In this test case, SWOPS is slow, as it is still faster than ROPS in OWL-mode. It seems that SWOPS is not as scalable as RACER. In addition, there is an interesting scenario that the running time of each query did not change much whether the query is complex or simple, whether the return answer size is big or small. It is most likely that Vampire spent a certain amount of time on preprocessing the ontology before processing the query. If the FOL reasoner can be optimized so that the duplicated reasoning process for each query can be avoided, the performance of the FOL reasoner can be improved.

Table 7: Testing results for Case 4 (in seconds)

	ROPS with RDFS	RACER		ROPS with RDFS	RACER
Loading t	4.276	17.456	Q9	1.782*	14.28
Q1	0.05	445.801	Q10	0.09*	0.03
Q2	0.09	24.595	Q11	<0.001	0.06
Q3	0.21	0.661	Q12	<0.001	0.02
Q4	0.01	0.882	Q13	<0.001	0.03
Q5	0.02	7.781	Q14	(<0.001)	5.779
Q6	0.16*	7.07	Q15	(<0.001)	0.34
Q7	0.1	0.471	Total	6.839	539.867
Q8	0.03*	31.906			

Table 7 shows the result of test case 4. SWOPS failed in this case. The reason is the limitation of the Vampire prover. It is not designed to process this large input sizes (in this case 6973 class instances and 28038 property instances).

ROPS in OWL-mode failed too in this test case. It occupied more than 700 MB memory during the execution, as there were extensive requests of memory page exchange to the disk. The CPU utilization is not very high since most of the time is spent on disk access. We stopped its execution after more than an hour of execution and considered ROPS in OWL-mode to be failed in this test case. If there had been more memory, ROPS in OWL-mode could have finished this test case.

RACER could still generate correct answers, but it was very slow for some queries. In addition, RACER required too much memory. The heap usage of memory was nearly 1GB. This amount is much bigger than ROPS in OWL-mode. However, it seems that RACER is optimized on the data storage in the physical memory. The disk access requests during the

execution were not as extensive as that of ROPS. As a result, it could finish in about ten minutes and ROPS could not finish in an hour. It seems RACER is a little better on scalability than ROPS in OWL-mode.

Table 8: Testing results of ROPS on Case 5 (in seconds)

Loading	Q1	Q2	Q3	Q4	Q5	Q6	Q7
8.182	0.05	0.2	0.421	0.01	0.02	0.14	0.231
Q8	Q9	Q10	Q11	Q12	Q13	Q14	Total
0.03	9.994	3.555	<0.001	<0.001	<0.001	(<0.001)	24.20

ROPS in RDFS-mode still worked well in this test case on those queries it could handle. It is the most scalable one in all the test cases. Table 8 shows the results of test case 5. In test case 5, RACER failed too due to insufficient memory. ROPS under RDFS-mode was the only one that can work. Its performance is still very good. In addition, the memory occupied by ROPS in RDFS-mode is very small compared to RACER. This test case proves that ROPS in RDFS-mode is very efficient and scalable for simple ontology and queries.

Table 9: Benchmark Query Set on Family Ontology

In this table, the name space “FN#” = “http://owl.man.ac.uk/ontologies/family#”

	Query
Q1	SELECT ?x WHERE (?x, <rdf: type>, <FN#Person>)
Q2	SELECT ?x, ?y WHERE (?x, <rdf: type>, <FN#Person>) (?x <FN#hasParent>, ?y)
Q3	(retrieve (?x ?y) (and (?x FN#Male) (?y FN#Female) (?x ?y FN#hasSiblings)))
Q4	(retrieve (?x ?y) (and (?x FN#Male) (?y FN#Female) (?x ?y FN#hasMother)))
Q5	?- Male(X), Female(Y), Person(Z), hasMother(X, Z), hasBrother(Y, Z)
Q6	? - Father(X)
Q7	input_formula(axiom_query, axiom, ((¬(Happy(X)))).
Q8	input_formula(axiom_xxx, axiom, (![A, B, C]: ((hasParent(A, B))=>(hasAncestor(A, B)))). input_formula(axiom_query, axiom, ((¬(hasAncestor(X, Y)))).

In the benchmark test cases, one problem may affect the applicability of the results, i.e., all the test cases have the same schema. Unlike ROPS and RACER, SWOPS is built on Vampire FOL prover, and is powerful on complex, but small ontology. Also, SWRL is more

powerful than DL and can provide the ability to compose new properties which is impossible for OWL. The testing ontology and queries provided by the benchmark package are not complex enough to show this difference. For this reason, we created a new test case which is a family ontology with a more complex schema and smaller instance size.

The family ontology contains fewer class and property instances than that of LUBM. However, there are much more complex relationships such as *equivalence*, *intersection*, *inversion*, and *someValueFrom*. The family ontology also contains *symmetricProperties*. In addition, it consists of a much more complex hierarchical structure. There are 24 classes and 18 properties. The number of instances is 121. For benchmarking, we create 8 queries simulating those in LUBM in both the complexity and format. The queries in this test case can be considered corresponding to queries 14, 6, 3, 5, 7, 9, 15, and 16 in LUBM query set, respectively. The queries are shown in Table 9. Similarly to Table 3, the queries are written in four languages. ROPS in RDFS mode can only process query 1, 2, and 4 and can not process correctly more than half of the queries. Queries 3, 5, 6, and 7 all involve axioms about equivalent classes or union of classes. Query 8 can only be processed correctly by SWOPS since it involve rule extension.

