Automatic Reasoner Composition and Selection

A thesis submitted to the
University of Dublin, Trinity College,
for the degree of
Doctor of Philosophy

Wei Tai

Knowledge and Data Engineering Group,
Department of Computer Science,
Trinity College, University of Dublin.

2011

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this or any other University, and that, unless otherwise stated, it is entirely my own work.

_____

Wei Tai

October 2011

Permission to Lend or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

_____

Wei Tai

October 2011

ACKNOWLEDGEMENTS

I would like to thank my supervisors Dr. Declan O'Sullivan and Dr. John Keeney for their unflagging support, insightful contributions and the many revisions of this thesis. I would also like to thank Dr. Rob Brennan for sharing his innovative ideas. My thanks to all in the KDEG research group for their lively discussions, their participation in my experiments, and also the proof reading/feedback.

I would like to especially thank my parents for their constant encouragement, support and the many sacrifices. A special word of thanks to my grandparents for their affection and encouragement.

Finally, thanks are due to Bei Gao for her absolute belief, unwavering support, and being wonderful reminders throughout this period that there is more to life than research!

ABSTRACTS

The development of OWL and OWL reasoning technologies has enabled them to be used for knowledge base (KB) modelling and/or intelligent data processing in applications of various areas. However the computation and memory intensive nature of OWL reasoners impedes the deployment of OWL ontology reasoning on resource-constrained devices. In order to address this issue, a possible approach is to compose the reasoners according to their application characteristics such that unnecessary reasoning capabilities are not loaded. This thesis introduces two novel automatic reasoner composition approaches, a selective rule loading algorithm and a two-phase RETE algorithm, that compose rule-entailment reasoners both at the rule level and inside the reasoning algorithm based on the ontology expressivity, in order to reduce resource consumption for reasoning on resource-constrained devices.

With the growth of usage of ontology reasoners and the introduction of new reasoner characteristics, it is envisaged that reasoner selection in the future will become too complicated for the current consultation based process between application developers and reasoner experts. In addition the thesis proposes a semi-automatic reasoner selection process (RESP) that allows users to independently select a most appropriate reasoner for their applications according to application characteristics. The solutions to the problems of how to achieve resource constrained reasoning and more automatic reasoning selection, have been respectively implemented, in a resource-constrained composable reasoner (COROR) and a semi-automatic reasoner selection tool (TARS).

Evaluation of the solutions indicates that the designed reasoner composition algorithms greatly reduce the time and memory requirement for reasoning and that proposed reasoner selection process helps users independently select a most appropriate reasoner for their applications, both of which will contribute to advancing the state of the art in the usage of reasoning within semantic applications.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE of TABLES

ABBREVIATIONS

ABox	Assertion Box
AC	Application Characteristic
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
COROR	COmposable Rule-entailment Owl Reasoner
CPU	Central Processing Unit
CWA	Closed World Assumption
DB	Database
DL	Description Logic
FOL	First Order Logic
IR	Intermediate Results
KB	Knowledge Base
L.H.S.	Left Hand Side
MIDP	Mobile Information Device Profile
NaF	Negation as Failure
OS	Operating System
OWA	Open World Assumption
OWL	Web Ontology Language
RAM	Random Access Memory
RC	Reasoner Characteristic
RDF	Resource Description Framework
RDFS	RDF Schema
RESP	REasoner Selection Process
R.H.S.	Right Hand Side
SPARQL	Simple Protocol and RDF Query Language
SUS	System Usability Scale
SWRL	Semantic Web Rule Language
TARS	Tool for Automatic Reasoner Selection
TBox	Terminology Box
XML	eXtensible Markup Language
XSLT	XML Stylesheet Transformation

Chapter 1
Introduction

1.1 *Motivation*

The Web Ontology Language (OWL) [McGuinness and Van Harmelen 2004] is an ontology language aiming to enable the content of information to be processed by machine rather than merely being displayed to humans. To enable this, OWL includes a set of formally defined constructs with logic-based semantics [Patel-Schneider et al 2004], which facilitates machine reasoning on an OWL ontology revealing implicit knowledge from knowledge that is explicitly stated. An example would be the inference from a subclass/superclass relationship that any individual of a subclass is also an individual of the superclass.

The formal definition and machine reasoning feature then enables OWL and its reasoning techniques to be used for knowledge base (KB) modelling and/or intelligent data processing in applications of various areas, such as clinical informatics [héja et al 2008, openGALEN, Golbeck et al 2003], bioinformatics [Harris et al 2004], battle field systems [Gomez et al 2008, Sensoy et al 2011], sensor network systems [Calder et al 2010, Kim et al 2008, Eid et al 2007], web services composition [Hatzi et al 2009] and so on, forming semantic applications. Such semantic applications usually have diverse (reasoning related) *application characteristics* (ACs) that impose different requirements on reasoning, leading to the need for different OWL reasoners with distinct *reasoner characteristics* (RCs) to be used. For example some sensor network systems require OWL reasoning to be performed on historical sensor readings stored in a database and therefore a database-enabled OWL reasoner is preferable, while for some semantic publish/subscribe systems the ability to quickly answer a query over a changing KB would be an important RC.

Currently the majority of research in the OWL reasoning area is targeted at building full-fledged, fast and powerful OWL reasoners that run on desktop computers. However with the

growth of development of semantic applications, for example for pervasive computing, there is an increasing demand for on-device OWL reasoning to facilitate intelligent on-device data processing, which in turn motivates the need for resource-constrained OWL reasoners [Kleemann and Sinner 2006, Brennan et al 2009, Koziuk et al, 2008]. Compared to the large amount of research performed to construct desktop OWL reasoners with different reasoner characteristics, very little research work has been conducted into how to minimize OWL reasoners so that they could operate on resource-constrained devices, in order to promote on-device intelligent data process and management [Ali and Kiefer 2009, Jang and Sohn 2004, Kim et al 2010].

Desktop OWL reasoners are often computationally and resource intensive therefore it is difficult to have them run on resource-constrained devices where resource limitations are imposed (e.g. the Sun SPOT sensor board has 180MHz 32-bit ARM920T core processor with 512K RAM and 4M Flash [SUN SPOT 2010]). A natural and reasonable thought would be to adjust the reasoning algorithms according to the application characteristics of a particular application such that only required reasoning capabilities are preserved and unneeded reasoning capabilities are not loaded, thereby hypothetically the amount of required processing and memory could be reduced. The term coined for such an approach in this thesis is *reasoner composition* whereby a customized reasoner is composed for a given application, ontology or platform.

Some reasoner composition mechanisms have been implemented in some OWL reasoners within the state of the art. For example Jena [Carroll et al 2004] and SwiftOWLIM [Kiryakov et al 2005] allow their reasoning rule set to be manually composed. However, such mechanisms are mostly static relying on tuning of the reasoner by reasoner experts or are only applicable to a specific rule set. The static reasoner composition mechanisms are appropriate if application characteristics are simple, relatively static and can be fixed before execution. However, they may be insufficient for some areas with a highly dynamic nature such as pervasive computing where applications often have changing application characteristics. An example would be the semantic publish/subscribe systems (refer to [Guo 2009] for a general discussion of the semantic publish/subscribe systems) that are used for scalable and efficient information delivery in highly dynamic environments. Those systems are often designed to be application and ontology independent in order to cope with the dynamic nature of wireless sensor networks and ideally they should tune itself at deployment depending on the client ontology in use [Keeney et al 2008, Pathan et al 2010]. Another example could be semantic context-aware systems where ambient information may

keep changing with time, e.g. users' interests, available services, the surrounding community and so on, which may need to alter ontologies and rules [Luther et al 2008, Ejigu et al 2007]. Hence it would be better that these systems are able to dynamically configure themselves to handle the diverse information from fast changing surroundings. Furthermore, as aforementioned, nowadays systems tend to push data processing toward the leaf nodes/edge of a network such as mobile phones or sensors, either to relieve servers from the heavy workloads and reduce throughputs of the networks from the system performance perspective or to protect users' privacy from being uploaded to servers from the human perspective [Kleemann and Sinner 2006, Brennan et al 2009, Koziuk et al, 2008]. Considering the large amount of resource-constrained devices involved as well as the often lack of expertise of the end users, this thesis argues that static reasoner composition mechanisms appear to be inappropriate for semantic applications in dynamic areas and automatic composition mechanisms show better suitability for such dynamic areas.

There are already many different reasoner characteristics out there for existing reasoners such as the support of rules, the support of database, and the support of conjunctive queries and so on, and the advance in OWL reasoning technologies will add more reasoner characteristics to the ever growing set of reasoner characteristics. For example the proposed reasoner composition mechanisms may enable reasoners to run on even smaller resource-constrained devices, adding a new reasoner characteristic. On the other hand, as mentioned earlier the different semantic applications emphasize different reasoner characteristics. Therefore this raises a problem that the selection of an appropriate reasoner is becoming ever more complicated and there is an increasing need to help people to be able to choose an appropriate reasoner for their applications.

In the state of the art, the selection of a reasoner relies largely on consultation between application developers and reasoner experts, and until recently this approach was straightforward and sufficient because of the relatively small number of OWL reasoners and reasoner characteristics envisaged. However, the ever widespread adoption of OWL and its reasoning technologies for applications in different domains and the rapid development and emergence of new OWL reasoning technologies makes this approach, in the opinion of the author, increasingly inadequate in the future for the following reasons. Firstly, as semantic applications grow more complicated and move beyond initial prototyping stages, these applications will be developed and extended by dedicated application developers with little or no knowledge of the intricacies of ontology reasoning. Furthermore reasoner experts may not always precisely understand some application characteristics expressed in domain

specific languages. The result could be a considerable amount of effort being required during consultations between an application developer and reasoner expert before an agreement is reached, or could even result in a wrong reasoner being selected. Secondly the existing approach requires that a reasoner expert is accessible to application developers, which will not always be the case. These inadequacies motivate the need for an automated approach to help application developers to limit consultation requirements or even to independently select a suitable reasoner for their semantic applications.

In summary, this thesis focuses on overcoming the problem of how to undertake reasoning in a dynamic and complex resource-constrained environment. Given the highly dynamic nature of pervasive computing, it is taken as an example of such an environment. Two solutions in particular are identified and focused upon: (1) the provision of an automatically composed reasoner based on application characteristics (e.g. the ontology to be reasoned), focusing on a resource-constrained environment, and (2) tool support for the semi-automatic selection of the most appropriate reasoner for a given semantic application.

1.2 *Research Question and Objectives*

This thesis addresses the question of:

> ***"How an appropriate resource-constrained OWL reasoner can be automatically composed and be selected based on application characteristics."***

Five objectives are derived:

- ***Objective 1:*** survey the state of the art OWL reasoners, identifying Reasoner Characteristics (RCs) and categorizing them. Identify an appropriate type of reasoner upon which the reasoner composition research should be based. Survey semantic applications, identifying reasoning-related Application Characteristics (ACs).

- ***Objective 2***: design automatic reasoner composition mechanisms and implement them in a resource-constrained reasoner.

- ***Objective 3***: study the performance impact on the resource-constrained reasoner brought by the application of composition algorithm(s).

- ***Objective 4***: design and implement a reasoner selection process that enables an application developer to automatically select a most appropriate reasoner for their semantic application based on application characteristics.

- ***Objective 5***: evaluate the usability of the reasoner selection process designed in objective 4.

Objective 1 establishes a foundation for the other research carried out in this thesis. The categorization of OWL reasoners enables the reasoner composition research (objective 2) to be carried out for a type of reasoner rather than a particular reasoner implementation, providing this research with a general grounding. Furthermore during the survey reasoner characteristics are identified, and they are examined for each survey reasoner. Similarly the survey on semantic applications can facilitate the identification of application characteristics, and furthermore it helps the study of how existing semantic applications select reasoners to fulfil their application characteristics. Both surveys provide a basis for objective 4 and 5 where the research on the automatic reasoner selection process is carried out.

Objective 2 and 3 together address the first half of the research question that examines *how an appropriate resource-constrained OWL reasoner can be automatically composed based on application characteristics*. Objective 2 designs the automatic reasoner composition mechanisms for the selected type of reasoners. The designed reasoner composition mechanisms need also to be integrated into a resource-constrained reasoner, implementing a composable resource-constrained reasoner. Objective 3 examines the performance impacts of the reasoner composition algorithms on the selected type of OWL reasoner. A natural approach is followed in the evaluation where the performance of the composable resource-constrained reasoner implementation with the use of composition mechanisms is compared with the reasoner without the composition mechanisms being used.

Objective 4 and 5 together address the second half of the research question, which is *how an appropriate OWL reasoner can be automatically selected based on application characteristics*. Design and implementation of an automatic reasoner selection process is performed in objective 4, and evaluation of such process is included in objective 5. The automatic reasoner selection process constructed in objective 4 will still require application developers to be involved at some stages of the automatic reasoner selection process, for example to perform some tasks such as identifying application characteristics for their applications. Thus the evaluation focuses on the usability of this reasoner selection process, achieved through user trials.

1.3 *Research Process and Approach*

An initial survey was conducted on 26 OWL reasoners in the state of the art and results are listed in Appendix A. For each reasoner 18 characteristics were surveyed, namely: *reasoning*

algorithm, *reasoner type*, *reasoner expressivity*, *completeness* (in terms of the supported expressivity), *reasoning tasks*, *materialization*, *incremental reasoning*, *query support*, *rule support*, *closed-world features*, *concrete domain*, *database support*, *remote interface*, *user access*, *explanation*, *ontology manipulation*, *platforms*, and *os*. Reasoners were then categorized into five types according to the algorithm used, including *DL-Tableaux reasoners* (DL stands for Description Logic), *rule-entailment reasoners*, *resolution-based reasoners*, *hybrid reasoners* and *miscellaneous reasoners* (as discussed in detail in section 2.3.1.1). For each type of reasoners, it's suitability for composition was discussed and the rule-entailment reasoner was found to have a good balance between expressivity and composability and thus was chosen as the target type of reasoner as the basis for the reasoner composition research. As reported in objective 1 another aim of this survey is the identification of reasoner characteristics. In fact as discussed in section 4.3 an example set of reasoner characteristics was identified based on this survey.

Then a second survey was performed over five diverse sample types of semantic applications, namely *semantic publish/subscribe systems*, *semantic context-aware systems*, *medical and bioinformatics systems*, *semantic sensor network management systems*, and *software engineering systems*, chosen due to their wide adoption of semantic reasoning technologies to solve previously encountered problems or to perform intelligent data processing/accessing. This survey mainly concentrated on the examination of the application characteristics of these applications and how these application characteristics could affect the selection of an appropriate reasoner. As a matter of fact the examination of them facilitated the identification of an example set of application characteristics and correspondingly the mapping between application characteristics and reasoner characteristics which is termed *connections* in this thesis.

The RETE algorithm [Forgy 1982] is the most common reasoning algorithm for rule-entailment reasoners and hence two reasoner composition algorithms were designed by the author to compose at different levels, i.e. the rule set level and inside the RETE algorithm. The implementation of both composition algorithms was performed on a modified version of μJena [Micro Jena 2010] extended with a J2ME-ported Jena RETE engine [Carroll et al 2004]. The choice of Jena RETE engine was motivated by the fact that it is a typical rule-entailment reasoner and it is open source. The resulting implementation is named a COmposable Rule-entailment Owl Reasoner (COROR).

In order to investigate the performance impacts brought by the application of the reasoner

composition algorithms, two experiments were performed. A first experiment was performed to measure and compare the time/memory performance of different COROR composition configurations (with one, two, or no composition algorithms) to fully reason over ontologies. This can directly reflect the performance change caused by the application of composition algorithms. A second experiment was then carried out to compare the performance of COROR with four counterpart rule-entailment reasoners namely Bossam [Jang and Sohn 2004], BaseVISor [Matheus et al 2006], SwiftOWLIM [Kiryakov et al 2005] and Jena RETE engine. This experiment could show how COROR performs compared to the other rule-entailment reasoner implementations. In addition it could reveal the performance merits and pitfalls of COROR compared to other state of the art rule-entailment reasoners

A REasoner Selection Process (RESP) was then designed which aims to assist application developers with little background on ontology reasoning to perform semi-automatic reasoner selection based only on application characteristics. A simple matchmaking approach is used in the process to check if the characteristics of an application are all satisfied by reasoner characteristics of a candidate reasoner. This process was designed to be a methodology without specifying any detailed technical solutions in order that it can be reused for different application domains. To make this process less abstract for demonstration and evaluation, a set of example application characteristics were identified based on the examination of 11 selected reasoning-related aspects of the surveyed semantic applications, ranging from frequently changing knowledge base to explanations of reasoning. Connections between the example application characteristics and the example reasoner characteristics were drawn based on the discussion in the survey. RESP was implemented as a prototype desktop application using Java, termed Tool for Automatic Reasoner Selection (TARS). The example application characteristics, example reasoner characteristics, and connections were implemented in TARS.

Since human-beings are still involved in TARS to identify application characteristics for their semantic applications, a usability experiment was designed and performed that allowed participants to experience different facets of TARS, including a reasoner selection task that asked semantic application developers/users to select an appropriate reasoner for a given semantic application following RESP, and a reasoner registration task that asked reasoning-aware users to register a candidate reasoner with TARS.

1.4 *Contributions*

Two contributions are identified.

Major contribution: **the design of two novel automatic reasoner composition algorithms for rule-entailment reasoners, termed the *selective rule loading algorithm* and the *two-phase RETE algorithm*, and the implementation of them as a resource-constrained rule-entailment OWL reasoner named COROR (*COmposable Rule-entailment Owl Reasoner*)**

Minor contribution: the design and implementation of an automatic *REasoner Selection Process* (RESP) and a prototypical implementation, termed the *Tool for Automatic Reasoner Selection* (TARS).