Table 10: Testing result on family ontology

Query	ROPS with RDFS Reasoner*	ROPS with OWL Reasoner	RACER	SWOPS
Loading t	0.971	-	0.55	0.2
Q1	0.06	-	1.211	0.4
Q2	0.02	-	0.07	0.4
Q3	(0.01)	-	0.051	0.4
Q4	0.01	-	0.02	0.4
Q5	(0.01)	-	4.887	0.4
Q6	(0.01)	-	4.927	0.4
Q7	-	-	0.03	0.5
Q8	-	-	-	0.5
Total	1.091	-	11.697	2.6

Table 10 shows the testing result for the family ontology. ROPS in OWL reasoning mode kept on computing without any output for more than half an hour. We consider it as failed. The RDFS mode reasoner could still work. However, it can only handle 3 out of 8 queries. The running times of the queries can not be handled by ROPS in RDFS mode are shown in the parenthesis. We can see that SWOPS performed very well in this test case. It is faster than RACER. Apparently, SWOPS is very efficient on small and complex reasoning tasks. On the contrary, RACER is not very efficient in this test case. RACER can not handle query 8.

To get a clear evaluation, we expand the size of the family ontology to 3 times. There are

396 individuals in this ontology. The depth of the hierarchical structure of this test case is 2 times deeper than that of the previous test case. The testing result of this test case is shown in Table 11.

In this test case, SWOPS is slower but was still the fastest. It is a little slower than the previous test case. This is possibly because SWOPS spent too much time loading the system and preprocessing the input ontology. The increasing time of reasoning is a small portion compared to the total running time.

Table 11: Testing result on *family* ontology

Query	ROPS with RDFS Reasoner*	ROPS with OWL Reasoner	RACER	SWOPS
Loading t	1.132	-	0.671	0.3
Q1	0.07	-	2.343	0.5
Q2	0.04	-	0.381	0.6
Q3	-(0.08)	-	0.02	0.5
Q4	0.1	-	0.07	0.5
Q5	-(0.071)	-	5.007	0.5
Q6	-(8.692)	-	4.356	0.7
Q7	-	-	0.09	0.9
Q8	-	-	-	0.7
Total	10.275	-	12.948	5.2

RACER is almost as fast as it was in the previous test cast. This shows that RACER needs some time to initialize before executing the query. It is also more scalable than other systems, but not as efficient on complex queries. This can be figured out by comparing the running time of RACER in this test case with that in other test case. The ROPS in RDFS mode is slower even without generating correct answer. This is probably because that the complex hierarchy relationships in this test case. The depth of the hierarchy relationship is up to 9 comparing to that is 3 in the previous test case.

From all the testing results, we identified the advantages and disadvantages of these systems. ROPS in RDFS-mode is very efficient on large and simple ontology querying. Its OWL-mode reasoner can only process small ontology. It also cannot handle complex ontology querying.

Though RACER is not as fast as Jena in RDFS-mode when performing simple queries and cannot deal with large size of ontology, it provides relatively good scalability and performance. It is not as good as FOL on very complex queries. However, the overall performance is still satisfying.

SWOPS, a FOL based querying system, can handle very complex ontology querying and still provide good performance, but because of limitations of Vampire, SWOPS is not that

scalable. Vampire is not designed for FOL query processing. Nevertheless, the performance of SWOPS in the tests is enough to prove that FOL systems have potential to be efficient for complex ontology querying.

7 Related Work

There have been several studies on ontology languages for the Semantic Web. For example, in [16], Ian Horrocks gives a valuable picture of the development history and the relationships among current languages. However, there are few investigations about ontology query language comparisons and evaluations.

There are several query systems not evaluated in this paper. The Fast Classification of Terminologies (FaCT) is a DL reasoning system based on the optimized tableaux algorithm [24]. It contains two reasoners, one supporting DL *SHF* and another supporting DL *SHIQ* [31]. Now a new generation of FaCT, FaCT++ is available. The FaCT reasoner does not provide a combined query language like the other systems in this paper, so we did not include it in our benchmark test. In [21], they provide a performance evaluation using LUBM benchmark too. But they did not compare with other query systems.

In [29], Dmitry Tsarkov, Alexandre Riazanov, Sean Bechhofer and Ian Horrocks give a comparison between FaCT++ and Hoolet⁹. Hoolet uses the Vampire FOL prover to implement an OWL-DL reasoner. It can be also be extended to process SWRL reasoning. The purpose of this paper is to verify the potential of FOL engine for ontology reasoning. The evaluation of ontology languages and query languages are not mentioned. The version of Vampire was not optimized at that time, so the result is not as promising as our results.

In [32], F-OWL, an ontology query language and its query system is based on FLORA-2¹⁰. F-OWL is intended to support OWL. However, there was no substantial result available now. FLORA-2 uses XSB as the query engine. XSB is a deductive database using tabling technique to resolve recursive queries. A study in [33] indicated that it is suitable for Description Horn Logic reasoning, although it need some translation before processing subsumption problems.

In [34], the FORTH Institute of Computer Science developed ICS-FORTH RDFSuite, which provides RDF validation, storage, and querying (using RDF Query Language -- RQL). ICS-FORTH RDFSuite uses an object-relational DBMS to store RDF(S) statements and takes the advantage of the technique of DBMS so that it can process very large data set. The RQL language it provided for querying is SQL-like and supports complex queries. We did not test it since it can not process OWL ontology and it has already provided a performance test.

⁹ Hoolet implements an OWL-DL reasoner using Vampire FOL prover. The website of Hoolet is <http://owl.man.ac.uk/hoolet/>.

¹⁰ <http://flora.sourceforge.net/docs/releasemanual.pdf>

In [30], the developers of the LUBM benchmark evaluated the performance of four knowledge base systems (KBS). The target systems they studied are different from those in our test and their intention for the benchmark test is different as well. Their test is focused on the scalability and performance of the chosen systems. These systems can only support reasoning tasks for partial OWL Lite. In our test, the systems we selected all support reasoning tasks at least as complex as OWL Lite. Our intention is to test query systems for ontology with rich semantics and complex hierarchical structure. In addition, the systems we tested all have query language available. This is important for practical users. This is not considered in that paper and as a result, there is no evaluation of query languages for those systems. In addition, in order to test large data set, they processed the queries separately. For Database-based systems, this is a usual approach. However, for complex ontology reasoning, reasoning systems like Jena2, FaCT++, and RACER all buffer the statement generated by earlier queries so that the following queries can use this information and improve performance. This can be clearly seen from our test result. In our approach, we focused not only on the performance, but also on other additional features related to real world usage. For example, the flexibility of I/O function, ontology languages supported, user friendliness and the power of query language used, the reasoning power of a system, the flexibility of user control, and so on. All these features can affect users' intention while choosing a system for a certain task.

8 Conclusion and Future Work

From the study of these ontology languages, we can conclude that RDF(S) is limited in representing real world information. The advantage of RDF(S) is that it is very simple and can be very efficient for reasoning.

OWL is a W3C recommended standard. It provides very flexible and good expressive powers. The standardization also makes it accessible and computable by computers and also human readable. It introduces a very attractive idea to design a family of sublanguages with different levels of expressive power and computational complexity. This gives users flexibility to select a proper ontology language for different information representation tasks. Another advantage is that it can easily upgrade existing ontology to a higher level of sublanguage. For example, users can start creating ontology in OWL Lite, since it is much easier to understand and manipulate. When users become more familiar with OWL Lite and the ontology requires more powerful features, users can smoothly upgrade ontology from OWL Lite to OWL DL. However, users should consider the increasing complexity before upgrading. Especially for OWL Full, since it is undecidable, it is difficult to find an efficient reasoner that can generate correct answers.

SWRL is still under development, but its expressive power and flexible and simple syntax make it very promising as an ontology language. As we mentioned in section 2.4, SWRL is very flexible and powerful in representing complex relationships. In addition, the

rule based pattern provides SWRL the potential to combine with traditional databases that are still dominating the information processing field. All these database systems are somewhat based on subsets of FOL (e.g., Relational Calculus). As a result, SWRL will be more compatible with and much easier to be combined with current database systems. The disadvantage of SWRL is its complexity (it is semi-decidable [6]). However, researchers are considering using a sublanguage strategy (as used for OWL) to solve this problem, by optimizing them more for query processing.

Each query system compared in this paper has its own advantages and disadvantages. RDQL is a flexible query language for RDF and OWL. Its SQL like syntax makes it very attractive to users. However, it can only provide limited schema (T-box) querying mechanism and more complex schema queries (such as direct subclass) have to be processed by calling Jena2 directly. The lack of disjunction in RDQL is another problem. ROPS built on Jena2 is the best solution for querying ontology with simple schema and large scale. For some queries beyond the reasoning scope of RDFS mode reasoner, we can still guarantee correct and fast answers by modifying the ontology and queries. Using the OWL mode reasoner, ROPS can process small ontology with moderately complex schema.

The Jena2 package also provides flexible interfaces for other ontology processing systems such as Protégé, RACER, and FaCT. By communicating with RACER or FaCT, Jena2 can enhance its ability for reasoning on large and complex ontology. Jena2 also provides interfaces for several database systems such as MySQL, Oracle and PostgreSQL. This provides Jena2 the ability to process information stored in these databases without having to translate and store the information.

Overall, RACER is the fastest system for OWL. It is also relatively scalable. It is efficient for reasoning on complex ontology with relatively large scale. It is better to exploit RACER for repeated queries on part of the ontology as we mentioned before. RACER also provides a powerful query language nRQL to support complex queries. With the complex queries, users can retrieve more information than the other DL systems. This makes RACER a better choice for the users dealing with complex semantic data and expecting more information from the ontology. The drawback of nRQL is as for RDQL, it does not provide schema querying mechanism currently (it is claimed to be available soon).

Our purpose of building SWOPS using Vampire is to test the potential ability of a FOL reasoner for ontology querying. From the testing result, it is satisfying that SWOPS performs very well when dealing with very complex, but relatively small ontology. Since it is not a specifically designed system for SWRL, its performance can be improved. As we discussed in section 6, it is possible that a FOL reasoner can be a DL reasoner. Although there are some limitations, it has been verified that SWOPS using Vampire is a fairly efficient OWL and SWRL reasoner by this study. It is a very good reasoning system for complex ontology.

The main contributions of this work are that we evaluated current ontology query

languages and query processors, systematically. The evaluation focused on their ability and performance. In the evaluation, we discussed the advantage and disadvantage of each query language processor. We also discussed the type of tasks that they are efficient for. For the ontology query languages and supporting systems, we compared them in the querying power and performance. We also discussed the functions they provided for particular tasks. Finally, we point out in what situations the query systems can be most efficient.

In the future, we will revise the OPS system to combine its two subsystems more tightly. We will also combine Jena2 with a more powerful query language, SPARQL [20], to provide more complex queries. To get more reasoning power, we plan to embed RACER into ROPS via the plug-in provided by Jena.

CHAPTER 3

CONCLUSION AND FUTURE WORK

In this thesis, we studied several languages for the Semantic Web. RDFS is a framework that provides limited expressiveness for representing metadata. The advantage of RDFS is that it is very simple and can be very efficient for reasoning.