The major contribution that distinguishes this research from the state of the art is the design of the two novel automatic reasoner composition algorithms for rule-based reasoners, i.e. the *selective rule loading* algorithm and the *two-phase RETE* algorithm, and the implementation of them into a resource-constrained rule-based OWL reasoner named COROR (COmposable Rule-entailment Owl Reasoner). As will be introduced in section 2.3.1.1.2 this research focuses on rule-based reasoners, more specifically on rule-entailment reasoners, because they provide good balance between semantic expressivity and composability. The reasoner composition algorithms respectively perform the reasoner composition at the rule set level (*selective rule loading* algorithm) and inside the RETE algorithm (*two-phase RETE* algorithm). Briefly the *selective rule loading* algorithm estimates the usage of OWL reasoning rules for reasoning a particular ontology, and then selectively loads into the engine the rules estimated as useful. Unneeded rules are omitted, avoiding the allocation of resources originally required. The *two-phase RETE* algorithm interrupts the RETE network construction [Forgy 1982] (which is the internal reasoning algorithm for rule-entailment reasoners) midway by matching ontology against already constructed network, such that some heuristic indicators about the ontology that was only known at runtime can be collected and used to optimize[1] the remaining RETE network construction. This enables the reasoning to be optimized taking into account information of the particular ontology.

Experiments were performed to investigate the performance impact by applying the algorithms. Results show that for the tested ontologies the application of the *selective rule*

[1] In this thesis optimize is referred to as having a better solution rather than having the best solution.

loading algorithm on average can reduce the time consumption by 33% and memory consumption by 35% for fully computing entailments. The application of *two-phase RETE* algorithm on average can reduce the time consumption by 78% and memory consumption by 74%. This contribution enables rule-entailment reasoners to be automatically composed in accordance with the ontology to be reasoned.

Hence reasoners can always remain tightly "fit" to the changing ontology, enabling customized OWL reasoning for applications with high dynamism. Good examples of such applications are context-aware computing on resource-constrained devices or semantic publish/subscribe systems as described above. As shown by the experiment results, such customization of reasoning algorithms can save a large amount of resources for reasoning the same ontology, which, in other words, means that more complex/bigger ontology than before can be reasoned on the same resource-constrained devices, increasing the amount of intelligent processing that can be pushed to the edge of a network. In addition, although targeted at resource-constrained environment, it is envisaged by the author that the capability to reduce the memory consumption of OWL rule-entailment reasoning also enables the two introduced composition algorithms to benefit (desktop-based) applications, for example, to reduce the memory consumption of applications requiring web-scale reasoning, e.g. social network applications, so as to increase reasoning scalability.

A minor contribution of this research is the design and implementation of an automatic REasoner Selection Process (RESP) and a prototypical implementation, termed Tool for Automatic Reasoner Selection (TARS). RESP is a process that automatically assists semantic application developers to select an appropriate reasoner for their semantic applications. It uses a relatively simple but useful matchmaking approach. In RESP semantic applications are represented as sets of application characteristics and reasoners are represented as sets of reasoner characteristics. Users identify and select the set of application characteristics representing their applications from the given candidate application characteristics and input them into RESP. The selection process is then one of matchmaking between the input application characteristics and reasoner characteristics of candidate reasoners through a set of predefined connections. Reasoners are ranked by the degree by which their reasoner characteristics match the selected application characteristics. The one(s) with 100% satisfaction is then the appropriate one(s). In the implementation users can also view which application characteristics are not satisfied with a reason for the mismatch. Based on this information users can adjust their required application characteristics if appropriate.

Usability experiments show RESP helps application developers in determining the reasoner to be used with little or even no help from reasoner experts (in contrast to the existing consultation-based reasoner selection process in which application developers totally rely on reasoner experts to select an appropriate reasoner).

This contribution enables the reasoner selection process to be moved from a human-to-human consultation-based approach to a human-to-computer semi-automatic approach. Application developers can easily select an appropriate reasoner for their applications without putting a lot of efforts in looking for reasoner experts or in the uptake of the intricacies of OWL reasoning technologies, facilitating the widespread of OWL and OWL reasoning technologies. Furthermore as a by-product when researching the semi-automatic reasoner selection approach, the identified application characteristics and reasoner characteristics will be useful as a starting point for other research in this area.

Four papers were published in relation to research carried out in this thesis:

"*An Automatically Composable OWL Reasoner for Resource Constrained Devices*", in Proceeding of the International Conference on Semantic Computing (ICSC'09), 2009.

This paper describes the *selective rule loading* algorithm and a prototype implementation of it on a desktop reasoner.

"*Open Framework Middleware for intelligent WSN topology adaption in smart buildings*", *Proceedings* of the International Conference on Ultra Modern Telecommunications (ICUMT'09), 2009.

This paper investigates a potential use of composable reasoner to perform localized fault correlation in clustered wireless sensor networks.

"*A COmposable Rule-Entailment Owl Reasoner for Resource-Constrained Devices*", *Proceedings* of the International Symposium on Rule-Based Reasoning, Programming, and Applications (RuleML'11), 2011.

This paper concentrates on the design, implementation and evaluation of COROR. The design of *selective rule loading* algorithm and the *two-phase RETE* algorithm is given in detail. The implementation of them in COROR as well as the evaluation are also described and discussed in this paper.

"*RESP: A Computer Aided OWL REasoner Selection Process*", *Proceedings* of the International Conference on Semantic Computing (ICSC'11), 2011.

This paper describes in detail the RESP process. Application characteristics are discussed from 11 reasoning-related aspects. A prototype implementation (TARS) and a usability evaluation are also presented and discussed.

It is planned to submit a more detailed description of COROR and RESP based on this thesis to the Journal of Web Semantics or IEEE transaction on knowledge and data engineering.

1.5 *Thesis Overview*

This thesis is comprised of seven chapters.

Chapter Two provides background knowledge and related work of this thesis. Background knowledge includes the OWL ontology language and its sublanguages, and the RETE algorithm and its optimizations. Related work includes a survey of OWL reasoners, a survey of semantic applications, a discussion of state of the art composition approaches for rule-based reasoners, and finally a discussion of resource-constrained reasoners.

Chapter Three presents the design of COROR, a composable rule-entailment reasoner. Two novel composition algorithms are described in detail.

Chapter Four describes the design of RESP, a process for selecting an appropriate reasoner for a given set of ACs. Some examples are also discussed.

Chapter Five describes a prototype implementation of COROR and also a prototype implementation of RESP (named TARS) with some code snippets and screen shots attached.

Chapter Six presents the design, settings and results of experiments performed for this thesis. Two experiments, i.e. an intra-reasoner comparison and an inter-reasoner comparison, were performed to evaluate how reasoning performance changes with the application of reasoner composition algorithms. This chapter also presents a usability experiment for RESP and TARS, and provides results and analysis.

Chapter Seven concludes this thesis. Contributions and future work of this work are discussed.

Seven appendices are attached at the end of this thesis. They are the results of the survey of

OWL reasoners, the scenario descriptions used in the usability experiment for TARS, the questionnaires used in the usability experiment for TARS, the pD* entailment rules and their implementations as Jena rules, the rule-construct mappings as a text implementation of rule-construct dependency graphs, the profiles of candidate reasoners registered with TARS, and finally a full list of the Java classes added to μJena in order to make it support OWL reasoning.

A DVD disc with all raw experiment results, ontologies used in the experiment and collected questionnaires is also submitted with this thesis.

Chapter 2
Background and Related Work

2.1 *Introduction*

This chapter presents the background as well as related work for the investigation of the research question of this thesis as to

"How an appropriate resource-constrained OWL reasoner can be automatically composed and be selected based on application characteristics."

As the overall context of this thesis, OWL and its sublanguages are discussed in the beginning in section 2.2.1. Both standard and non-standard OWL sublanguages are discussed. As indicated by the research objective 1, to kick off this research and to establish a solid foundation for the subsequent research, two surveys needed to be conducted: a survey of state of the art OWL reasoners and a survey of semantic applications. For the survey of OWL reasoners, two major problems needed to be solved. Firstly, in order to enable the research composition research to be applicable for a type of reasoners rather than a specific reasoner implementation, OWL reasoners needed to be categorized in some sense. A natural approach was to obtain this categorization based on the reasoning algorithms, as to compose OWL reasoners at the reasoning algorithms level would be the most intuitive way, given that the motivation to reduce resource consumption. Later investigation indicated that rule-entailment reasoner algorithm would be most amenable for composition in a way that would achieve resource efficient reasoning. A second problem that needed to be addressed was the distillation and survey of reasoner characteristics, which would also facilitate the research as to automatically select OWL reasoners. The survey of OWL reasoners is presented in 2.3.1 with the categorization of OWL reasoners and the identification of reasoner characteristics separately presented in section 2.3.1.1 and section 2.3.1.2.

The survey of semantic applications was to help the author to learn the interplays between the application characteristics and OWL reasoner characteristics, and also help the identification of (reasoner-related) application characteristics, that would facilitate the design of an automatic reasoner selection process. This survey and the survey of reasoners mentioned earlier influenced each other to some extent: the distillation of some specific reasoner characteristics, e.g. closed-world features, was inspired by the observation that the satisfaction of some application characteristics, e.g. integrity constraints, greatly depends on the possession of such reasoner characteristics and vice versa. The survey of semantic applications is presented in section 2.3.2.

Then in order to identify a type of reasoner upon which to carry out the automatic reasoner composition research, the composability for different OWL reasoner categories is discussed in section 2.3.3. Rule-based reasoners were found to have better potential in composition and hence show better suitability for the automatic reasoner composition research. Existing reasoner composition approaches for rule-entailment reasoners were then studied (although none of these approaches are called “reasoner composition approach”). These reasoner composition approaches can fall into three types according to different composition levels they work on: to manually set the reasoning capability using pre-defined OWL sublanguages, to manually add/remove inference rules loaded into the reasoner, and automatic reasoning capability composition. Further investigation of these approaches revealed a gap where the automatic reasoner composition research could fit in: the existing reasoner composition approaches are either static relying on human tuning or designed to work on a specific semantics, therefore emphasizing the need for the design of an automatic reasoner composition approaches at the reasoning algorithm level and independent of specific semantics.

As the designed reasoner composition approaches are implemented upon resource-constrained devices, existing resource-constrained reasoners are presented in section 2.3.4. Surprisingly, it is rare that resource-constrained reasoners use reasoner composition approaches. In fact most of them are a direct migration of the desktop OWL reasoning algorithm onto resource-constrained devices. One resource-constrained reasoner adopts an automatic reasoner composition approach however it is restricted to only a specific OWL sublanguage. In addition the investigation of existing resource-constrained reasoners also reveals another gap for the reasoner composition research (at the implementation perspective) that none of the existing resource-constrained reasoners is designed for small devices with very limited resources, e.g. wireless sensors.

As will be seen in Chapter 3, rule-entailment reasoners are selected as the basis to carry out the reasoner composition research. Since RETE [Forgy 1982] is the major reasoning algorithm for this type of reasoner, a detailed description of RETE is given in the background section.

2.2 *Background*

In this section background knowledge of this thesis is presented. It includes two major parts: the OWL and its sublanguages and the RETE algorithm and its optimizations.

2.2.1 OWL and OWL Sublanguages

OWL is an ontology modelling language standardized and recommended by W3C. It consists of a set of formally defined OWL constructs each of which is given a logic-based semantic [Patel-Schneider et al 2004]. OWL has three standard sublanguages, i.e. OWL-Full, OWL-DL and OWL-Lite, varying in the set of constructs supported and the semantic expressivity. Non-standard OWL sublanguages are also designed for different usages according to the OWL features supported, such as the pD* family [ter Horst 2005a, ter Horst 2005b] and so on. Standard and non-standard OWL sublanguages are respectively discussed in section 2.2.1.1 and 2.2.1.2. OWL 2 [OWL 2 Overview] is the latest update of OWL. It extends OWL with more semantic expressivities and three application-oriented sublanguages, i.e. sublanguages that have been carefully crafted to suit for different usages. OWL 2 is introduced in section 2.2.1.3.

2.2.1.1 OWL-Full, -DL and -Lite

OWL original has three standard sublanguages: OWL-Full, OWL-DL and OWL-Lite. OWL-Full provides complete support for all OWL constructs and it is fully compatible with RDF. It is very expressive for modelling domain knowledge however its reasoning tasks are not decidable, and fully automated reasoning is not possible. OWL-DL (OWL Description Logic) supports the same set of OWL constructs as OWL-Full but restricts support for some of the semantics possible in RDF in order to have decidable reasoning tasks. OWL-DL is currently the most widely used of the OWL sublanguages. OWL-Lite has a reduced construct set in order to provide the minimal useful language features for tool builders to support. In the rest of this thesis when OWL is mentioned without specifying the sublanguage it should be taken to mean OWL-DL.

2.2.1.2 Nonstandard OWL Sublanguages

Some nonstandard OWL sublanguages are designed to have computational or modelling advantages for some dedicated reasoning tasks. In this section some nonstandard OWL

sublanguages are discussed including the pD* semantics [ter Horst 2005a, ter Horst 2005b], EL++ [Baader et al 2005, Baader et al 2008], DL-Lite [Calvanese et al 2007], and DLP [Grosof et al 2003].

The *pD** semantics (including pD* and pD*sv) defines a weakened but tractable[2] subset of the OWL semantics. It adds to the D* semantics (the complete RDFS semantics extended with various datatypes, refer to [ter Horst 2005a]) with P entailment rules (a set of entailment rules implementing the partial semantics for some OWL constructs, refer to [ter Horst 2005a]). Some OWL constructs are missing, such as cardinality constructs (*cardinality*, *minCardinality* and *maxCardinality*), some (in)equality constructs (*allDifferent* and *distinctMembers*), Boolean combination constructs (*unionOf*, *complementOf* and *intersectionOf*), and *oneOf*. Still a substantial subset of OWL-DL constructs is kept (as indicated in Table 2-1). Semantics are encoded using a set of RDFS-like *if semantic* conditions and therefore the pD* semantics have PTIME[3] entailment complexity when variables are not used in the target ontology, i.e. checking if a variable-free target ontology G′ is the logical consequence of an ontology G for the given semantics, and NPTIME[4] entailment complexity when variables are used in the target ontology. A later work in [ter Horst 2005b] extends the pD* semantics with the support of the *iff semantics* for the *owl:someValuesFrom* forming the *pD*sv* entailment. However it does not have PTIME complexity. The pD* semantics inspired the standardization of OWL 2 RL [OWL 2 Profiles], an OWL 2 sublanguage with RDFS-like *if semantics*. A full set of pD* entailment rules can be found in Appendix C.

Table 2-1: OWL constructs supported by the pD* semantics

owl:FunctionalProperty	owl:Restriction
owl:InverseFunctionalProperty	owl:onProperty
owl:SymmetricProperty	owl:hasValue
owl:TransitiveProperty	owl:someValuesFrom
owl:sameAs	owl:allValuesFrom
owl:inverseOf	owl:differentFrom
owl:equivalentClass	owl:disjointWith
owl:equivalentProperty	

[2] Tractable is a concept in computational complexity theory. A problem is tractable if it can be solved in polynomial time.
[3] PTIME (Polynomial TIME): A PTIME problem can be solved by a deterministic Turing machine using a polynomial amount of computation time.
[4] NPTIME (Nonuniform Polynomial TIME): A NPTIME problem can be solved in polynomial time on a non-deterministic Turing machine.

The EL++ is a specially dimensioned description logic to have tractable subsumption problems for ontology with a very large number of properties and classes. A very large subset of it (without property chain) can correspond to a subset of OWL. Some expressivity of OWL needs to be removed for EL++ to ensure its tractability, such as atomic negation, disjunction, at least restriction, inverse property, cardinality, and functional/inverse functional property. However EL++ is still expressive enough for modelling bioinformatics or medical ontologies such as Gene [Harris et al 2004] and SNOMED [héja et al 2008]. Part of the Galen ontology [openGALEN] can be expressed in EL++ as well. The EL family description logic has been standardized as an OWL 2 sublanguage, i.e. OWL 2 EL [OWL 2 Profiles].

The DL-Lite is a family of description logics carefully tailored to provide efficient conjunctive query answering capabilities over knowledge bases with large datasets. DL-$\text{Lite}_{\text{core}}$ is the basic logic of the DL-Lite family. It can express someValuesFrom (unqualified), concept conjunction, concept disjointness, domains and ranges of properties, inverse properties. There are several variants for DL-$\text{Lite}_{\text{core}}$ extending it with different features without complicating the reasoning problems. For example DL-Lite_{R} extends DL-$\text{Lite}_{\text{core}}$ with inclusion assertions between object properties and DL-Lite_{F} extends with functional and inverse functional property. DL-Lite_{R} is standardized in OWL 2 as the logic underpinning of OWL 2 QL [OWL 2 Profiles].

DLP (Description Logic Programmes) is a significant subset of OWL DL and Horn logic. It aims to provide bi-directional translation between semantic web ontologies and horn rules, coined DLP-fusion (for both ontology and reasoning problems), which allows an ontology built on top of rules and conversely rules built on top of an ontology. DLP supports the following features of OWL DL, namely concept disjointness, domains and range of properties, inverse and symmetric properties, functional and inverse-functional properties, inclusion and equivalence of object properties, transitive properties and a limited form of General Concept Inclusion (GCIs, i.e. $C \sqsubseteq D$ where both C and D are complex concept description rather than concept names). The translation between OWL and rules adopted by DLP is directly based on the semantic connections between horn logic and DL. Therefore rather than translating the ontology into a set of facts and using a set of OWL entailment rules to perform OWL reasoning, DLP translates the ontology into a horn rule program.

2.2.1.3 OWL 2 and OWL 2 Sublanguages

OWL 2 is the latest update of OWL. Although the names of some OWL 2 constructs are

different from those in OWL, it is backward compatible with OWL 1. All OWL DL and OWL Lite ontologies are also considered as OWL 2 ontologies [OWL 2 Overview]. OWL 2 adds some new features into OWL, including some new syntactical features, such as disjoint union of classes, and some new expressivity enhancements, such as: keys; property chains; richer datatypes/data ranges; qualified cardinality restriction; asymmetric, reflective and disjoint properties; and some enhanced annotation properties. These new expressivities together with OWL DL amount to an equivalent expressivity as the description logic $\mathcal{SROIQ}(\mathcal{D})$ (refer to [Baader et al 2007] for the naming scheme for a description logic). As in OWL 1, there are two types of semantics for OWL 2, a direct model-theoretic semantics [OWL 2 Direct Semantics] and a RDF-based semantics [OWL 2 RDF-Based Semantics]. A correspondence theorem is defined between them.