OWL is a successful ontology language which is recommended by W3C. It provides standardized syntax and is downward compatible with RDFS. That is anything represented in RDFS can be translated into OWL. To overcome the complexity problem, OWL introduces a very attractive idea to design a family of sublanguages with different levels of expressive power and computational complexity. Users can flexibly select a proper sublanguage for a specific representation task. Another advantage of this strategy is that users can easily upgrade existing ontology into higher level of sublanguages. For example, users can start creating ontology in OWL Lite since it is much easier to understand and manipulate. When users become more familiar with OWL Lite and the ontology requires more powerful features, users can smoothly upgrade ontology from OWL Lite to OWL DL. For those who want even more expressive power, the process of upgrading from OWL DL to OWL FULL is very similar. However, users should consider the increasing complexity before upgrading. Especially for OWL Full, since it is undecidable, it is difficult to find an efficient reasoner that can generate correct answers.

SWRL extends OWL DL with Horn like rules. This rule extension enriches OWL DL with ability to define more complex relationships. The rule extension also provides SWRL

the potential to be easily combined with traditional databases that are still dominating the information processing field, since these database systems are all related with First Order Logic (FOL). Cooperation with database will also makes SWRL more scalable by using database system to store and retrieve information not suitable to be loaded into memory. The drawback of SWRL is its complexity. It is for sure to be undecidable. However, the sublanguage strategy (as used for OWL) and improve existing FOL query engines by optimizing them for query processing can help solve this problem.

We also evaluate the ontology query languages and implementing systems. The nRQL query language is very powerful and flexible. It supports compound query and negation as failure to build complex queries. It also supports complex queries with datatypes which is usually weak in ontology query system. The RACER system using nRQL is the most powerful Description Logic (DL) reasoner available currently. In the benchmark test, RACER is the fastest system for OWL in most of the case. It can process relatively complex ontology and is also scalable compared to ROPS in OWL mode. RACER uses lazy reasoning strategy by default. This may save time when frequently switching among different ontology. The drawback of RACER is that it uses large amount of memory and for repeated query, it is not as fast as ROPS..

The ROPS subsystem in OPS is good at small ontology. RDQL is a simple and flexible query language. Although it does not support compound queries it can easily retrieve almost all the information in the model including schema data. The advantage of ROPS is its

multiple reasoner strategy. This makes it suitable for most of the query task. In the test, ROPS can perform very well for queries over simple and large ontology. It can also process complex ontology, but may not be very efficient. ROPS is also very robust for input format. This is a big issue since to create an ontology in the proper version of syntax is a challenge for most of users. In addition, Jena2 provides interfaces to communicate with many other applications and database products, e.g., Protégé, MySQL, and ORACLE. Jena2 can also embed RACER, FaCT, and FaCT++ as imported reasoners. This means it can do everything these reasoners can do with similar performance.

The SWOPS subsystem performs very well after we optimize it for ontology queries. It can handle very complex, but relatively small ontology and runs faster than RACER and ROPS. Since it is not a specifically designed system for SWRL, its performance can be further improved. SWOPS is also more powerful for reasoning. Although there are some limitations, it has been verified that SWOPS using vampire is a very efficient OWL and SWRL reasoner by this study. The study of SWOPS proves the potential to query about ontology using FOL query engine. The rule extension also makes SWOPS very flexible to build complex queries.

REFERENCE

- [1] McBride, P.H.a.B., *RDF Semantics*. W3C, 2004.
- [2] Mike Dean, G.S., Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider and Lynn Andrea Stein, *OWL Web Ontology Language Reference*, ed. W3C. 2004.
- [3] Ian Horrocks, P.F.P.-S., Harold Boley, Said Tabet, Benjamin Grosz and Mike Dean, *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. available at <http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/>, 2004.
- [4] Horrocks, I., *DAML+OIL: a Description Logic for the Semantic Web*. the IEEE Computer Society Technical Committee on Data Engineering, 2002. 25(1): pp. 4-9.
- [5] Michael K. Smith, C.W.a.D.L.M., *OWL Web Ontology Language Guide*. W3C, 2004.
- [6] Becker, M.Y. *Godel's Completeness Theorem*. in *Computer Laboratory University of Cambridge*. 28.2.2003.
- [7] Seaborne, A., *Jena Tutorial A Programmer's Introduction to RDQL*. HP Labs, 2002.
- [8] Richard Fikes, P.H., and Ian Horrocks, *OWL-QL – A Language for Deductive Query Answering on the Semantic Web*. KSL 03-14, 2003.
- [9] Voronkov, A.R.a.A., *Vampire 1.1*. IJCAR, 2001. LNAI 2083: pp. 376-380.
- [10] Palmer, S.B., *The Semantic Web: the introduction*. W3C, 2001.