OWL 2 defines five sublanguages. OWL 2 Full and OWL 2 DL are used to (informally) represent the entire OWL 2 capabilities when respectively interpreted using RDF-based semantics and direct semantics. OWL 2 DL can be viewed as OWL 2 Full with restrictions on some RDF features, such as the separation of properties, individuals and classes. In addition OWL 2 also defines three other more frequently used sublanguages, i.e. OWL 2 EL, OWL 2 QL and OWL 2 RL, in order to obtain tractable reasoning problems for some specific applications while retaining sufficient expressivities. OWL 2 EL extends EL++ and is designed to provide tractable reasoning services for very large ontologies with classes. OWL 2 QL enables efficient conjunctive query answering. OWL 2 RL defines a set of entailment rules (based on the pD* semantics), enabling OWL ontologies to be reasoned using rule systems.

Note that in the rest of this thesis that if not specifically mentioned, the term OWL is used only for OWL DL. The terms OWL 2 will be specifically mentioned when OWL 2 is to be discussed.

2.2.2 RETE and RETE Optimizations

As will be discussed in Chapter 3, the rule-entailment reasoners (using the ontology independent translation approach) show better suitability than the other reasoner categories for reasoner composition research. As the typical reasoning algorithm for rule-entailment reasoners, the RETE algorithm [Forgy 1982] is described and discussed in detail. Some optimizations have been designed to reduce its resource consumption, and they are described and discussed as well.

2.2.2.1 An Overview of RETE

The RETE algorithm is an efficient forward chaining pattern matching algorithm that matches facts in working memory against rules using a discrimination network termed *RETE network* (Figure 2-1). RETE forms the basis for most modern production rule engines and is also the internal reasoning algorithm for rule-entailment reasoners. RETE incorporates two of the three types of knowledge that are considered may be included in a production system algorithm [McDermott et al 1978], which are

memory support, i.e. a scheme indicating the subset of working memory elements matching each condition element, and

condition relationship, i.e. interaction between condition elements within a rule.

The above two types of knowledge correspond to the *alpha network* and *beta network* of a typical RETE network (as shown Figure 2-1). The alpha network is comprised of one-input nodes named *alpha nodes* (maybe chained for some implementations), each of which performs intra-element test for each individual *condition element* in the left hand side (l.h.s.) of each rule. This intra-element test operation is termed *match*. Successfully matched facts are stored in the *alpha memory* of the corresponding alpha node as an *intermediate result* (IR), forming the *memory support*.

As given in Figure 2-1, the beta network is comprised of networked two-input *beta nodes* that check for the consistency of variable bindings for pairs of intermediate results. The consistency checking operation is termed a *join*. Each input has associated memory where inputs from alpha memory or previous beta memory joins are stored, called *beta memory*. A typical beta node receives an intermediate result from one input each time (and stores it in the connected beta memory) and joins it against intermediate results stored in the opposite input. A joined intermediate result is generated for the paired tokens with consistent variable bindings and it is propagated to its successors which could be another beta node performing similar inter-condition checks or the *conflict set* (where intermediate results successfully matched to the entire rule reside) waiting for firing. The intermediate results joined by the last beta node are termed as an *instantiation* of the rule. The beta network realizes the *condition relationship*.

Note that the above mainly describes how RETE handles additions (non-negative fact). Deletion (negative fact) is handled in the same way as addition. However rather than storing the deleted intermediate result in (alpha/beta) memory, RETE removes any same

intermediate results stored in the memory, and then propagates to the next node.

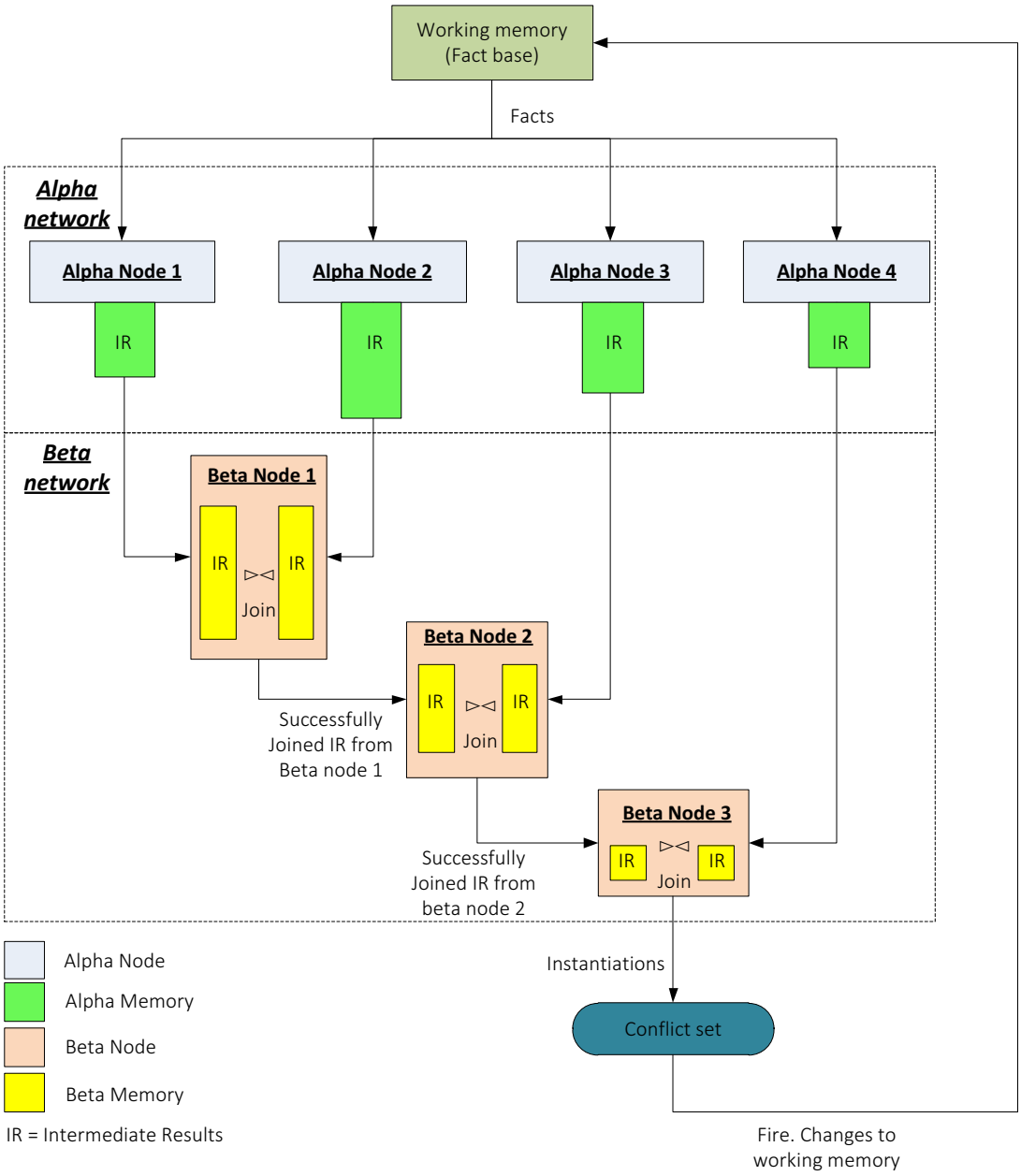


Figure 2-1: An example RETE network

The RETE algorithm operates iteratively with each iteration (termed a *RETE cycle* in this thesis) containing facts matching, joining and firing. The firing of rules may change the working memory by adding/removing any inferred facts, and changes are reflected in the RETE network by inserting the newly inferred facts into the network, matching and joining as normal facts. RETE blocks when no more changes are made to the working memory, and the result in working memory then contains all asserted and inferred facts.

RETE does not solve two problems that are commonly encountered in practical implementations. They are *conflict resolution*, i.e. how multiple instantiations can be ordered in the conflict set for firing, and *rule firing*, i.e. how firing a rule can change the working memory (e.g. adding/removing facts) and then affect the execution of the RETE algorithm (e.g. removing a fact can cause the retraction of an instantiation in conflict set). Many solutions have been proposed but since they are out the scope of this thesis they are not discussed in detail here.

2.2.2.2 Other Pattern Matching Algorithms

There are some other pattern-matching algorithms besides RETE. The two most famous ones are TREAT [Miranker 1987] and LEAPS [Miranker et al 1990].

TREAT is designed following the conjecture made in [McDermott et al 1978] that sometimes maintaining intermediate results may require more efforts than re-testing and hence it becomes not worthwhile. It incorporates three types of knowledge: *condition membership*, *memory support* and *conflict set support*, where the former two are already defined in [McDermott et al 1978] as two of the three knowledge types should be included in a production system algorithm, and the *conflict set support* is proposed by TREAT as a forth type of the knowledge. It decides whether to generate or to invalidate instantiations for addition and retraction of facts. TREAT uses alpha memory only and excludes beta memory, realizing *memory support* but excluding the *condition relationship* knowledge of a RETE–based production system.

An initial empirical test by counting joins required in reasoning show TREAT outperforms RETE in all cases [Miranker 1987]. However a later work in [Nayak et al 1988] points out that although TREAT performs faster deletion operation for positive nodes, RETE outperforms it in most cases as TREAT is more likely to suffer from a *long-chain effect*, i.e. where the absence of successful matching is only detected after a large amount of expensive matching operations have been performed, leading to a waste of computational resources. Another performance comparison conducted in [Wang and Hanson 1992] shows TREAT always outperforms RETE in terms of the number of joins and storage required for evaluating database rule conditions. However later work in [Ding et al 2009] replaces the RETE engine in CLIPS with a TREAT engine and the experiment shows TREAT scales better than RETE. As recently pointed out by Miranker on his personal website[5], RETE and TREAT share the same algorithmic complexity and the algorithmic superior of TREAT can

[5] http://www.cs.utexas.edu/~miranker/treator.htm

easily be overcome by implementation. Furthermore since a *single pass* method is outlined for producing code for RETE, a RETE implementation is easier than a TREAT implementation.

LEAPS (Lazy Evaluation Algorithm for Production Systems) is a variant of TREAT with best-first search for instantiations [Miranker et al 1990]. It is developed to avoid time and space wastage in the eager evaluation of rules [Miranker et al 1990]. Rather than enumerating the conflict set searching for the first to fire, LEAPS executes a best-first search for instantiations on a total ordered working memory (by recentness). Search states are stored in a stack. For each addition (deletion) that is matched to a non-negative fact (negative fact), an initial search state is generated and pushed into a stack. Once an instantiation is found the best-first search pauses immediately. The current search state is pushed into stack and the instantiation is fired. A next search state is popped as the root of the following search if the current search is exhausted. The algorithm halts if no more search states can be popped. A preliminary evaluation in [Miranker et al 1990] shows it substantially improves execution time as well as the space requirement for some application programs. Descriptions of the LEAPS implementation in OPS5 can be found in [Miranker et al 1990, Batory 1994]. LEAPS was supported in the Drools 3.x rule engine as well but not in later versions due to poor maintenance [Drools Documentation V4.x].

Although TREAT and LEAPS outperform RETE in some cases, RETE is widely implemented in general rule engines and a lot of optimizations have been developed (as will be discussed in the next section), improving its performance. In addition TREAT and LEAPS have not yet been implemented in any surveyed rule-entailment reasoners. As a matter of fact RETE is the major algorithm used by the surveyed rule-entailment OWL reasoners.

2.2.2.3 RETE Optimizations

RETE stores intermediate results, enabling rule matching to be performed incrementally based on previous matching/join results. This enhances reuse however may require effort and memory to maintain the RETE network. Some potential problems include beta memory explosion, inefficient deletion, and so on [McDermott et al 1978, Miranker 1987]. To approach these shortcomings a number of RETE optimizations have been proposed in the state of the art. This section mainly describes and discusses *RETE join sequence optimization heuristics*, a group of widely applied optimizations to reduce the resource consumption of beta memory by reordering the join sequences of conditions. Furthermore

how they are applied automatically by RETE optimizers is also discussed. Finally some other heuristics are also discussed briefly.

2.2.2.3.1 RETE Join Sequence Optimization Heuristics

By default conditions are joined in the sequence in which they appear in the rule, which in many cases is authored by domain experts, and therefore join sequences of these rules are not already optimized (in most cases they might not). However as indicated by previous research [Ullman 1982, Brownston et al 1985, Wang and Hanson 1992, Ishida 1994] the join sequences of condition elements can greatly impact the performance of join operations and an inappropriate join sequence (e.g. cross production join) could cause drastically more join operations to be performed, hence leading to a lot more memory (more intermediate results generated) and time required (more joins performed) by the beta network. Therefore some join sequence optimization heuristics have been proposed to cope with the excessive overhead caused by inappropriate join sequences. Two most widely applied are the *most specific condition first* [Ishida 1988, Wang and Hanson 1992, Ishida 1994, Özacar et al 2007] and the *pre-evaluation of join connectivity* [Scales 1986, Nayak et al 1988, Ishida 1988, Ishida 1994, Özacar et al 2007].

The *most specific condition element first* heuristic is derived from the observation that the more specific a condition element is it is likely to match less facts, and therefore pushing more specific conditions towards the front of a join sequence is more likely to generate less intermediate results in a beta network [Wang and Hanson 1992, Özacar et al 2007]. To determine the specificity of condition elements some criteria are proposed. The *counting matched facts* criterion is the most straight forward one which uses the number of matched facts of a condition as its specificity. However this criterion appears to be paradoxical, as this number usually cannot be decided before the execution of RETE engine. A second criterion is *counting variables*, which assumes the more variables a condition element has the less specificity the condition element has and the higher likelihood it generates more intermediate results. This criterion is easy to implement but sometimes is coarse in determining the specificity, e.g. it is difficult to determine the specificity if two condition elements have the same number of variables. Deciding the specificity of a condition element by the *complexity of the OWL predicate* is a criterion specific for OWL reasoners [Özacar et al 2007]. It assumes the alpha memory for conditions with complex predicate is more likely to be smaller than conditions with simple assertion (e.g. type) or subsumption predicates (e.g. subClassOf) [Zhang et al 2004, Özacar et al 2007]. Although easy to implement and more accurate, this approach requires a predefined partial ordering of OWL predicates

according to their specificity, which is (1) static, that is the same order is used regardless of the particular ontology to be reasoned, and (2) somewhat inaccurate, e.g. it is not always correct to say that owl:subPropertyOf is more specific than owl:subClassOf.

Pre-evaluation of join connectivity between condition elements is another widely adopted join sequence optimization heuristic [Scales 1986, Nayak et al 1988, Ishida 1988, Ishida 1994, Özacar et al 2007]. It ensures two joining condition elements should at least have one common variable or otherwise it changes the join sequence. This heuristic is designed to avoid Cartesian product joins.

There are some other less commonly applied RETE join sequence optimization heuristics. The *pushing volatile conditions last* heuristic [Ishida 1988, Ishida 1994] pushes the condition elements that match the frequently changing facts towards the end of the join sequence such that less joins are needed for changes. The *join cluster sharing* heuristic [Ishida 1988, Scales 1986, Ishida 1994] re-orders the join sequence to enable common join structures to be shared among rules, reducing the required efforts and resource to 1/n (if the sharable join structure is shared by n rules). However they are either not suitable for OWL reasoning (e.g. OWL ontology does not usually change) or highly constrained limiting their wide adaptation in general cases (e.g. the left-associative join tree of RETE network limits that only common join structures in the front of rules can be shared). Therefore they are not discussed in detail in this thesis.

2.2.2.3.2 Application of RETE Join Sequence Optimization Heuristics

Direct application of different RETE join sequence optimization heuristics may cause conflict with one another, e.g. pushing the most specific condition to the start of a join sequence may break the sharable join structure [Scales 1986, Ishida 1988, Wang and Hanson 1992, Ishida 1994, Özacar et al 2007]. Therefore some approaches are proposed to carefully plan the application of these heuristics.

In the work in [Scales 1986] a condition reorderer for SOAR, a RETE-based production system, is described to determine an optimized join sequence by repeatedly appending the "best" unordered condition to the ordered conditions (a best condition is the condition regarded by the reorderer to add most constraints to the existing ordered conditions). It classifies conditions into eight predefined ranks (ranging from 1 to 8) according to the static information that the reorderer knows about the conditions, that is if variables of an unordered condition is bound by the ordered conditions, if an unordered condition is the goal condition, and which attribute is multi-attribute (an attribute can have multiple values).

A condition with a lower rank is then regarded as one that adds more constraints than a condition with a higher rank. For example a condition with rank 1, i.e. condition with bound value in identifier, and constants in attributes and values, is regarded as to add more constraint than a condition with rank 8, i.e. condition with unbound variable in identifier. This approach is shown to be effective for SOAR. However it is specifically designed for SOAR rules and lacks generality. In addition, it uses only static information on the conditions to approximate optimal join sequence, which will generate the same join sequence even for two totally different fact bases.

In the work in [Ishida 1994] the author proposes to use a priori execution to collect join costs for all conditions and then to enumerate all possible join structures and use a predefined cost model to estimate the cost for them. The join structure with the minimal cost is considered as the optimal join structure. Three heuristics are used to constrain the number of enumerated join structures: connectivity, i.e. all generated join structures should have common variable, minimal-cost, i.e. newly generated join structure should have lower cost than previously generated ones, and priority, i.e. avoid joining lower priority condition if a higher priority condition is available. This approach can find an optimal RETE structure however there are two obvious drawbacks: (1) enumerating all possible candidate join structures and computing costs for them may require a large amount of resources, and (2) a priori execution might not always be practical, particularly for the context of this research where limited resources are available.

2.2.2.3.3 Miscellaneous RETE Optimizations

Some other optimizations are designed to speed up or improve RETE algorithm in other aspects, such as: using indices to speed up searching for intermediate results; using non-linear join structures to improve join performance or to enhance join structure sharing; enhancing RETE with the ability to process time sensitive events, and so on. They are described in this section.

Indexing is a frequently used optimization to speed up production systems. Work in [Kang and Cheng 2004] builds two indices for conditions: a sequential index for finding the most recent fact and a tree shaped index for performing joins. Work in [Özacar et al 2007] uses a pyramid technique [Berchtold et al 1998] to index ontological data for efficient access. [Scales 1986] suggests indexing facts by class type and attribute name to minimise the number of tests on alpha nodes. [Obermeyer et al 1995] suggests selectively building and applying a relatively simple index structure to speed up production systems.

Some research considers replacing the linear join network of RETE with other join structures such as a binary network or a bilinear network to gain more performance benefit. The work in [Scales 1986] discusses in detail three possible effects of using a non-linear join network, including the change of the number of generated intermediate results, the improvement on the likelihood of join cluster sharing, and the elimination of long-chain effects. However a potential negative effect is that Cartesian production joins could occur more frequently, which may easily cancel out the benefits. Uni-RETE [Tambe et al 1992] adopts a bilinear join structure as an alternative of the linear join structure to increase sharing of common join structure. Experiment results show that the bilinear version of uni-RETE requires less time than the linear version of uni-RETE and much less time than RETE. [Lee and Schor 1992] proposes a matching algorithm for generalized RETE network. Research in [Wright and Marshall 2003] proposes to prune the beta network at runtime so the amount of beta memory can be adjusted dynamically according to the availability of memory. Joins for the pruned part of the beta network is calculated on demand (as in TREAT). Therefore TREAT is a special case with all the beta networks pruned.

Optimizations are proposed to improve the poor deletion performance of RETE. For example the work in [Wright and Marshall 2003] proposes to use a search-based asymmetric deletion approach to speed up the deletion operation in RETE network. It keeps the original facts in intermediate results and therefore deletion turns out to be searching for all the intermediate results containing the deleted fact.

Some other optimizations that are not specially designed for RETE are also applicable or potentially applicable to it. For example, the work in [Schmolze and Snyder 1997] presents an approach for detecting redundant production rules for general production systems. The Gator network [Hanson and Hasan 1993] has been proposed as a generalized RETE/TREAT by allowing generalized join network. Similarly optimizations used in the Gator network [Hanson et al 2002] can also be studied and used by RETE optimizers. In addition given the close connection between RETE and query processing in database systems some optimizations strategies, such as query transformation heuristics, selectivity and cost estimation and so on, can be modified and applied for RETE optimization [Ullman 1982, Jarke and Koch 1984, Smitha and Geneseretha 1985, Elmasri and Navathe 2003].

While some of the optimizations discussed in this subsection are not the focus of this research, some of them, such as indexing, could be introduced to this research without fundamentally changing the approach described in later chapters.

2.3 *Related Work*

Related work of this this research is four-fold. Two surveys are presented in the first instance: a survey of OWL reasoners and a survey of semantic applications. There were two goals for the survey of OWL reasoners: a categorization of OWL reasoners that enables the reasoner composition research to be conducted on a general grounding and the distillation of a set of reasoner characteristics. The survey of semantic applications helps the understanding of the complicated interplay between semantic applications and OWL reasoners, which may facilitate the design of an automatic reasoner selection process. Results of these two surveys form the foundation of research carried out in this thesis. Then existing reasoner composition approaches are presented. By discussing the existing reasoner composition approaches, gaps are identified which the automatic reasoner composition research can attempt to fill. Finally since the reasoner composition research is targeted at resource-constrained OWL reasoners, existing resource-constrained OWL reasoners are examined and how they are designed in order to more efficiently run on resource-constrained devices are discussed.

2.3.1 Survey of OWL Reasoners

As informed by research objective 1, a survey of state of the art OWL reasoners was performed on a set of 26 state of the art reasoners. These reasoners were encountered while the author was doing literature review. Two goals are targeted for this survey. First, a categorization of OWL reasoners needs to be obtained in order to enable the reasoner composition research to be conducted on a type of reasoner rather than a particular reasoner implementation, providing this research with a more general grounding. Therefore to categorize reasoners can provide a foundation for this research. A second goal needs to be achieved in the survey is that a set of reasoner characteristics needs to be distilled, which may help the automatic reasoner selection research. A summary of the results of this survey can be found in appendix A. In the next two subsections how the above two goals are achieved in this survey is discussed in detail.

2.3.1.1 A Categorization of OWL Reasoners

As mentioned earlier the goal for obtaining a categorization of reasoners is to provide a general grounding for the reasoner composition research. A natural approach identified by the author was to obtain this categorization based on the reasoning algorithms, since to compose OWL reasoners at the reasoning algorithms level tends to be the most intuitive way to conduct the reasoner composition research given the motivation to reduce the resource consumption of OWL reasoners.

A similar previous categorization was found in [Zou et al 2004] which categorizes OWL reasoners into three types according to logic that OWL can be translated into: OWL reasoners using specialized DL engines, OWL reasoners using full first order logic (FOL) theorem provers, and OWL reasoners using a reasoner designed for FOL subsets. This categorization to some extent satisfied the needs of the author as described above. However, there are some aspects of this categorization that impeded its use for the automatic reasoner composition research. First, this categorization is not based on reasoning algorithms and therefore the reasoning algorithms may be different even for reasoners of the same type. For example, this categorization does not distinguish between rule-based reasoners using forward-chaining rule engines and those using backward-chaining logic programming (LP) engines although both of them belong to OWL reasoners using a reasoner designed for FOL subsets according to this categorization. To distinguish them is important for the automatic reasoner composition research since forward-chaining rule engines and backward-chaining LP engines use different rule matching algorithms, which may need totally different automatic composition approaches. A second aspect is that there have been reasoning algorithms developed since this categorization was created, and new reasoners have emerged that do not fall into any category of this categorization, e.g. hybrid reasoners such as DLEJena [Meditskos and Bassiliades 2010] combine more than one reasoning algorithms.

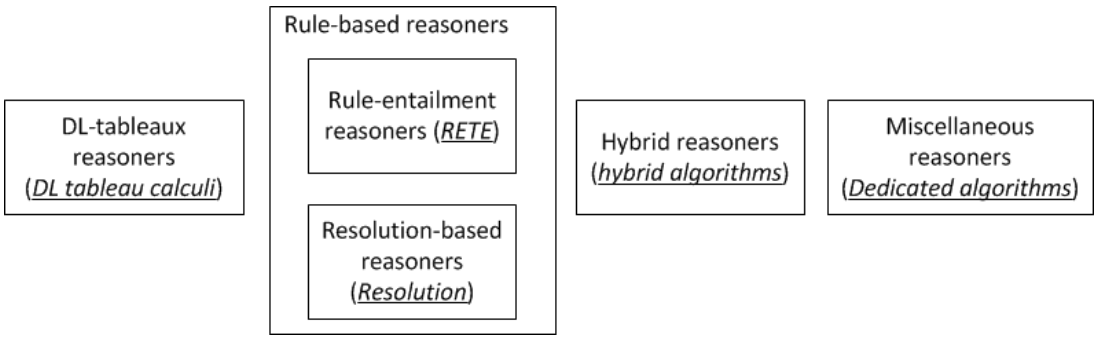


Figure 2-2: Reasoner categorization used in this thesis

Hence, a new categorization was created by the author, based on the reasoning algorithms of OWL reasoners (as given in Figure 2-2). The new categorization was inspired by the above categorization, however with finer categorization. The first category, namely OWL reasoners using specialized DL engines, is kept however renamed as *DL-tableaux reasoners*. The last two types in the previous categorization, i.e. OWL reasoners using full first order logic (FOL) theorem provers and OWL reasoners using a reasoner designed for FOL subsets, are replaced by *rule-based reasoners using forward chaining engines* (*rule-entailment*

reasoners) and *rule-based reasoners using backward chaining engines* (*resolution-based reasoners*). Furthermore two new reasoner categories are added in correspondence to the development of new reasoning algorithms: the *hybrid reasoners* that combine more than one type of reasoners and the *miscellaneous reasoners* that differentiate other less common reasoning algorithms. This new categorization was then used in the survey of 26 state of the art reasoners. In the following subsections, each reasoner category is described.

2.3.1.1.1 DL-Tableaux Reasoners

DL-tableaux reasoners convert OWL axioms into DL axioms and then reduce OWL entailment to determining the satisfiability of the reduced KB, that is to check there is a valid model for the reduced KB [Horrocks and Patel-Schneider 2004a]. As DL tableaux calculus is widely adopted in determining KB satisfiability [Baader and Sattler 2001, Baader et al 2007], reasoners of this type are termed DL-tableaux reasoners.

KB satisfiability checking is the key DL reasoning task as (1) arbitrary conclusions can be drawn from a contradictory KB [Tobies 2001], and (2) other DL reasoning tasks such as concept satisfiability, concept subsumption, instance checking and so on can be reduced to it with the presence of negation in the logic [Baader et al 2007]. For example given C and D as two concept names, $\mathcal{T}$ a TBox[6], a an individual, the subsumption $C \sqsubseteq D$ can be reduced to checking the unsatisfiability of KB $\{\mathcal{T}, (C \sqcap \neg D):a\}$.

In general, DL tableaux algorithm applies a set of (often hard-coded) consistency-preserving transformation rules to the ABox[7] until no more rules apply. If the transformed ABox obtained this way consists a contradiction, then the ABox is consistent and inconsistent otherwise. An example transformation rule could be like:

The $\rightarrow\sqcap$ - rule

Condition: $\mathcal{A}$ contains $(C_1 \sqcap C_2)(x)$, but it does not contain both $C_1(x)$ and $C_2(x)$.

Action: $\mathcal{A}' = \mathcal{A} \bigcup \{C_1(x), C_2(x)\}$.

This rule basically states that if the original ABox $\mathcal{A}$ contains a conjunction of two concepts

[6] TBox is one of the two components of a typical KB. It contains intentional/terminological knowledge. The other component of a KB is ABox.
[7] ABox is one of the two components of a typical KB. It contains extensional/assertional knowledge. The other component of a KB is TBox.

C_1 and C_2, and they are not individually included in $\mathcal{A}$, then the newly transformed ABox $\mathcal{A}'$ needs to included $C_1(x)$ and $C_2(x)$. Hence if $C_1 \sqcap C_2$ is not satisfiable, say $C_1 = D$ and $C_2 = \neg D$, an obvious contradiction will be detected in the $\mathcal{A}'$ that the individual x belongs to two contradicting concepts D and ¬D. In order for better performance and ensure its termination, transformation rules are usually hardcoded and many optimizations are developed and applied. More detail on DL tableaux algorithms can be found in [Baader and Sattler 2001, Baader et al 2007].

While DL tableau calculi usually perform sound and complete TBox reasoning for the supported DL, they usually have difficulties in efficiently reasoning large ABox [Haarslev and Möller 1999, Horrocks et al 2004, Motik and Sattler 2006, Meditskos and Bassiliades 2008a]. This has been partly solved by some newly proposed approaches such as DLE reasoning [Meditskos and Bassiliades 2008a], a hybrid approach using DL tableau calculi for efficient TBox reasoning and forward-chaining Datalog rules for scalable ABox reasoning, and hypertableau [Motik et al 2009], a variety of DL tableau calculi combining the idea of hyper-resolution to reduce the non-determinism. Since a hybrid approach, e.g. combining DL-tableaux reasoner with rule-based reasoner, is used in the DLE approach, reasoners using the DLE approach are considered in this categorization as within the hybrid reasoner category which will be introduced in section 2.3.1.1.3. Reasoners using the hypertableau algorithm are still considered as DL-tableaux reasoners.

Some OWL reasoners of this type include FaCT++ [Tsarkov and Horrocks 2006], Pellet [Sirin and Parsia 2007], RacerPro[8], and HermiT [Motik 2007], among which HermiT is the first and only reasoner implementing the hypertableau calculus while the others implement DL tableau calculi.

2.3.1.1.2 Rule-based Reasoners

The interaction between rules (or FOL implication) and ontology is a long-discussed issue in the OWL community. Many approaches are proposed for combining them and a good number of rule-based reasoners have been implemented. According to the different algorithms used to perform reasoning, they fall into two reasoner categories: the forward chaining rule-entailment reasoners that use forward chaining (RETE-based) rule engines and the backward chaining resolution-based reasoners that use resolution-based backward chaining engines (including partial or full FOL engine). Before going into more detail about

[8] http://www.racer-systems.com/

the above two reasoner categories, a general description is given on how OWL ontology interoperates with rules in order to achieve rule-based OWL reasoning.

2.3.1.1.2.1 A General Description of Rule-based OWL reasoning

In order to process an ontology using rule-based approaches, the ontology needs to be converted into a rules-compatible form first. This conversion can follow two approaches: an *ontology independent* approach that translates OWL ontology syntactically into a set of facts and (partial) OWL semantics into a fixed set of ontology-independent entailment rules, and an *ontology specific* approach that semantically transforms the OWL ontology into a set of facts/ontology-specific rules according to the OWL direct semantics (following the DLP approach [Grosof et al 2003]). For example given a set of OWL axioms as

Individual(a type(C))

Class(*B* partial *C*)

Class(*C* partial *D*)

Class(*D* complete restriction(*P* someValuesFrom *E*)

which states D is a someValuesFrom constraint restricting the existence of instances of class E on property P, and C is a subclass of D. An ontology independent translation could be a set of facts in triple format such as

a rdf:type *C*

B rdfs:subClassOf *C*

C rdfs:subClassOf *D*

D rdf:type owl:Restriction

D owl:onProperty *P*

D owl:someValuesFrom *E*

and ontology-independent OWL entailment rules can be implemented as

(?*a* rdfs:subClassOf ?*b*), (?b rdfs:subClassOf ?c) → (?*a* rdfs:subClassOf ?*c*)

(*?a* rdfs:subClassOf *?b*), (*?c* rdf:type *?a*)→ (*?c* rdf:type *?b*)

(*?a* owl:someValuesFrom *?b*), (*?a* owl:onProperty *?p*), (*?o* rdf:type *?a*), makeTemp(*?x*) → (*?o ?p ?x*).

However an ontology specific translation of the example OWL axioms could generate a set of ontology specific rules (FOL implication):

$C(a)$

$\forall x.C_L(x) \rightarrow D_L(x)$

$\forall x.B_L(x) \rightarrow C_L(x)$

$\forall x \exists y.D_L(x) \rightarrow P_L(x, y) \wedge E_L(y)$

where B_L, C_L, D_L,E_L and P_L are the rule predicates for the corresponding classes or predicate.

It is clear from the above examples that the ontology independent approach translates OWL ontology into a set of facts and uses ontology independent entailment rules to perform OWL reasoning. Entailment rules are usually static and they are fine grained in terms of the supported OWL semantics: each rule usually realizes a fixed and small portion of OWL semantics. Furthermore entailment rules are independent of the ontology and therefore any changes to the ontology do not change the rule set.

However, it can be seen from the above examples, the ontology specific approach translates OWL axioms into rules (or FOL implication) by embedding the OWL semantics in this translation. The rule program (FOL program) obtained through this way is a direct and exact rule translation of the ontology and since OWL semantics are embedded in the translation no separate OWL entailment rules are required to perform reasoning. However some drawbacks also exist. Firstly, this approach embeds semantics in the translation, and therefore the same semantics could be realized multiple times for similar axioms, causing extra effort in the translation. For example, semantics for rdfs:subClassOf are realized twice for both Class(*B* partial *C*) and Class(*C* partial *D*), generating two different rules. Secondly, compared to the ontology independent approach, although the ontology specific approach may generate less complex individual rules (complex concept axioms, e.g. GCI, will still lead to complex rules to be generated by the ontology specific translation), many

more rules than the OWL entailment rules will be generated (for each concept/property axiom in the ontology a rule is generated).

Depending on the different paradigms/algorithms used to process the translated rule program, rule-based reasoners are further divided into two reasoner categories: the (forward-chaining) rule-entailment reasoners and the (backward-chaining) resolution-based reasoners. Figure 2-3 gives a general structure of rule-based reasoner. The next two subsections describe these two categories in detail. As will be found out that most surveyed rule-entailment reasoners use the ontology independent approach and the resolution-based reasoners use both translation approaches evenly.

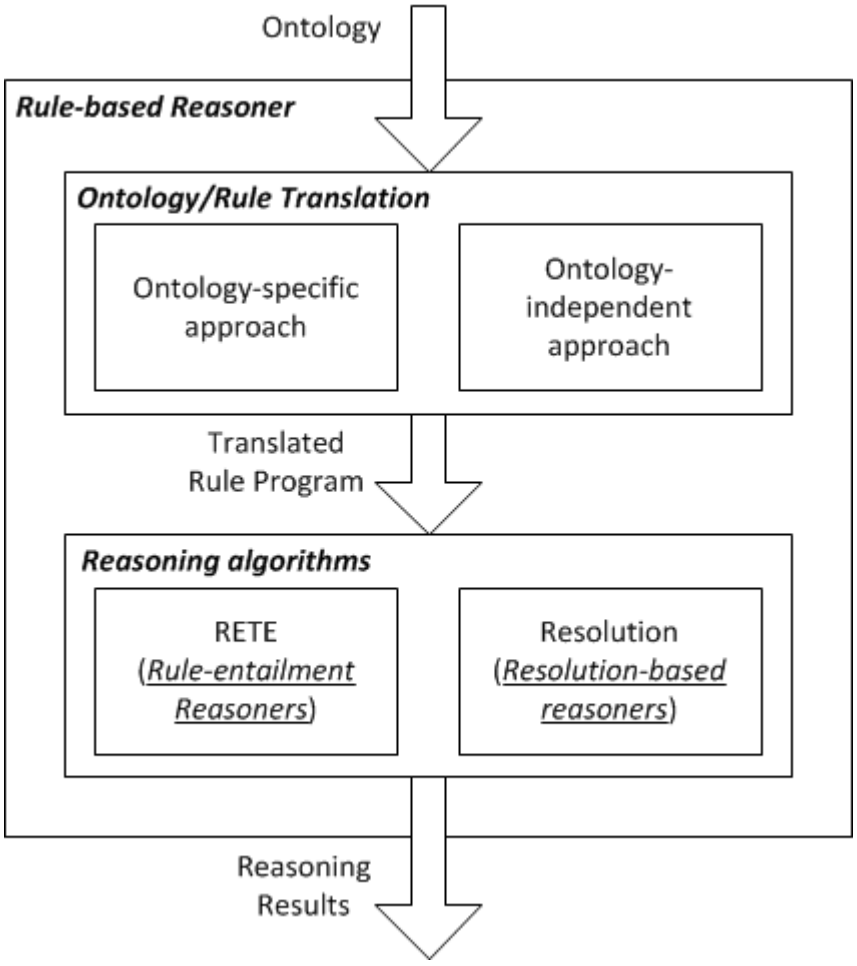


Figure 2-3: A general structure of rule-based reasoner

2.3.1.1.2.2 (Forward Chaining) Rule-Entailment Reasoners

Forward chaining rule-entailment reasoners (for brief they will be referred to as *rule-entailment reasoners* in the reminder of this thesis) use (often RETE-based) forward chaining engine to process the translated rule program. *Ontology entailment* is the major

reasoning task for rule-entailment reasoners. It checks if a target ontology O' is a logical consequence of a given source ontology O. If so, O is said to *entail* O' (or O' is entailed by O) and it is written down as $O \models O'$. In general a rule-entailment reasoner fully calculate all possible entailments according to the OWL semantics either expressed as static entailment rules or embedded in the translated ontology-specific rule program. As a typical algorithm for rule-entailment reasoners, RETE materializes all inferred entailments and therefore queries can be laid on the 'completely' inferred ontology, enabling fast query evaluation [Kiryakov et al 2005, ter Horst 2005a, Meditskos and Bassiliades 2008b].