- [11] Frank Manola, E.M.a.B.M., *RDF Primer*. W3C, 2004.
- [12] Brian McBride, J.G.a.D.B., *RDF Test Cases*. W3C, 2004.
- [13] Graham Klyne, J.J.C.a.B.M., *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C, 2004.
- [14] Ratnakar, Y.G.a.V., *A Comparison of Markup Languages*. 15 th International FLAIRS Conference, 2002.
- [15] D. Fensel, F.v.H., I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider., *OIL: An ontology infrastructure for the semantic web*. IEEE Intelligent Systems, 2001. 16(2).
- [16] Ian Horrocks, P.F.P.-S.a.F.v.H., *From SHIQ and RDF to OWL: The Making of a Web Ontology Language*. Journal of Web Semantics, 2003. 1(1):7-26.
- [17] Harold Boley, M.D., Benjamin Grosf, Michael Sintek, Bruce Spencer, Said Tabet, Gerd Wagner, *FOL RuleML: The First-Order Logic Web Language*. Available at <http://www.ruleml.org/>, 2004.
- [18] Dean, B.G.a.M., *DAML Rules Report for PI Mtg*. 2004.
- [19] Ruckhaus, E., *Efficiently Answering Queries to DL and Rules Web Ontologies*. W3C, 2004.
- [20] Seaborne, E.P.h.a.A., *SPARQL Query Language for RDF*. 2005.
- [21] Volker Haarslev, R.M.o., and Michael Wessel, *Querying the Semantic Web with Racer + nRQL*. In Proceedings of the KI-2004 International Workshop on ADL'04, 2004.
- [22] Franz Baader, I.H., Ulrike Sattler, *Description Logics for the Semantic Web*. KI - Künstliche

- Intelligenz, 2002. 16(4):57-59.
- [23] Möller, V.H.a.R., *Proceedings of the International Workshop on Description Logics*. DL-2001, 2001. 1.-3: pp. 132-141.
- [24] U, B.F.a.s., *An Overview of Tableau Algorithms for Description Logics*. Studia Logica, 2001. 69(1): pp. 5-40(36).
- [25] Sean Bechhofer, I.H., Carole Goble, Robert Stevens, *OilEd: a Reason-able Ontology Editor for the Semantic Web*. 2001. 2174 of LNAI: pp. 396-408.
- [26] Forgy, C., L., *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*. Artificial Intelligence, 1982.
- [27] Benjamin N. Grosz, I.H., Raphael Volz and Stefan Decker. *Description logic programs: Combining logic programs with description logic*. in *In Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*. 2003.
- [28] B. Motik, U.S., and R. Studer., *Query Answering for OWL-DL with Rules*. ISWC2004, 2004. 7-11.
- [29] Dmitry Tsarkov, A.R., Sean Bechhofer and Ian Horrocks, *Using Vampire to Reason with OWL*. ISWC2004, 2004.
- [30] Y. Guo, Z.P., and J. Heflin., *An Evaluation of Knowledge Base Systems for Large OWL Datasets*. Technical Report LU-CSE-, 2004. 04-012.
- [31] I.Horrocks, U.S., and S.Tobies, *Reasoning with individuals for the description logic SHIQ*.

Proceedings of the 17th International Conference on Automated Deduction (CADE-17), 2000. number 1831 in Lecture Notes in Computer Science.

- [32] Michael Hinchey, J.L.R., Walter F. Truszkowski, and Christopher A. Rouff, *Formal Approaches to Agent-Based Systems*. proceedings of the Third International Workshop (FAABS), 2004.
- [33] Jos de Bruijn, C.F., Uwe Keller, Ruben Lara, Axel Polleres, Livia Predoiu, and Holger Lausen, *WSML Deliverable D16.2 v0.2 WSML Reasoning Implementation*. Dec, 2004.
- [34] G. Karvounarakis, A.M., S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, K. Tolle, *Querying the Semantic Web with RQL*. Computer Networks and ISDN Systems Journal, 2003. 42(5): pp. 617-640.

APPENDIX A; THE DESCRIPTION LOGIC FAMILIES

The table A.1 contains information about Description Logic language **AL** family. The table A.2 contains information about DL language SH family.

Table A.1: Description Logic *AL* Families:

	Syntax	Description
<i>AL</i>	$C, D \rightarrow A$ \top \perp $\neg A$ $C \cap D$ $\forall R.C$ $\exists R.T$	atomic concept universal concept, top bottom concept atomic negation conjunction value restriction limited existential quantification
<i>C</i>	$\neg C$	concept negation
<i>U</i>	$C \cup D$	disjunction
<i>E</i>	$\exists R.C$	existential quantification
<i>N</i>	$\geq n R, \leq n R$	number restriction
<i>Q</i>	$\geq n R.C, \leq n R.C$	qualified number restriction
<i>R</i>	$R \cap S$	role conjunction
<i>I</i>	R-	inverse roles
<i>H</i>		role hierarchy
<i>F</i>	$u_1 = u_2, u_1 \neq u_2$	feature (dis)agreement

Table A.2: Description Logic of *SH* Families:

	Syntax
<i>S</i>	<i>ALC</i> + transitive roles
<i>H</i>	role hierarchy
<i>I</i>	inverse role
<i>Q</i>	qualified number restriction
<i>O</i>	concept enumeration
<i>(D)</i>	datatypes and values

APPENDIX B: USER GUIDE

The ROP system provides a visual user interface for ontology query. The GUI contains three functional modules: I/O control panel, Configuration panel, and Query panel.

Before submitting an ontology query, user must choose the subsystem as reasoner. From the *menu* shown in figure B.1, user can select between the ROPS and SWOPS subsystems.

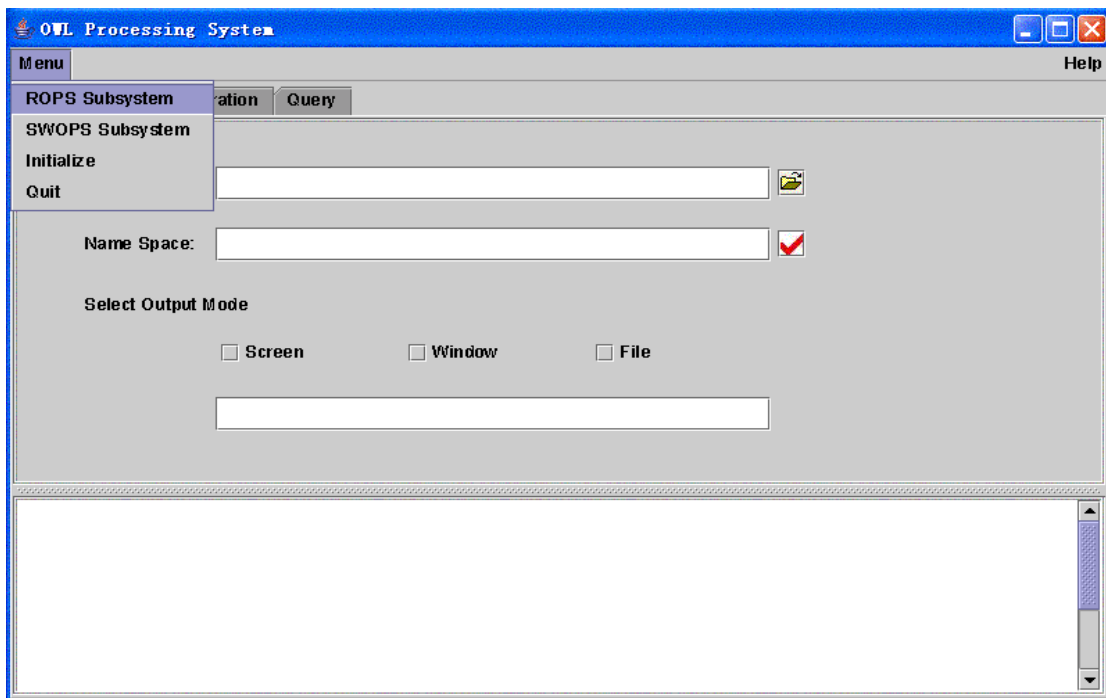


Figure B.1: Before starting

The I/O control panel appears the same for both subsystems. There is no difference between while using different subsystems. Figure B.2 displays how to use I/O control panel. From the *Input File* field, user can input the file name in which the ontology is stored. In the *Name Space* field, user can input the base name space. In the query process, users can then replace the long name space in the query with given abbreviation.

In the I/O control panel, user can also control the output mode. The output can be redirected to the combination of screen, result window of the GUI, and a local file. If the *File* check box is selected, a pop-up dialog window will notify user to choose an output file like what is shown in Figure B.2.

The ROPS subsystem can recognize input format in OWL, DAML, and RDF. The system will automatically recognize the input format. User can also select input format in the configuration panel. This selection will affect the performance and reasoning power of ROPS. The SWOPS subsystem can only recognize TPTP format.

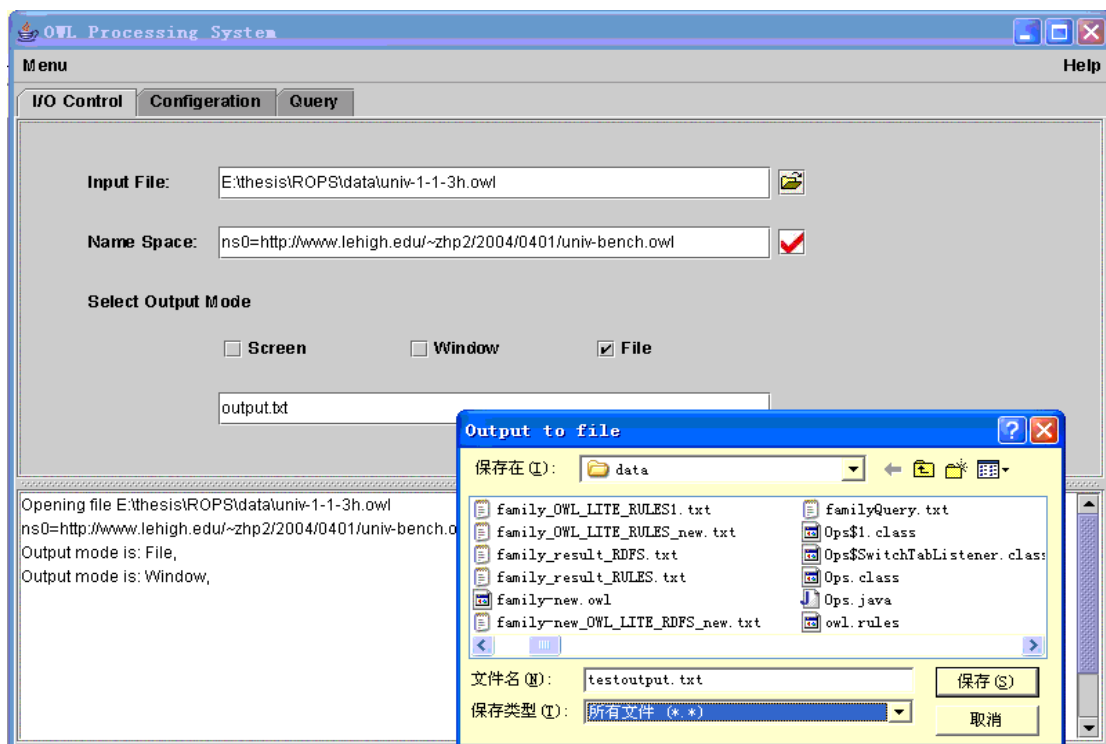


Figure B.2: I/O control panel

Figure B.3 displays the *configuration* panel. The combo lists on the left part control reasoners bound to ROPS subsystem. User can choose the *input type* and *model type* to

configure the system how much information to be recognized. The *memory mode* tells the system whether to execute in memory or to bind with an outside DBMS.