Bossam [Jang and Sohn 2004], OWLIM [Kiryakov et al 2005], OWLJessKB [OWLJessKB 2011], OWL2Jess [Mei et al 2005], BaseVISor [Matheus et al 2006], all use the ontology independent approach. O-DEVICE [Meditskos and Bassiliades 2008b] combines the ontology independent approach for TBox reasoning and the ontology specific approach for ABox reasoning.

2.3.1.1.2.3 (Backward Chaining) Resolution-based Reasoners

Backward chaining resolution-based reasoners (for brief they will be referred to as *resolution-based reasoners* in the reminder of this thesis) translate OWL ontology into a rule program (using full FOL clauses or partial FOL clauses such as Prolog, Datalog, Flora) and hand-off OWL reasoning to the corresponding resolution engines (refer to [Nerode and Shore 1997] for resolution). The goal-directed nature of resolution enables reasoning to be performed at query evaluation time and therefore in contrast to rule-entailment reasoners in theory resolution-based reasoners do not require a priori full entailment calculation. Therefore it is expected that a resolution-based reasoner may have more flexibility in efficiently answering queries over a frequently changing KB. However some resolution-based reasoner implementations still pre-calculate and materialize (part of) entailments in order to gain better runtime query performance, e.g. KAON2 [Hustadt et al 2004a].

Resolution-based reasoners using the ontology specific translation include KAON2 [Hustadt et al 2004a], Thea [Vassiliadis et al 2009], Hoolet [Tsarkov et al 2006] and Bubo [Volz et al 2003]. Resolution-based reasoners using ontology independent translation include F-OWL [Zou et al 2004], and Surnia [Surnia 2011].

2.3.1.1.3 Hybrid Reasoners

Some OWL reasoners combine two of different algorithms in order to take advantage of both. Reasoners using this hybrid reasoning approach are categorized as *hybrid reasoners*.

Two well-known hybrid reasoners are Minerva [Zhou et al 2006] which uses DL-tableaux reasoners to handle TBox reasoning and uses RDBMS and DLP rules to achieve scalable ABox reasoning, DLEJena [Meditskos and Bassiliades 2010], a hybrid reasoning framework delegating terminological reasoning to DL-tableaux reasoners and ABox reasoning to a rule-entailment reasoner [Meditskos and Bassiliades 2008a], and Jena [Carroll et al 2004] which is a hybrid engine combining a resolution engine to evaluate the backward chaining part of Jena rules and a RETE engine to evaluate forward-chaining part of Jena rules. Pellet now supports rules as well but it is still primarily a DL reasoner, therefore it is categorized into the DL-Tableaux reasoners.

2.3.1.1.4 Miscellaneous Reasoners

There are some other OWL reasoners using dedicated reasoning algorithms other than those mentioned above in order to have efficient reasoning services for some specific purposes. CEL uses a polynomial time classification algorithm for DL EL++ [Baader et al 2006]. Owlgres [Stocker and Smith 2008], QuOnto [Acciarri et al 2005] and MASTRO [Calvanese et al 2011] are designed to provide efficient conjunctive query answering algorithm over large ABox and an algorithm specifically designed for the DL-Lite subset of OWL is used combining TBox knowledge in query evaluation [Calvanese et al 2007]. The latest version of Owlgres (can be found here[9]) supports efficient conjunctive OWL 2 QL reasoning. Oracle database 11g [Wu et al 2008] evaluates entailment rules by converting them into equivalent SQL statements and evaluates them in RDBMS, enabling OWL ontology to be stored and reasoned in Oracle database. The SPIN technology [SPIN 2011] authors the OWL 2 RL rules using SPARQL rules (with the assistance of the CONSTRUCT keyword) and handles OWL reasoning using SPARQL engine. Although a similar approach as rule-entailment reasoners or some of the resolution-based reasoners is used where OWL ontology is reduced to rules and entailment rules (SPARQL rules) are used to express meaning of OWL vocabulary, the implementations of SPARQL engine vary and therefore this approach is classified as miscellaneous reasoner.

Some other algorithms are also proposed to reason DL ontologies, for example the structural subsumption algorithm [Baader et al 2007] tests subsumption by analysing the syntactic structure of normalized DL axioms, however it is not yet implemented in any OWL reasoners and therefore it is out of the scope of this thesis.

[9] http://pellet.owldl.com/owlgres/

2.3.1.2 Reasoner Characteristics

As mentioned earlier a second goal of having a survey on state of the art reasoners is the distillation of a set of reasoner characteristics. In this survey, 18 reasoner characteristics were identified and surveyed (as given in Table 2-2). The selection of these reasoner characteristics was based on an initial survey over semantic reasoners published online[10] in 2009. Although this survey is an online document and may have not received strict peer reviews, its selection of reasoner characteristics is reasonable and therefore it serves as a good basis to start with. However the distillation of reasoner characteristics was not a one-off effort and they were not fixed even after the survey had started. Some reasoner characteristics from the initial survey were removed (*version* and *licensing*) since they appear little relevance to application characteristics of semantic applications. Some others were combined for brevity (*OWL-DL entailment* and *consistency checking* were combined into *reasoning tasks*). More reasoner characteristics were added by the author. Some are common for OWL reasoners such as reasoning completeness, query support, concrete domain reasoning. Some others characteristics such as *incremental reasoning*, *closed-word features* were inspired when surveying semantic applications. They were then added into this survey. A full list of the surveyed reasoner characteristics and their descriptions can be found in Table 2-2. For each reasoner characteristic a code (given in parenthesis) is assigned in order for better brevity and preciseness when referenced. Surveying values of reasoner characteristics for reasoners mainly relied on two approaches: literature review (including published papers, webpages, and product release documents) and looking into the API documents, which required a large amount of reading and researching: more than 90 different types of literature were reviewed. Still the values of some reasoner characteristics are unknown for some reasoners. A detailed discussion of how the correlation between these reasoner characteristics and characteristics of different types of applications can be found in section 4.3. Detailed results of the survey can be found in Appendix A.

[10] http://en.wikipedia.org/wiki/Semantic_reasoner

Table 2-2: Reasoner characteristics used in the survey of OWL reasoners

Reasoner characteristic	Descriptions
Reasoning algorithm (ALGM)	The algorithm used by the reasoner.
Reasoner type (TYPE)	The type of the reasoner.
Expressivity (EXPR)	The expressivity of the reasoner.
Completeness (CPLT)	If the reasoner can completely reason its supported expressivity.
Reasoning tasks (TASK)	The reasoning tasks supported by the reasoner.
Materialization (MTLZ)	If the reasoner materialize reasoning results.
Incremental reasoning (INCR)	The types of incremental reasoning supported by the reasoner.
Query support (QUERY)	The type of queries the reasoner supports.
Rule support (RULE)	The type of rule languages the reasoner supports.
Closed-world features (CWA)	The type of closed-world features the reasoner natively supports.
Concrete domain (CD)	The type of concrete domain reasoning supported by the reasoner.
Database (DB)	How database is supported by the reasoner.
Remote interface (RINF)	If the reasoner supports remote interface.
User access (ACCESS)	How users can access the reasoner.
Explanation (EXPL)	If the reasoner provides reasoning explanation function.
Ontology manipulation (MANI)	How ontology can be manipulated and accessed by applications.
Platforms (PLAT)	The platforms required by the reasoner.
OS (OS)	The operating systems that the reasoner can run on.

2.3.2 Survey of Semantic Applications

This section discusses the survey of five diverse sample types of semantic applications. They include *semantic publish/subscribe systems*, *semantic context-aware systems*, *medical and bioinformatics systems*, *semantic sensor network management systems*, and *software engineering systems*. Their selection is motivated by the fact that the use of OWL reasoning technologies to solve problems within them has been widely investigated, and it is shown that the use of reasoning technologies indeed addresses some problems within these domains. The rationale behind this discussion is two-fold: (1) to enable the author to learn the interplay between semantic applications and OWL reasoners, and to showcase the amount of intricacies application developers need to know in order to select an appropriate reasoner using a manual reasoner selection approach, and (2) to assist the identification of a set of example reasoning-related application characteristics.

2.3.2.1 Semantic Publish/Subscribe Systems

In publish/subscribe systems (pub/sub systems) subscribers register subscriptions in a broker (or networked brokers) and publishers present publications to the broker. A conventional pub/sub broker syntactically matches the content of publications against registered subscriptions. Successfully matched publications are propagated to the corresponding subscribers. However semantic pub/sub systems extend this approach by matching based on the semantics of publication/subscription, informed by an associated ontology and facilitated by an ontology reasoner in the broker.

In general publications in such systems are semantically annotated and the semantic filtering is delegated to either subsumption checking, conjunctive queries answering [Keeney et al 2008] or instance checking [Haarslev and Möller 2003a, Ushchold et al 2003] according to the manner that publications/subscriptions are modelled. Thus the systems which use concept subsumption to perform semantic filtering would require concept-centric reasoning and in some cases may require to reason over very expressive ontology [Ushchold et al 2003]. On the other hand the systems based on conjunctive query answering and instance checking may rely more on data-centric reasoning tasks [Haarslev and Möller 2003a, Ushchold et al 2003].

Some other reasoning related characteristics that are common to semantic pub/sub systems are also observed. The first and most common one is the ability to efficiently reason and query over a frequently changing knowledge base. This is a problem encountered by many semantic pub/sub systems as most existing OWL reasoners are designed to reason over static datasets. To cope with this issue incremental reasoning approaches and incremental

query answering approaches are developed and integrated into some semantic pub/sub systems [Haarslev and Möller 2003a, Halaschek-Wiener and Kolovski 2008]. A second characteristic is the vast amount of individuals, motivating the usage of a database-enabled reasoner [Ushchold et al 2003]. Thirdly, some systems need datatypes to model concrete information such as date, time or space [Ushchold et al 2003], emphasizing the ability to reason over concrete domains. Fourthly, for some applications such as battle field systems or stocks exchange systems, a complete OWL reasoner would be critical for them to function correctly. A fifth characteristic is the ability to perform temporal or spatial reasoning, which is of particular importance for complex event systems [Ushchold et al 2003, Keeney et al 2008, Keeney et al 2010]. Other characteristics of some particular pub/sub systems include the ability to handle closed-world queries [Haarslev and Möller 2003a] and so on.

2.3.2.2 Semantic context-aware systems

Semantic context-aware systems are another type of applications where the use of OWL ontology and its reasoning technologies are well studied. There is now a consensus that the use of OWL ontology and reasoning technologies can (1) provide a more expressive method to model and process complex context such as human interests or activities that are hard to model using attribute/value pairs [Agostini et al 2005], (2) bring more intelligence into the utilization and aggregation of ambient information, enabling the personalisation and adaptation of application behaviours or services [Luther et al 2008, Ejigu et al 2007], and (3) enhance the interoperability between heterogeneous environmental entities engaged in the domain by allowing knowledge to be correctly interpreted and reasoned in different entities [Ali and Kiefer 2009].

An important characteristic of many semantic context-aware systems is the need for rules to perform (application-specific) reasoning tasks such as triggering actions or aggregating context to derive high-level context [Wang et al 2004, Weißenberg 2004, Agostini et al 2005, Chen et al 2005, Gu et al 2007, Ejigu et al 2007, Luther et al 2008, Ali and Kiefer 2009]. Therefore it is vital for them to be able to use rule-enabled OWL reasoners, e.g. to integrate a SWRL component in the reasoner, or to interface a standalone rule engine for rules processing outside the reasoner. A second characteristic is the use of conjunctive queries (i.e. a type of query where conditions are connected using conjunction) to perform more powerfully and (possibly) human-accessible context retrieving [Chen et al 2005, Gu et al 2007, Ali and Kiefer 2009]. Thirdly in some systems the aggregation of low-level context data needs to process concrete values, emphasizing the support of concrete domain

reasoning in the selected reasoner [Weißenberg 2004, Luther et al 2008, Chen et al 2005, Boehm et al 2008]. Existing OWL profiles do not process more complex concrete domain relations other than comparison of datatypes, and so complex concrete values processing are mostly performed by rule engines, where many builtins, such as algebraic comparison and computations, are constructed to handle complex concrete value processing. However this is still considered as a potential reasoning related characteristic, as this can be handled by rule-enabled OWL reasoners. Fourthly some context-aware systems use repositories to store context [Gu et al 2007, Boehm et al 2008], which then requires the reasoner to be able to access and reason over databases. Some other characteristics also exist such as using the DL Implementation Group (DIG) interface [DIG] to connect distributed applications and reasoners [Luther et al 2008], using incremental reasoning to incrementally handle context data increasing the response speed [Luther et al 2008], temporal reasoning [Chen et al 2005, Boehm et al 2008] and the need to perform runtime ontology manipulation [Boehm et al 2008].

2.3.2.3 Clinical, Medical and Bioinformatics Systems

OWL and OWL reasoning technologies have also been applied in clinical, bioinformatics and medical projects to enable knowledge to be modelled in a more formally defined and structured manner, which (1) facilitates the sharing and reusing of knowledge and (2) enables more intelligent data processing through OWL/application-specific reasoning. Some well-known projects/ontology are the Gene project [Harris et al 2004] which provides a controlled vocabulary of terms (concepts) for describing genes and gene product attributes; SNOMED [héja et al 2008] which provides a scientifically validated set of terms for practitioners to structure and computerize medical records, enhancing the sharing of medical records; the openGALEN project [openGALEN] which aims to construct a reusable and application independent ontology for medical procedures; the National Cancer Institute thesaurus (NCI) [Golbeck et al 2003] which aims to construct ontology on the vocabulary used in the cancer domain; and finally MGED which is aimed to develop ontology for describing samples used in microarray experiments.

As an ontology approach is used in the above projects to define structured terminology, an important characteristic of these projects is their large and sometimes expressive TBox, which therefore emphasizes the requirement for an efficient and complete classification reasoning service to be provided by the underlying reasoner. A second characteristic is the support of a more powerful access mechanism, e.g. conjunctive query answering in graphical user interfaces, to enable users to query and browse the ontology. This

characteristic is not explicitly mentioned for most of the above projects however it can be deduced from that fact that they are often large in size to be jafit for manually browsing and accessing (e.g. GO has more than 47,000 concepts, SNOMED has more than 364,000 concepts).

The research in [Keet et al 2007] identifies nine requirements that OWL-based bio-ontologies may have on OWL reasoning. They are: supporting the ontology development process; classification; model checking; finding gaps in an ontology and discovering new relations; comparison of ontologies; reasoning with mereological parthood and other (part-whole) relations; using a hierarchy of relations; reasoning across linked ontologies; and complex queries. Some of these, e.g. classification, can be solved by existing OWL reasoners, whereas some others, e.g. finding gaps and new relations, are quite specific to life science and were not yet feasible for general OWL reasoners.

2.3.2.4 Semantic sensor network systems/sensor ontology

Semantic Web technologies are widely applied in sensor network systems. A typical usage is to annotate sensor readings (or sensors descriptions) using semantically rich tags to facilitate intelligent data processing [Calder et al 2010, Kim et al 2008, Eid et al 2007] and to increase interoperability [Russomanno et al 2005, Sheth et al 2008, Eid et al 2007]. Another usage is to perform complex management tasks, e.g. sensor tasks assignments [Gomez et al 2008] or fault correlation [Brennan et al 2009].

The ability to process rules is considered as an important characteristic for many sensor network systems in order to perform sensor observation validation [Calder et al 2010], sensor observation processing [Sheth et al 2008], and network management [Brennan et al 2009]. A second characteristic is the ability to process complex queries, in particular conjunctive queries [Sheth et al 2008, Russomanno et al 2005, Eid et al 2007, Kim et al 2008]. It provides semantic sensor network systems with a powerful approach to retrieve information from ontology. Third the support of database is also a vital characteristic of some systems to store sensor observations [Sheth et al 2008, Calder et al 2010]. It is worth noting that although the capability to handle concrete domain objects such as numbers, time and so on is not specifically mentioned by most systems, it is again not difficult to infer that the ability to handle concrete objects is also an important characteristic in such system where sensor observations are mainly comprised of simple datatypes, e.g. numbers.

Some other characteristics include the provision of graphical interface to allow users to specify rules [Calder et al 2010], to use distributed reasoning to decentralise data processing

workload [Calder et al 2010], to complete OWL-DL classification to ensure complete and accurate sensor mission assignment [Gomez et al 2008], remote reasoning [Sheth et al 2008], explanation of deductions [Gomez et al 2008] and resource-constrained reasoning [Brennan et al 2009]. Furthermore the development semantic sensor ontology usually requires the support of ontology authoring tools such as Protégé, so a good integration with the ontology authoring tool of the selected reasoner can be considered as an important characteristic enabling fast prototyping.

The work in [Compton et al 2009a] surveys a set of 12 sensor ontologies and points out that conjunctive queries, rules and OWL reasoning were key technologies to provide semantics support at different layers in semantic sensor networks.

2.3.2.5 Software engineering

OWL and OWL reasoning technologies are also applied in the area of engineering systems to perform varieties of tasks ranging from detecting inconsistencies in software/system configurations [Shahri et al 2007, Kaviani et al 2008], to semantic-based source code searching [Keivanloo et al 2010], and bug tracking [Schuegerl et al 2008].

Characteristics vary from system to system. A relatively common characteristic is the use of conjunctive query languages. As these systems often interact with human software developers or system administrators, the capability to answer conjunctive queries turns out to be a major method on which users can rely to access the information [Keivanloo et al 2010, Kaviani et al 2008, Schuegerl et al 2008]. In addition a GUI for posing queries is sometimes important for users without background knowledge on the conjunctive query language in use in the system [Keivanloo et al 2010]. Other characteristics include the requirement to provide justification for inconsistencies in configuration [Shahri et al 2007], the use of a database to store meta-models of the source code [Shahri et al 2007] or for the bug repository [Schuegerl et al 2008].