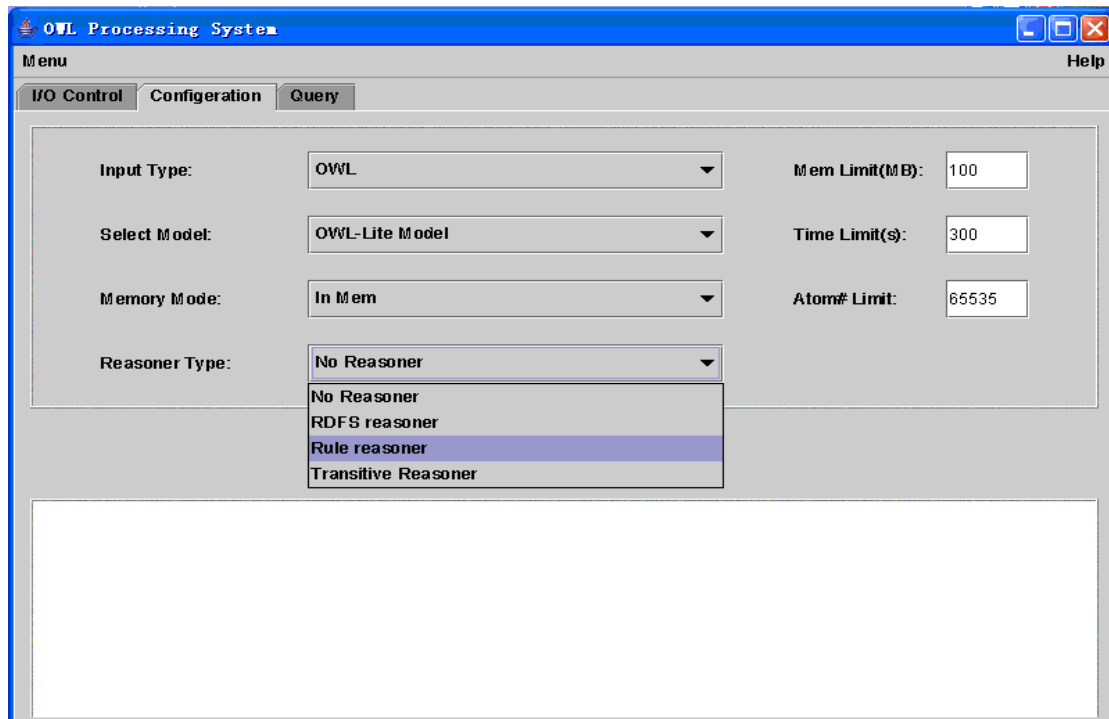


Figure B.3: Configuration panel

The last combo box controls the reasoner type or no reasoner. Different types of reasoner provide different level of reasoning power. The default reasoner is the *Rule reasoner*, the most powerful one.

On the right side is the configuration for the SWOPS subsystem. The memory limit control the maximum memory allocated to SWOPS. The time lime control the longest time SWOPS can execute before ending. This control avoids the infinite execution for some complex problems.

Figure B.4 illustrates the *Query panel* using ROPS as query system. After the loading of the ontology and configuration, users can submit queries in this panel. This panel also includes the rule extension mechanism. Users can add new rules to the ontology to get more information and make queries simpler.

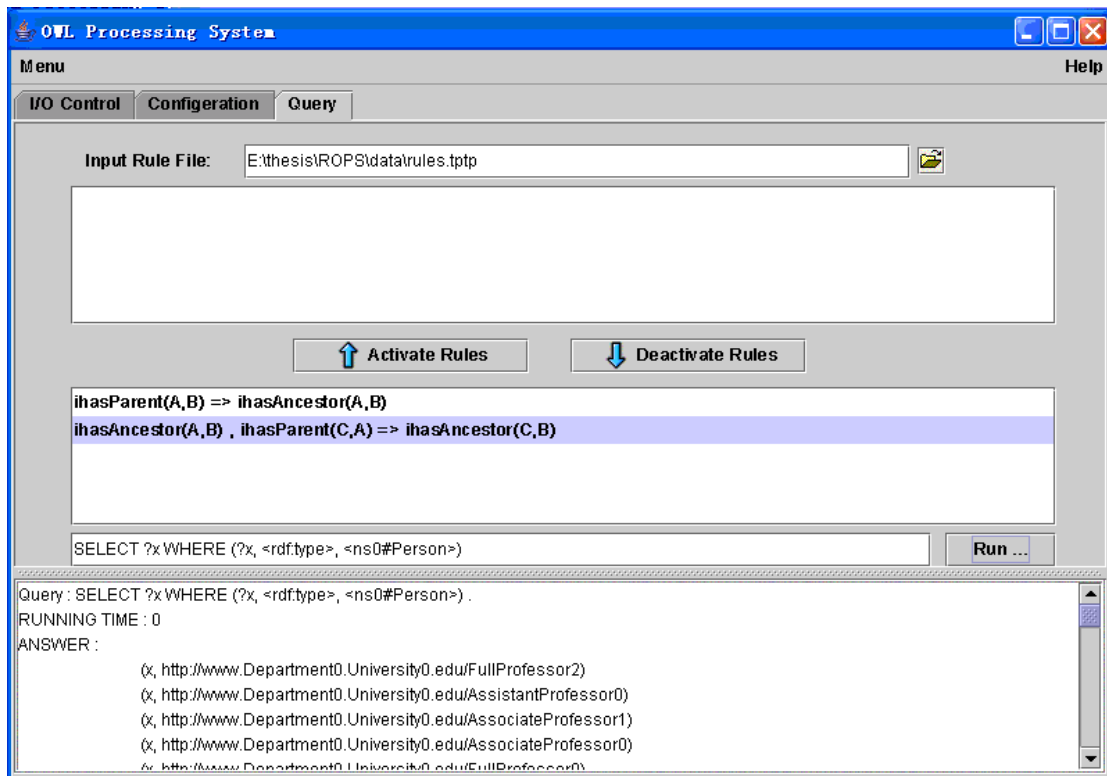


Figure B.4: Query panel of the ROPS subsystem

The rules are loaded from a local file. These rules will be passed to proper format and added to the ontology model in ROPS or SWOPS. Notice that the extended rules may significantly increase the complexity comparing to the original ontology. The computing time may be unbearable. The loaded rules are shown in the *inactive rules list* in the bottom combo lists in the panel. User can select which rule or rules to be activate or deactivate

between *active rules list* and *inactive rules list*. User can select multiple rules at the same time by press on the *ctrl* key during selection.

In the query input filed, user can type in query in proper format. For the ROPS system, the input query is in RDQL syntax. If the *Name space* was defined in the *I/O Control* panel, user can replace the long name space with the defined abbreviation to make the query more easily to be read and input. The output will be directed to the channel defined by user in the *I/OControl* panel. Notice that user can move the division bar to expand the result filed to show more results in the window.

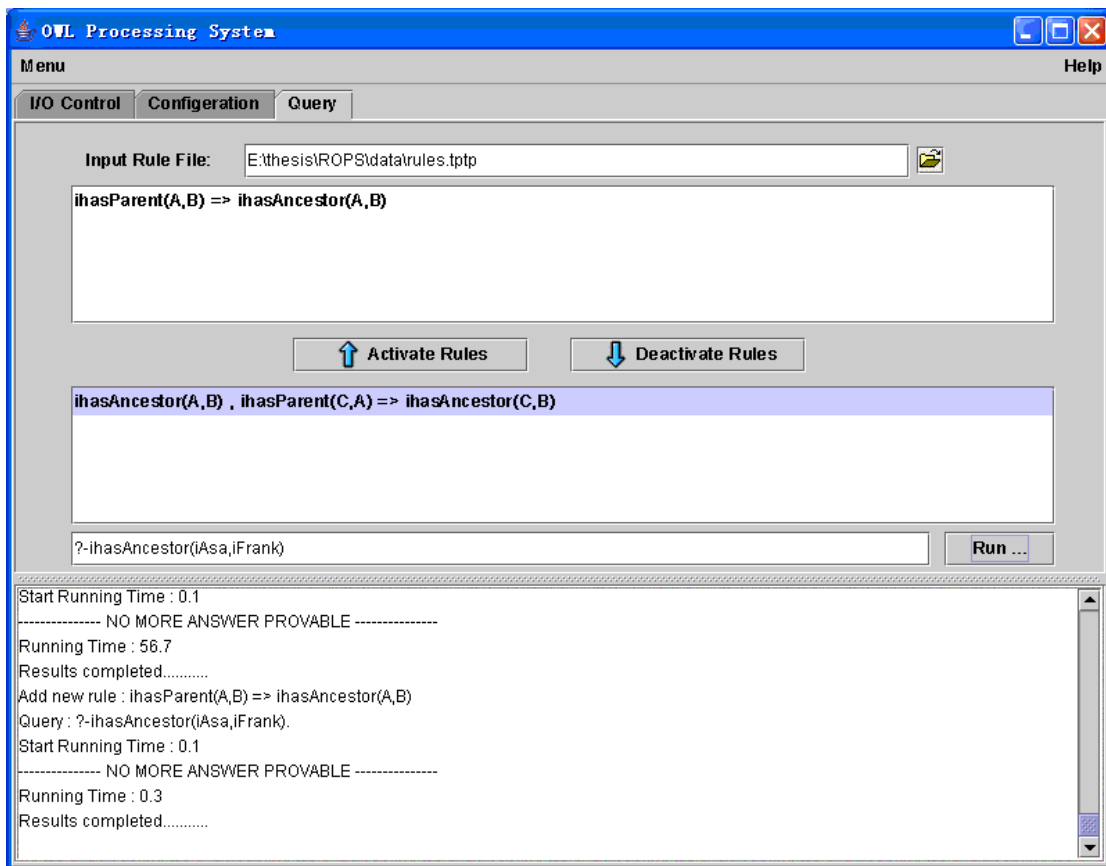


Figure B.5: Query panel of the SWOPS subsystem

The figure B.5 is the screen shot of SWOPS subsystem. The GUI is similar to that of ROPS. The difference is they use different query syntax. SWOPS uses a Datalog-like query and it will be translated into TPTP format for the underlying Vampire prover. As a result, the output format of SWOPS is different from that of ROPS too.

APPENDIX C: INSTALLATION GUIDE

The prototype of the OPS was developed in Java1.4.2. Because of the reasoner bounded, OPS can only work in Windows environment currently. It also requires Jena2.2 to be properly installed before running. Notice that the JENA_HOME environment variable must be properly defined or the Jar files for Jena2 should be copied to the %JAVA_HOME%\jre\lib\ext\ directory.

The OPS system does not need installation. Users can simply unzip the archive to local disk. There should be a directory *Ops*. Run the *RunOps.bat* will load the GUI of OPS. The usage of OPS is in the user guild.