2.3.3 Reasoner Composability

The categorization of OWL reasoners in section 2.3.1.1 classifies OWL reasoners into five types according to their reasoning algorithms. Discussion of the composability is required for each type of reasoner in order to identify a reasoner type that is better suited as a starting point for the automatic reasoner composition research of this thesis.

As aforementioned, DL-tableaux reasoners adopt highly complicated DL-tableaux algorithms to perform OWL reasoning. Although the use of transformation rules to detect

obvious contradictions in the KB shows some potential for composition (a rule set can be easily decomposed and composed to retain only required rules), DL-tableaux reasoners still have some limitations in terms of composability. Firstly, compared to rule-based reasoners, transformation rules are still coarse-grained in terms of OWL semantics. For example, the pD* entailment rule set has 41 fine-grained entailment rules however the tableau algorithm for the DL $\mathcal{SHIF}$ (with a similar OWL expressivity as pD*) only has 7 transformation rules [Horrocks et al 2000]. Hence the amount of semantics implemented in each pD* entailment rules is finer grained than that in each tableau transformation rule, leading to better potential for composition. Secondly, transformation rules are hard coded in the tableau algorithms in order for better performance, which may complicate the composition algorithmThirdly, a lot of complicated optimizations and loop detection approaches (blocking) are hardwired in practical DL-tableaux reasoner in order to ensure the termination of DL-tableaux algorithms [Baader et al 2007]. In such cases the application of composition algorithms may greatly increase the complexity of the tableau algorithm: the automatic composing of transformation rules requires different blocking approaches to be swapped in/out on the fly.

The heave adoption of rules in rule-based reasoners shows good composability at a first glance. A natural and straightforward approach to conduct reasoning composition is to add/remove the OWL entailment rules depending on the particular ontology to be reasoned. Then reasoners using the ontology-specific translation appear to have a good potential to be composed: the fine-grained, text-coded nature of the rule set (e.g. OWL entailment rules as well as domain specific rules) enables a selective rule set to be constructed for the particular ontology to be reasoned. The ontology-specific translation approach embeds OWL semantics in the translation and therefore the translation itself can be viewed as a composition process: the translation can always generate a rule program with the exactly required amount of OWL semantics for the ontology. However there are some potential limitations for performing reasoner composition using the ontology-specific translation: (1) the ontology-specific translation is limited to OWL semantics only and hence lacks flexibility to handle other semantics (modelled as domain specific rules), especially domain specific semantics which may be onerously required in some applications [Calder et al 2010, Sheth et al 2008, Compton et al 2009a, Brennan et al 2009, Rector 2002, Ejigu et al 2007], and (2) unlike ontology-independent translation which is straightforward and easy to perform, the ontology-specific translation may require analysis of the structure of the ontology, which may be non-trivial for resource-constrained devices.

Some miscellaneous reasoners are tightly bound to specific underlying implementation

mechanisms, such as SPIN which uses a SPARQL engine; Oracle uses RDBMS; and the composability of those mechanisms are outside the scope of this thesis. Some other reasoners such as CEL, QuOnto are designed for specific reasoning tasks, specific ontology expressivities, or specific application areas so their lack of general applicability, low expressivity and hardwired algorithms makes them less appealing as candidates for composability studies. The composability of a hybrid reasoner relies on each individual reasoner of the hybrid reasoner and it is already discussed above.

In summary, rule-based reasoners have better potential for composition and hence it shows more suitability than the other reasoner types for the automatic reasoner composition research. To explore how the automatic composition research can fit in rule-based reasoners, existing reasoner composition approaches used in rule-based reasoners are examined in the reminder of this section. In fact, at the moment no existing OWL reasoner claims to be a composable reasoner or to use reasoner composition approaches. Still, some mechanisms have been adopted by some reasoners in order to (potentially) (mostly manually) compose their reasoning capabilities/algorithms. These mechanisms are discussed.

All surveyed rule-based reasoners support one or more of three types of composition: the selection of one of several *predefined reasoning levels* via the reasoner API, an *editable rule set* allowing users to able to re-write or change the entailment rule base (although this is not deliberately aimed at composition by most implementers) and the support for some *algorithm-level* composition mechanisms. A typical example is Jena that allows users to select from three predefined reasoning levels, OWL, OWL Mini and OWL Micro, forming the first type of composition. In addition its use of plain text encoded rule files potentially allows users to freely modify the rule set to construct their own reasoning level, forming the second type of composition. These two types of composition are easy to implement however are limited in some aspects. Firstly their applications are manual, therefore requiring user knowledge of OWL reasoning. For example, users need to have enough knowledge on the required amount of reasoning capability in order to decide the correct reasoning level, or users need to be familiar with the semantics and the syntax of the rule language to author a suitable rule set, which is sometimes hard as the rule language used by many OWL reasoners is not formally documented, e.g. BaseVISor. Furthermore, manual composition mechanisms will not be suitable for situations with dynamism nature, e.g. as discussed earlier the ontology is only known at runtime for some semantic publish/subscribe systems. Secondly selecting from predefined reasoning levels cannot always provide the most suitable reasoning capability due to its coarse granularity.

An *algorithm-level* composition mechanism performs (automatic) reasoner composition at the reasoning algorithm level. Three existing mechanisms fall into this type: *dynamic rule generation*, *incremental loading of rules/triples* (ILR/ILT), and *rule dependency*.

In general the *dynamic rule generation* mechanism dynamically generates inference rules for the particular ontology to be reasoned according to pre-defined *rule patterns*. Since the ontology is considered in the dynamic rule generation, the generated rules are specially customized for the particular ontology and are less complex (in terms of joins) compared to static entailment rules, which hence can ensure efficient reasoning for the particular ontology [Meditskos and Bassiliades 2008b].

In [Meditskos and Bassiliades 2008b], O-DEVICE dynamically constructs ontology specific ABox reasoning rules by materializing predefined ABox rule templates with concepts/properties defined in the ontology. For example, a rule template

```
(defrule <rule-name>
  (object (is-a <p-domain>) (name ?obj1)
    (<p> $? ?obj2 &: (transitive ?obj1 ?obj2 <p>) $?))
=> (bind $?v1 (send ?obj1 get-<p>))
   (bind $?v2 (send ?obj2 get-<p>))
   (send ?obj1 put-<p> (union$ $?v1 $?v2)))
```

is defined to handle transitive properties. The <p> and the <p-domain> are (meta) variables that will be replaced with each occurrence of a transitive property and its given domain as drawn from the ontology to be reasoned.

Another work in [Meditskos and Bassiliades 2008a] proposes the DLE reasoning framework in which TBox classification is handled by a complete DL classifier and ABox reasoning is delegated to dynamically generated ABox entailment rules by grounding the T-triples (i.e. concept related triples) in rules with queries to the classified TBox. The generated ABox entailment rules are then ontology-specific taking into consideration the ontology to be reasoned. For example, the rule rdfp14a is transformed into a meta-rule as

hasValue(var(*r*), var(*y*)), onProperty(var(*r*), var(*p*)), <*x p y*>T→ <*x* type *r*>T

where hasValue(var(*r*), var(*y*)), onProperty(var(*r*), var(*p*)) are two queries to the previously reasoned Tbox. Multiple rules will be generated for the answers retrieved by these two queries. Any absence of a terminological axiom in the ontology will cause no answers to the corresponding TBox queries, which leads to the failure to generate such a rule instance. For example, if there is no hasValue construct in the ontology of the above example, then no such rules will be generated. Therefore this mechanism can make sure that the generated ABox rule set consists of only needed rules. In addition dynamically generated rules have fewer conditions, reducing the joining complexity of rules. The latest work in [Meditskos and Bassiliades 2010] apply this onto an OWL 2 RL reasoner, DLEJena.

A similar approach is also employed in μOR [Ali and Kiefer 2009, Ali 2010], a mobile OWL reasoner for ambient intelligence devices. It employs a dynamic rule generation process that searches the ontology to be reasoned for OWL constructs and then dynamically generates entailment rules according to rule patterns pre-defined for the found OWL constructs. For instance if a triple (*s* rdfs:subClassOf *o*) is detected in the ontology, a rule will be generated to infer that every instance of *s* is also an instance of *o*, i.e.

(*?t* rdf:type *s*) → (*?t* rdf:type *o*).

In general the *dynamic rule generation* mechanisms take into account the ontology to be reasoned in the rule generation process and hence compared to static entailment rules dynamic rules generated this way can be simpler and more specific for the particular ontology. Experiments show that they do improve the memory efficiency of OWL reasoning [Meditskos and Bassiliades 2008a, Meditskos and Bassiliades 2008b, Ali and Kiefer 2009, Ali 2010].

The *incremental loading of rules/triples* (ILR/ILT) is another algorithm level reasoner composition mechanism [Meditskos and Bassiliades 2008b]. ILR and ILT are designed to respectively reduce the amount of rules and triples in the reasoning engine in the same time. ILR separates the ABox reasoning rules into ten pre-defined subsets (i.e. transitive, symmetric, subproperty, inverse, equivalent, functional, inverse functional, universal quantifier, existential and classification) and they are evaluated one after another in a circular manner until no more rules are fired. However application of TBox reasoning rules is static without pre-analysis of their applicability for the ontology. ILT partitions the

ontology into segments of a pre-defined size and incrementally loads them into the reasoner for reasoning.

Experiments show that ILT and ILR improve the memory efficiency [Meditskos and Bassiliades 2008b], enabling larger ontology to be processed within a given size of memory. However ILT can only efficiently handle ontology segments of size 4K-6K triples or 20K triples (by default). In addition, although a complicated decision process is designed to use a parameter *p* to automatically determine which of the two predefined sizes (i.e. 5K or 20K) to use, this process requires complicated a priori analysis of the class hierarchy and the complexity of rule subsets. Furthermore, pre-defined weights need to be specified for dynamic rules in the a priori analysis.

A third algorithm level reasoner composition mechanism is the *study of dependencies among OWL inference rules* to avoid unnecessary rule evaluation in linear rule evaluation paradigms [Wu et al 2008]. Dependencies are used to decide at runtime if a rule should be evaluated: a rule is evaluated in round *n* only when in the round *n*-1 there is at least one new triple generated for at least one of the predicates contained in the rule. Experiments show it reduces the number of fired rules and the total inference time. However one drawback of this approach is there is a high chance that the generation of at least one new triple in round *n*-1 of a rule does not guarantee the fire of this rule in the round *n* therefore leading to still unnecessary memory to be wastage and processing.

Limitations of existing reasoner composition approaches

The algorithm level reasoner composition mechanisms perform automatic reasoner composition, and furthermore do improve the time/memory efficiency for their implementing reasoners. However they still have some potential limitations.

For the dynamic rule generation mechanism, three limitations are identified. A first limitation is that since dynamic rules are small and very specific about the particular ontology, the number of dynamic rules may increase dramatically with the size of TBox, which may to some extent reduce the memory benefit gained by having a smaller and more specific dynamic rule set. While on the other hand, unlike dynamic rules since static entailment rules are independent of the ontology to be reasoned, the number of static entailment rules can keep unchanged for different ontologies. A second limitation is the dynamic rule generation mechanism lacks applicability in TBox reasoning. In fact none of

the three reasoners adopting this mechanism applies it to TBox reasoning. O-DEIVCE only applies dynamic rule generation to ABox reasoning while TBox rules still relies on static entailment rules (without application of any composition mechanisms) [Meditskos and Bassiliades 2008b]. DLEJena uses a full-fledged DL reasoner for TBox reasoning and use dynamic rule generation for ABox only. Similarly dynamic rules generated in μOR are only for processing extensional knowledge. As a matter of fact, no dynamic rules are credited for computing class hierarchy, and hence TBox reasoning is not handled. For example, even a simple class hierarchy, e.g. *a* rdfs:subClassOf *b*, *b* rdfs:subClassOf *c* → *a* rdfs:subClassOf *c*, cannot be calculated by μOR. Thirdly, dynamic rules generation is based on rule patterns *pre-defined* by reasoner experts for a *specific* OWL subset. Therefore different rule patterns need to be generated once a different semantic or rule set is used, which requires careful manual analysis from reasoner experts. This will greatly limit its application in situations with high dynamism, e.g. domain-specific semantics are changing at runtime. Furthermore μOR hardcodes the rule patterns into the algorithm, restricting its flexibility to changes and extensibility.

In terms of the ILR/ILT mechanism, some limitations are also found. Firstly, ILR only operates on ABox rules and all TBox rules are loaded without any composition. Therefore unused TBox rules are still loaded and applied to the ontology, resulting in a waste of memory. Secondly, experiments show ILT will only have benefit if each ontology partition contains either 4K-6K triples or 20K triples [Meditskos and Bassiliades 2008b]. These beneficial partition sizes are much larger than that of many commonly used ontologies, e.g. wine (1.8K triples), food (0.9K triples) pizza (1.8K triples). Therefore it may not be suitable for the targeted context for the automatic reasoner composition research in this thesis, namely resource-constrained devices where the size of ontology may be (much) smaller than the beneficial sizes. Thirdly, although a decision process is designed to help determine which of the two beneficial partition sizes is better for the given ontology, the requirement of a priori analysis to the rule set and the ontology largely limits the application of this approach in a dynamic environment.

The drawback of the approach of studying rule dependencies is obvious: it is designed for a very specific linear rule evaluation implementation and therefore lacks general applicability on other reasoners.

To summarise, the discussion of the composability of different types of reasoners indicates that rule-based reasoners have better potential for composition and hence they show more

suitability than the other reasoner types for the automatic reasoner composition research. The discussion on the merits and limitations of existing reasoner composition algorithms reveals some features that the reasoner composition research can consider bringing in, as listed below.

Firstly, an automatic composition mechanism would be more appropriate than a static one for situations with dynamic nature which is targeted by the reasoner composition research of this thesis.

Secondly, existing automatic reasoner composition algorithms still require some a prior manual analysis of the rule set or the semantics they are going compose on, therefore manual re-generation of rule patterns, manual re-grouping of rules, or manual re-assignment of weight values to dynamic rules is required for them in order to handle a different semantics or rule set. Such prior manual analysis may cause problems for some applications such as context-aware system or semantic sensor network systems where application semantics may alter at runtime. Therefore an automatic composition algorithm independent of semantics is another important direction for this research.

Thirdly, existing automatic reasoner composition approaches can only compose ABox reasoning. Hence to compose on both ABox and TBox reasoning could be another challenge for this research.

Finally, the above described algorithm-level composition approaches compose at the rules level or ontology level (e.g. to generate simpler rules, to load only a subset of rules, to load ontology incrementally) and their reasoning algorithms, e.g. RETE or resolution, are still uncomposed, therefore leaving an additional opportunity for this research.

2.3.4 Resource-Constrained OWL Reasoners

While very few resource-constrained reasoners use reasoner composition approaches, this section however discusses all previous research on resource-constrained OWL reasoners including MiRE4OWL, μOR, Pocket KRHyper, Bossam, and the work done in [Gu et al 2007, Seitz et al 2010]. Two major goals motivate this discussion. First, given the resource-constrained environment as the context of the automatic reasoner composition research conducted in this thesis, how existing resource-constrained OWL reasoners optimize or compose themselves needs to be examined. The second goal is to assist the identification of a particular reasoner type for carrying out automatic reasoner composition research for the *resource-constrained environment*. Although rule-based reasoners are identified to have

better potential than the other reasoner types in terms of composability, still rule-entailment reasoners and resolution-based reasoners have totally different reasoning algorithms which may require total different reasoner composition mechanisms to be designed. Therefore a discussion of the merits and drawbacks of both existing resource-constrained rule-entailment reasoners and resource-constrained resolution-based reasoners would be beneficial.

MiRE4OWL [Kim et al 2010] is a resource-constrained rule-entailment OWL reasoner developed using C++ for the pocket PC platform (PPC). It is constructed based on the MiRE resource-constrained forward chaining rule engine [Choi et al 2008]. Two mechanisms are adopted to reduce the memory usage of the RETE engine. One is to restrict the number of facts of the same type and the other is to use a primary key to detect duplication of facts and to use an update key to specify the operation to take for duplications. These mechanisms are useful for keeping a light-weight and up-to-date fact base with continuously incoming facts. However there is no evidence that its RETE implementation is optimized and therefore it is likely that inefficient production joins may occur, if the rules are not tuned by rule experts.

µOR [Ali and Kiefer 2009, Ali 2010] is a resolution-based OWL-Lite reasoner for ambient intelligent devices (J2ME CDC compliance). As already discussed in section 2.3.3, it implements a dynamic rule generation mechanism that can automatically generate and compose a set of ABox inference rules for the given ontology according to pre-defined rule patterns. Dynamic rules are small and specific. However as already discussed, the drawbacks of this approach are obvious: (1) the size of rule set could increase rapidly with the size of the TBox, (2) rule patterns are pre-defined and hardcoded, limiting the flexibility to apply this approach on another semantics or rule set, especially in a dynamic environment when the semantics or the rule set is changing, and (3) this approach is only applied to ABox reasoning.

Bossam [Jang and Sohn 2004] is a forward-chaining OWL reasoner designed for desktop applications however its core engine is compatible to the J2ME CDC platform. However no evidence show Bossam has any implemented optimization to reduce the resource consumption for the resource-constrained environment.

In the work [Gu et al 2007] the authors present a framework supporting ontology processing and reasoning on mobile devices. It is built on CLDC 1.1 and MIDP 2.0. A forward-chaining rule engine is integrated in this framework to process both user-defined rules and OWL entailment rules. Context information is stored in a local context repository. A light weight

RDQL query engine is implemented to answer conjunctive queries. However no evidence show which particular algorithm is used for the forward-chaining engine and also no evidence show that any optimizations are used in the rule engine to reduce the memory consumption.

The work [Seitz et al 2010] presents a Digital Product Memory for storing product information and controlling product environment. OWL 2 RL is used to describe product information and the CLIPS engine [CLIPS] is used to perform forward-chaining rules matching. An ontology specific translation approach is used to translate between rules and OWL (refer to the section 2.3.1.1 for ontology specific translation). This work is implemented on a Crossbow Imote2 module (with 32bit PXA271 XScale CPU 624MHz, 32MB SDRAM and 32MB flash) using C#. However no optimizations are reported to reduce the resource consumption of the reasoner.

Some mobile DL tableaux reasoners also exist. The mTableaux [Steller and Krishnaswamy 2008] is a resource-constrained DL tableaux OWL reasoner. Three optimisation strategies are implemented to reduce its memory usage including (1) selective application of consistency rules, (2) skipping disjunctions, and (3) establishing pathways of individuals and disjunctions which if applied would lead to potential clashes, and associating weight values to these elements such that the most likely disjunctions are applied first. Experiments on a PPC show mTableaux uses less time and memory than Pellet and Racer. Pocket KRHyper [Sinner and Kleemann 2005, Kleemann 2006, Kleemann and Sinner 2006] is a DL reasoner based on hyper tableau calculus. However it does not directly handle OWL ontologies.

In summary, the state of the art review indicates that although some optimizations are applied to existing resource-constrained OWL reasoners, e.g. MiRE4OWL, in order to have better time/memory performance, very few of them use the automatic reasoner composition mechanisms as described in the previous section, and therefore their reasoning capabilities and reasoning algorithms are still static, further motivating the need for the development and application of automatic reasoner composition approaches for resource-constrained OWL reasoning.

In terms of the second goal as to choose between rule-entailment reasoners and resolution-based reasoners to support the reasoner composition research for resource-constrained environment, both reasoner types show their merits and drawbacks. In general fully pre-computing and materialising reasoning results in the RETE algorithm (of rule-entailment

reasoners) show better potential to efficiently handle data which are frequently accessed, efficient to store, and expensive to calculate at runtime, while backward chaining enables reasoning to be performed at runtime on-demand, which gives it more flexibility in handling changes (adding/deleting facts). Although reasoning is required for RETE to handle each change, the caching of intermediate results in RETE enables the reasoning required for changes to be performed *incrementally*, which is still efficient enough for resource-constrained devices. Furthermore, unlike rule-entailment reasoners that fully pre-compute and materialize reasoning results, enabling very fast and memory efficient query answering at runtime, the on-demand reasoning nature of resolution gives it less scope to reuse reasoning results and reasoning needs to be performed at runtime, which may require more runtime processing and power. Although pre-computation and materialization can be applied for resolution-based reasoners to enhance the runtime query answering performance, however, this also has the following limitations for the resource-constrained environment where processing power, memory, and power are restricted: (1) the fully pre-computation and materialization of reasoning results may require much more effort than RETE since the goal-directed feature of resolution requires to enumerate and to test a large number of possible goals; (2) memory- and time- expensive materialization maintenance algorithms need to be implemented for changes [Staudt et al 1996, Volz et al 2005], which on one hand may consume a lot more memory, processing, and power and on the other hand will greatly reduce its flexibility in handling changes; (3) the "answer space" of resolution is sometimes too large to materialize in resource-constrained devices.

The above discussion provided the motivation for the selection by the author of rule-entailment reasoners as the basis upon which the automatic reasoner composition research for resource-constrained environment would be conducted.

2.4 *Summary*

This chapter discusses the background knowledge (section 2.2) and related work (section 2.3) of this research.

Two parts are included in the background knowledge: OWL and its sublanguages (including both standard and non-standard OWL sublanguages for OWL 1 and sublanguages for OWL 2) and a detailed description of RETE algorithm and its optimizations.

The related work of this thesis consists of four parts. A first related work is a survey of state of the art OWL reasoners (section 2.3.1), in which a categorization of OWL reasoners was drawn and a set of reasoner characteristics was distilled. Five reasoner categories were

obtained which are the DL-tableaux reasoners, the rule-entailment reasoners, the resolution-based reasoners, the hybrid reasoners and the miscellaneous reasoners. This survey offers a basis for the following research to carry out. A second related work (section 2.3.2) is then presented. Five selected types of applications were surveyed: a semantic publish/subscribe systems type, a semantic context-aware systems type, a clinical, medical and bioinformatics type and a semantic sensor network systems type and finally a software engineering systems type. This survey investigated the requirements of particular applications/application types and the interplay between these requirements and the selected reasoner, facilitating the research of an automatic reasoner selection process. A third related work is the discussion of the composability for different reasoner categories derived in section a survey of semantic applications, a discussion of reasoner composability for each type of reasoner identified above. The rule-entailment reasoners and the resolution-based reasoners were found to have the best potential to be composed. Then a fourth related work discusses about the state of the art resource-constrained reasoners. Based on this discussion, the suitability for the rule-entailment reasoners and the resolution-based reasoners to be applied in resource-constrained environment is discussed. It is found out that rule-entailment reasoners have better suitability for running in a resource-constrained device. Considering its high composability and suitability to run in resource-constrained environment, rule-entailment reasoners are selected as the most appropriate type of reasoner on which the resource-constrained reasoner composition research will be carried out.

The work undertaken in the related work section achieves objective 1, which is the state of the art survey objective of this thesis. In the next chapter, the design of the reasoner composition approach, including two novel automatic reasoner composition algorithms, is described in detail.

Chapter 3

COROR: A *CO*mposable *R*ule-entailment *O*wl *R*easoner for Resource-Constrained Environments

3.1 *Introduction*

The discussion of the composability for different reasoner types in section 2.3.3 shows rule-based reasoners has better potential for composition and show better suitability for carrying out the composition research. Further discussion of the pros and cons of existing reasoner composition algorithms has identified some aspects that this reasoner composition research could further explore. Firstly, the newly designed composition mechanism needs to be an automatic process in order to perform composition for applications with some dynamism. Secondly, although the existing automatic composition mechanisms perform well, they still require some a priori manual analysis of the specific semantics or the specific rule set to be composed, limiting the flexibility to apply them to a different semantics or rule set at runtime, which is sometimes the case for applications with dynamism. Hence it would be preferable that the newly designed composition mechanism can be "fully automatic" without any a priori manual analysis. Thirdly, the newly designed composition mechanism needs to operate for both ABox reasoning and TBox reasoning. Finally, investigating the composability of the reasoning algorithms themselves (rather than how rules are loaded, how ontologies are partitioned, how dynamic rules are generated, and so on) has never been studied by previous work.

A later discussion in section 2.3.4 on existing resource-constrained OWL reasoners indicates that only μOR has adopted a *dynamic rule generation* mechanism to perform reasoner composition. Although some optimizations are adopted for some of the other resource-

constrained reasoners, their reasoning algorithms remain uncomposed, showing opportunities for the application of reasoner composition research for resource-constrained reasoning. A further investigation on whether rule-entailment reasoners or resolution-based reasoners are more suitable for resource-constrained environment indicates that rule-entailment reasoners need less runtime processing and therefore power consumption for query answering due to the pre-computation and materialization of reasoning results. Although pre-computation and materialization can also be adopted for resolution-based reasoners to reduce the processing required for answering queries at runtime, the requirement for materializing large "answer spaces", the requirement of very complicated materialization maintenance algorithms, and the large degradation of its flexibility to handle changes, makes it is quite expensive from a resources perspective to apply materialization to resolution-based reasoners in resource-constrained environment.

Based on the discussions on the composability of different types of reasoners and their suitability for resource-constrained environments, rule-entailment reasoners are then identified as the best suitable type of reasoners to carry out the reasoner composition research for resource-constrained environments.

The discussion of different existing reasoner composition algorithms in section 2.3.3 has already pointed the author to some directions where the reasoner composition research can be developed. Considering that different ontologies may vary greatly in expressivity, e.g. the Pizza ontology used in this thesis has a DL expressivity $\mathcal{ALCF}(\mathcal{D})$ while the Wine ontology used in this thesis has a DL expressivity $\mathcal{SHIO}\mathcal{N}(\mathcal{D})$, the required entailment rules may be different for them. Hence a natural avenue for achieving composition is to automatically compose a required set of entailment rules according to the expressivity of the particular ontology to be reasoned, and unnecessary entailment rules are removed so intuitively less processing and memory are required. This approach may require an analysis of the expressivity of the ontology. A straight-forward way to conduct this would be to examine the OWL constructs contained by the ontology, and this can also be easily and automatically achieved by existing ontology frameworks such as Jena or OWLAPI [Horridge and Bechhofer 2011]. Therefore if a condition of an entailment rule does not match any of the included OWL constructs the rule is never fired for this ontology and thus the entailment rule needs not to be loaded.

The result entailment rule set composed in this way can then tightly fit the required expressivity of the ontology. Similar to all the automatic composition mechanisms discussed

in the related work, this approach composes only the entailment rule set, and still the RETE algorithm itself would be noncomposable. However, the discussion of RETE algorithms in the background knowledge section in Chapter 2 shows that RETE caches all intermediate results in the RETE network, which is the major source of the memory cost of RETE. To compose the RETE network such that less memory and processing (e.g. match/join operations) are required would be a second avenue for achieving composition.

As discussed earlier in section 2.2.2.3.1, the quality of rules, in particular join sequences, has been found to be the key factor that determines the structure of a RETE network and inappropriate join sequences can cause a dramatic waste of memory and processing time. Thus many join sequence reordering optimizations have been proposed in order to have a better join sequence so that memory- and time- efficient RETE network can be obtained. It is shown in the discussion in the background knowledge that automatic application of these join sequence optimizations taking characteristics of the fact base to be processed into consideration can generate better join sequences for the particular fact base [Scales 1986, Ishida 1994]. Such automatic application of optimizations pointed a good avenue for the RETE composition research to follow. However such existing approaches either require a priori execution of the entire fact base to determine its characteristics [Ishida 1994] or are designed for a specific production system only [Scales 1986], which are not suitable for resource-constrained environment targeted by this thesis. Hence a new approach needs to be designed that requires no a priori analysis to collect required information for optimizing join sequences.

By looking into the RETE network, two interesting observations are found: (1) facts are matched against alpha network and stored in alpha memories; hence some analyses of the alpha memories can obtain some information on the fact base which may be used for join sequence optimization; (2) the join sequence does not affect the construction of the alpha network, i.e. regardless of the join sequence the alpha network remains the same as long as the conditions keep the same. Following these observations, a possible way to perform automatic composition then could be to have an interrupted RETE network construction process: to construct only the alpha network first, then to match the ontology against the alpha network only so some information relevant to join sequence optimization can be collected at this stage, e.g. the number of matched facts for each condition, the joining selectivity and so on, then according to these information join sequences are optimized for the particular ontology, then an optimized and customized beta network is constructed, finally the fact matching resumes joining in the beta network and finishing the rest RETE

cycles.

Following the above two intuitive avenues, this chapter presents COROR, a COmposable Rule-Entailment Owl Reasoner for resource-constrained environments incorporating two novel reasoner composition algorithms arising from the two avenues presented above. They automatically compose the reasoner at different levels according to the particular ontology to be reasoned such that less memory and time is required by the reasoner, facilitating the execution of OWL reasoning in highly resource-constrained environments. Section 3.2 gives an overview of COROR and how the composition algorithms are applied. Then detail on the two composition algorithms are presented and discussed in section 3.4. Finally section 3.5 studies the possibility to extend this research to support OWL 2 from a design perspective.

3.2 *An Overview*

COROR is a *composable* reasoner because of the use of two reasoner composition algorithms, i.e. a *selective rule loading algorithm* and a *two-phase RETE algorithm*, which are designed to compose the reasoner at different levels (rule set level and inside RETE algorithm) according to the semantics of the particular ontology to be reasoned. The use of composition algorithms facilitates the construction of a customized reasoner for the particular ontology and application such that redundant reasoning capabilities and processes are avoided by minimizing the resources required by the reasoner.

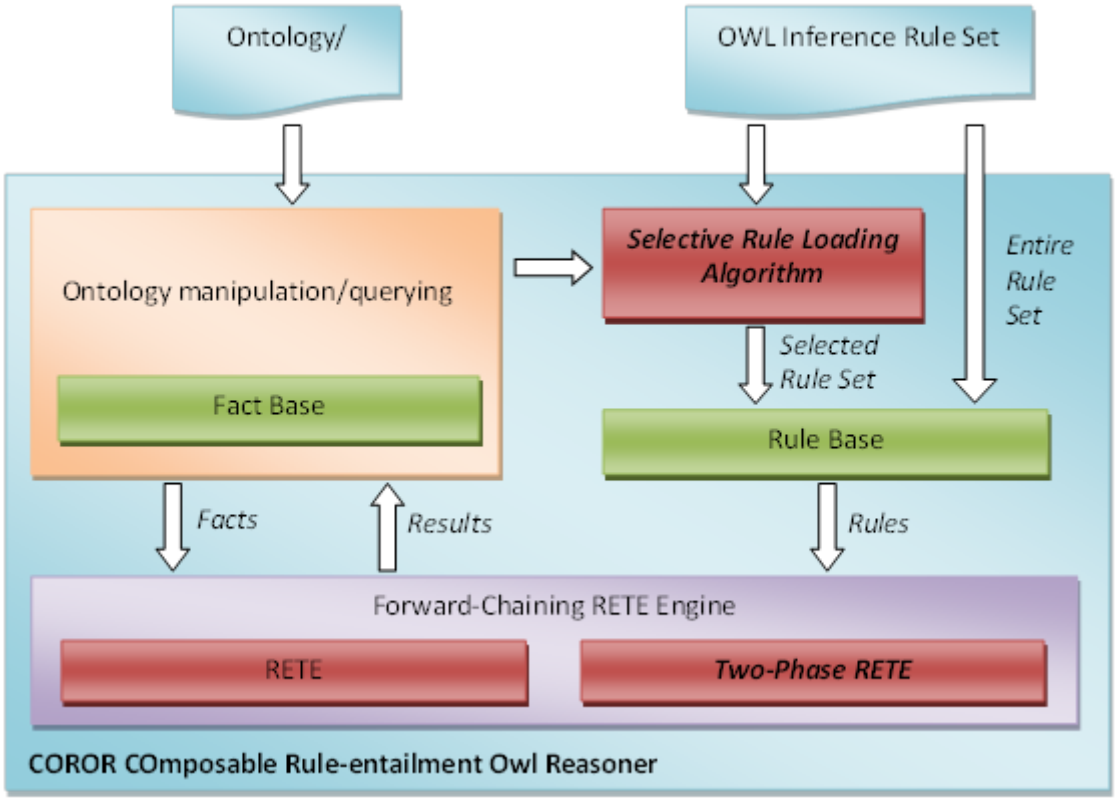


Figure 3-1: An Overview of COROR

Figure 3-1 illustrates how different components in COROR interplay with each other. The composition algorithms are marked using a bold and italic font. The OWL Ontology is loaded and stored in a fact base where it can be manipulated and queried. Rule loading can follow two modes, either directly loading the full rule set or selectively loading through the use of the *selective rule loading* algorithm. Reasoning is performed by a forward-chaining RETE engine that can be configured to use either the standard RETE approach, or a novel *two-phase RETE* approach that will be presented in this thesis. Results of the reasoning are fed back into the fact base and the engine halts when an inference closure is reached (no new entailments can be generated).

3.3 *The pD* Semantics*

The pD* semantics is chosen as the semantics for COROR for three reasons. Firstly, as discussed earlier in section 2.2.1.2, it is composed of a definitive set of entailment rules, which then shows perfect suitability for COROR since it is a rule-entailment reasoner. Secondly pD* has tractable entailment reasoning. It has PTIME entailment complexity when variables are not used in the target ontology and NPTIME entailment complexity when variables are used in the target ontology. However its minor extension, pD*sv, does

not enjoy such low complexity. A tractable entailment problem of pD* shows its great suitability to be applied in resource-constrained environments given the low resource availability. Thirdly, although some OWL-DL constructs are missing, such as cardinality constructs, some (in)equality constructs, Boolean combination constructs, and *oneOf*, it still preserves a substantial subset of OWL-DL constructs (as indicated in Table 2-1). Given the resource-constrained context where this research will be applied, any ontology will be generally much less complex than OWL-DL. It can be envisaged that the pD* generally has sufficient expressivity and semantics to model the KBs in resource-constrained domains to an acceptable degree. As a matter of fact, pD* is also used by some state of the art desktop commercialized rule-entailment reasoners (such as OWLIM and BaseVISor), which demonstrates its sufficiency in terms of semantics.

3.4 *Composition Algorithms*

Two novel reasoner composition algorithms are used in COROR both at the rule set level and inside the RETE algorithm, according to the semantics of the particular ontology to be reasoned. Customized rule set and RETE network are therefore constructed for the ontology. These composition algorithms form the core design of COROR. In this section the detail of the two composition algorithms are described.

3.4.1 Selective Rule Loading Algorithm

The design of the *selective rule loading* algorithm is originated from the first thought presented in the introduction of this chapter as to compose at the entailment rule set level. In general the *selective rule loading* algorithm dimensions a selected entailment rule set by estimating the usage of each entailment rule for reasoning the ontology to be reasoned, using predefined *rule-construct dependencies* that describe the containment of OWL constructs in rules. If a construct is referred to in the left hand side (l.h.s.) of a rule, then the rule is said to *depend* on the construct and the construct is said to be a *premise* of the rule in the dependency relationship. Constructs in the right hand side (r.h.s) of a rule are said to depend on the rule, and the constructs are *consequences* of the rule in the dependency relationship. For example, as shown in Appendix C, the construct rdfs:subPropertyOf is the premise of rule rdfp13c and owl:equivalentProperty is the consequence. Multiple premises and consequences may exist for a rule.

The rule-construct dependencies are used by the *selective rule loading* algorithm to decide if a rule should be loaded for reasoning a given ontology. A rule is loaded if there is possibility that it could be fired, and otherwise not loaded. One necessary condition of the rule's

(potential) firing is all its premises are included in the ontology (Note that for brevity a premise included by the ontology to be reasoned is termed as a *valid* premise and otherwise an *invalid* premise). In other words, if any of its premises are invalid (for the given ontology), the condition containing the invalid premise will not be met (as the construct is not included in the ontology) and therefore there is no possibility that this rule is fired. Therefore the *selective rule loading* algorithm regulates *a rule to be loaded if and only if all its premises are valid premises*.

However the firing of some loaded rules may produce consequences that are themselves premises for other unloaded rules, validating some previously invalid premises and causing the loading of unloaded rules. For example, the firing of the rdfp13c rule will add owl:equivalentProperty into the ontology, validating the premises of rule rdfp13a and rdfp13b and therefore causing them to be loaded into the engine (supposing the ontology originally contains rdfs:subPropertyOf but not owl:equivalentProperty). Therefore the *selective rule loading* algorithm also regulates that *if a rule is loaded then its consequence can be used to validate premises of other unloaded rules*.

The dependency relationships described above, especially these chain-like dependency relationships, where some premises of one rule are consequences of some other rules, can be better illustrated as graphs, termed in this research as rule-construct dependency graph. Figure 3-2and Figure 3-3 respectively illustrate the rule-construct dependency graphs for the D* entailment rules and the P entailment rules (refer to [ter Horst 2005b] for a full set of D* and P entailment rules). Note that D* entailment rules and P entailment rules together comprise the pD* entailment rules used in COROR. All pD* entailment rules in Jena rule format can be found in Appendix C.

Rules and OWL constructs are represented as nodes in the rule-construct graphs, respectively represented as regular and rounded rectangles (Figure 3-2 and Figure 3-3). Each rule/construct corresponds to one node. Different colours are used to mark different types of nodes, e.g. purple for D* constructs nodes, orange for P constructs nodes and blue for rules nodes. Dependencies are represented as both dashed arrows (links between *core rules* and *core constructs*) and solid arrows (links between *expressivity constructs* and *candidate rules*). An arrow always points from a premise (if any) to a rule or points from a rule to its consequence (if any). Note that the differences between these two types of arrows and the definitions of *core rules*, *core constructs*, *expressivity constructs* and *condution rules*

are discussed in detail in the following paragraphs. A bi-directional arrow linking between a construct node and a rule node means the construct is both a consequence and a premise of the rule, i.e. the construct is contained either in the both the l.h.s. and the r.h.s. of the rule, e.g. rdfs:subClassOf is contained in both the l.h.s. and r.h.s. of the rule rdfs11.

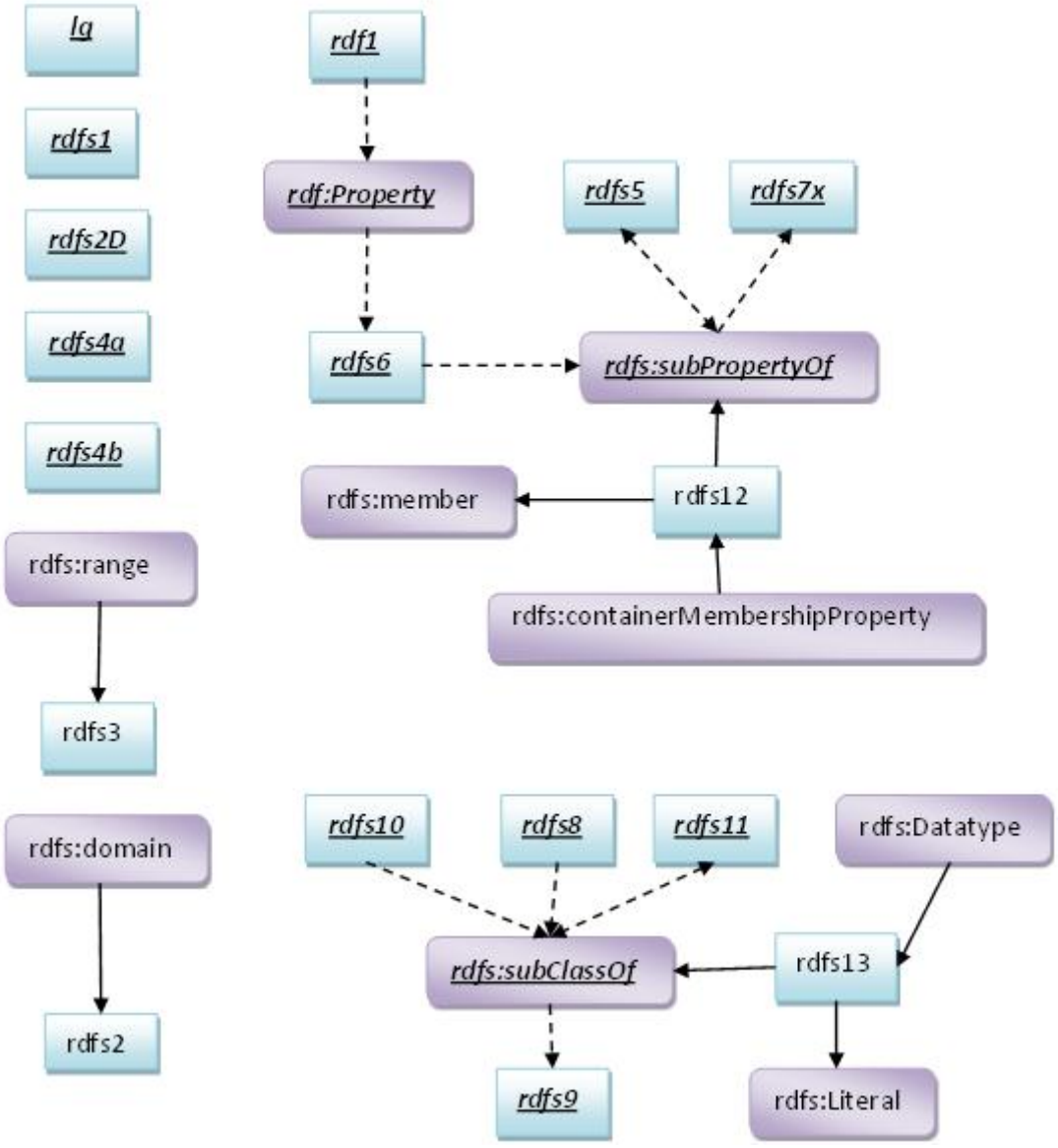


Figure 3-2: Rule-construct dependency graphs (D* entailment rules)

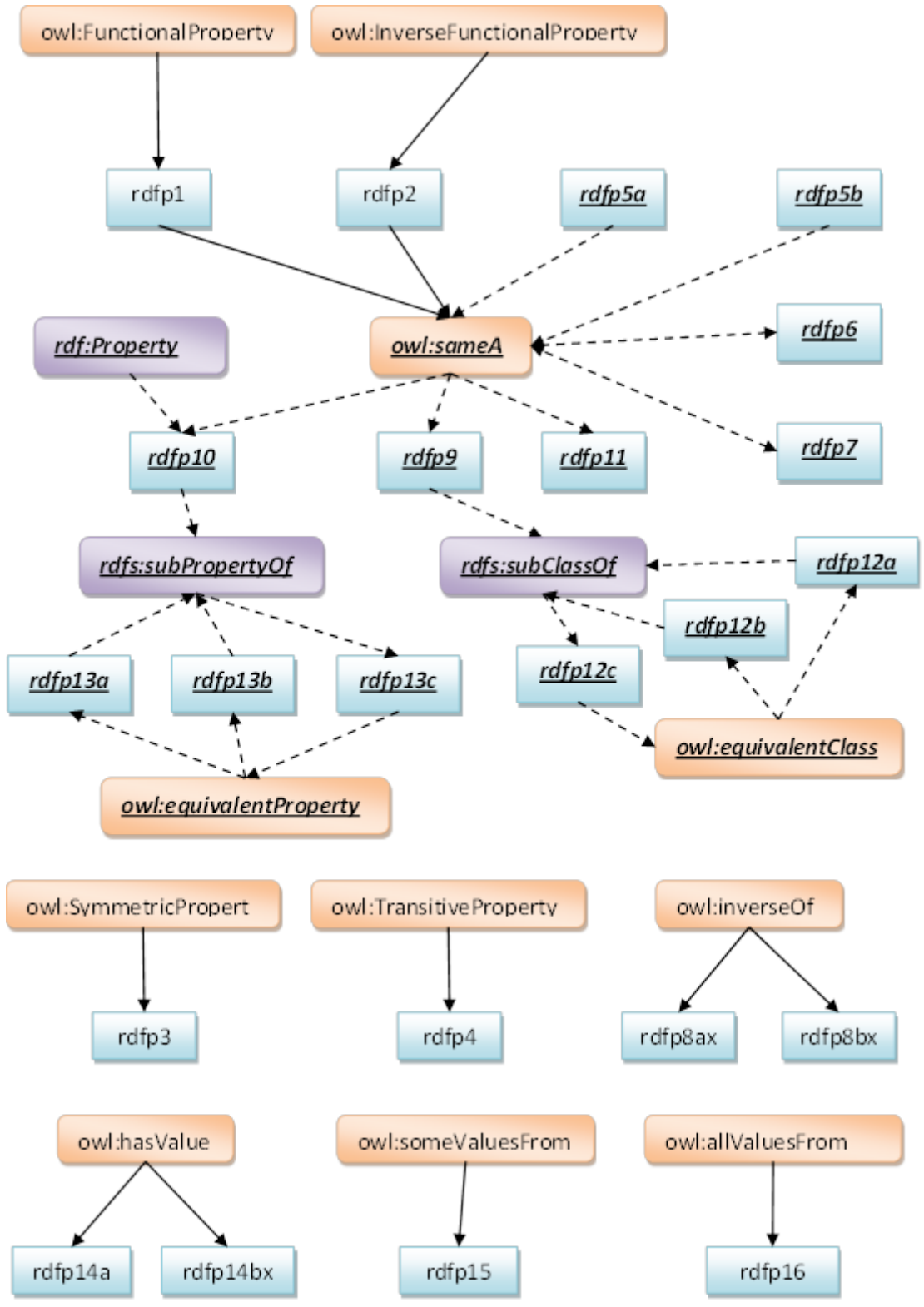


Figure 3-3: Rule-construct dependency graphs (P entailment rules)

Two types of constructs are omitted from the rule-construct graphs, i.e. *basic constructs* and

auxiliary constructs. In the *selective rule loading* algorithm three OWL constructs are considered as *basic constructs*, i.e. rdf:type, rdfs:Class and rdfs:Resource, since they are included in almost all (practical) ontologies. *Basic construct premises* are then deemed as valid by default for all ontologies, and for brevity they are omitted from the rule-construct graphs. One *auxiliary construct*, i.e. owl:onProperty, is identified as they always appear with OWL restrictions. Similarly they are also omitted from the rule-construct graphs for brevity.

The omission of basic constructs and auxiliary constructs from Figure 3-2 and Figure 3-3 may cause the absence of premises or consequences for some rules on the graph. For example the rule rdfs8 and rdfs10 has no displayed premises since all their premises are basic constructs. Some other rules also have no premises (and therefore no displayed premises), including lg, gl, rdf1, rdf2-D, rdfs1, rdfs4a, rdfs4b, rdfp5a, rdfp5b, as the conditions in their l.h.s. are wildcard conditions (i.e. conditions such as (*?x ?y ?z*) that match all kinds of facts). Similar reasons also apply to the rules without displayed consequence, such as rdfs7x (wildcard r.h.s., no consequences), rdfs3 (basic construct consequences). However omitting these constructs from the diagrams is only for the purpose of having a clearer rule-construct dependency graph and does not mean omitting their premises or consequences. The premises and consequences for these rules remain unchanged.

According to the rule loading regulation presented above, rules with no displayed premises are loaded automatically for all ontology as either they have no premises or their premises are basic constructs. However their automatic loading causes the validation of some other premises, loading some other rules. For example, the automatic loading of rdf1 validates rdf:Property, and then causes the automatic loading of rdfs6, which again validates rdfs:subPropertyOf, causing the loading of rdfs5 and rdfs7x. The cascaded validating and loading paradigm causes a set of premises to be always validated for all ontologies automatically and therefore a set of rules to be always loaded for all ontologies automatically. They are respectively termed as *core constructs* (including basic constructs) and *core rules*. A full list of core constructs include rdf:type, rdfs:Class, rdfs:Resource, rdf:Property, rdfs:subPropertyOf, rdfs:subClassOf, owl:sameAs, owl:equivalentClass, owl:equivalentProperty. A full list of core rules include lg, gl, rdf1, rdf2-D, rdfs1, rdfs4a, rdfs4b, rdfp5a, rdfp5b, rdfs6, rdfs8, rdfs9, rdfs10, rdfp5a,

rdfp5b, rdfp6, rdfp7, rdfp9, rdfp10, rdfp11, rdfp12a, rdfp12b, rdfp12c, rdfp13a, rdfp13b, and rdfp13c. Core constructs and core rules are represented in the diagrams of graphs using an italic, bold and underlined font. Dependencies in between core rules and core constructs (both premises and consequences) are represented using dashed arrows. The other rules are then *candidate rules* for *selective rule loading*, and the other constructs are termed as *expressivity constructs*. Although a lot of core rules are automatically loaded there are still many candidate rules for selection, including rdfs2, rdfs3, rdfs12, rdfs13, rdfp1, rdfp2, rdfp3, rdfp4, rdfp8ax, rdfp8bx, rdfp14a, rdfp14bx, rdfp15, and rdfp16. They are represented in the diagrams of the graphs using normal fonts. Dependencies between candidate rules and expressivity constructs are represented as solid arrows.

Hence in general, the *selective rule loading* algorithm using the rule-dependency graphs then turns out to be identifying valid premises according to the given ontology and then follow the dependencies searching for rules to be loaded. For example, given an ontology in triple format as

ex:Car rdf:type rdfs:Class.

ex:Engine rdf:type rdfs:Class.

ex:Car rdfs:subClassOf ex:hasEngineRestriction.

ex:hasEngineRestriction rdf:type owl:Restriction.

ex:hasEngineRestriction owl:onProperty ex:hasComponent.

ex:hasEngineRestriction owl:someValuesFrom ex:Engine.

Valid premises include rdf:type, rdfs:Class, rdfs:subClassOf, owl:Restriction, owl:onProperty and owl:someValuesFrom, among which rdf:type, rdfs:Class and rdfs:subClassOf are core constructs, owl:Restriction and owl:onProperty are auxiliary constructs, owl:someValuesFrom is expressivity construct. Apart from core rules, following the dependency relationships starting from the valid premise owl:someValuesFrom causes the loading of the rule rdfp15. The other rules are not loaded as their premises are not valid for this ontology.

The regulations made by this algorithm ensure unloaded rules that are *definitely* not fired for the ontology (as an invalid premise means the construct is not included in the ontology) and loaded rules are *possibly* fired. Therefore these regulations may construct a super set of the required rule set for the ontology: not all loaded rules are guaranteed to fire.

Discussion of the *Selective Rule Loading* Algorithm:

The *selective rule loading* algorithm automatically performs composition at rule set level. It takes advantage of the intrinsic composability of the pD* entailment rule set, i.e. entailment rules are fine-granulated in terms of OWL semantics and can be freely loaded and unloaded according to the semantics of the ontology. What makes it different from previous work is the use of rule-construct dependencies to determine if a rule is necessary to be loaded for a particular ontology and the capability to work on both ABox rules and TBox rules..

There are some merits about this approach. Firstly, it is independent of reasoning algorithms and therefore it can be applied to RETE or resolution. Secondly, although pre-analysis is required to construct the rule-construct dependency graphs limiting its dynamic application to a different rule set, this approach is independent of the rule set. Thirdly, a lot of core rules are loaded by default for all ontologies, however there are still 14 candidate rules for selective rule loading and the candidate rules are in general more complex than core rules (e.g. they have more conditions). Therefore the loading of all these candidate rules in an uncomposed reasoning approach is likely to cost a lot of time and memory consumption, necessitating a selective loading of them into the engine.

Some drawbacks of the *selective rule loading* algorithms are also identified. A first drawback is the selected rule set may cause re-execution of the rule selection process once the ontology is changed at runtime and new constructs are introduced, thereby causing a potential waste in resources. A second drawback is pre-analysis is required in order to generate rule-construct dependencies, which, as discussed in section 2.3.3, limits the dynamic application of this composition approach to an environment with changing semantics or rule sets.

3.4.2 Two-Phase RETE Algorithm

The design of the *two-phase RETE* algorithm follows the second thought as presented in the introduction of this chapter. It performs composition inside the RETE algorithm. Rather than fully constructing the RETE network and then match facts along the RETE network as normal RETE does, it uses a novel *interrupted RETE construction mechanism* (Figure 3-4):

firstly a shared alpha network is built; then the RETE network construction is interrupted by an initial fact matching against the constructed alpha network, with matched facts stored in alpha memory; some information about the ontology is then collected; according to the collected information a customized beta network is then built using optimization heuristics for the particular ontology; and finally the fact matching resumes as normal RETE algorithm. The initial fact matching breaks the RETE network construction into two phases: the alpha network construction and information collection phase (the *first phase* for short) and the beta network optimization and construction phase (the *second phase* for short), inspiring the name *two-phase RETE* algorithm. This algorithm only changes the way a RETE network is constructed and the following RETE cycles are performed in a same way as normal RETE algorithm as described in section 2.2.2. This section presents the two phases separately in detail.

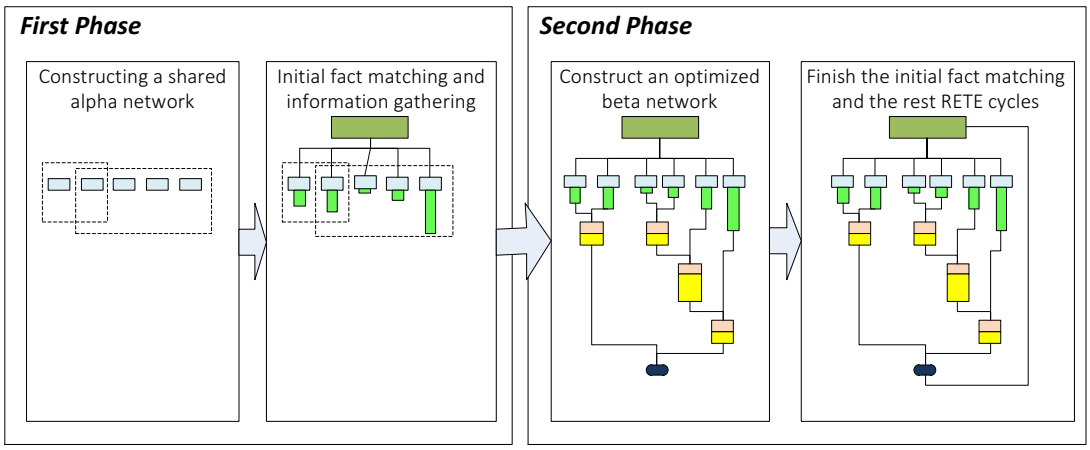


Figure 3-4: Flow of the Two-Phase RETE Algorithm

3.4.2.1 First Phase

In the first phase the shared alpha network is first built according to a *node sharing* mechanism in order to reduce the size of alpha network. Since the same condition may appear in several rules, e.g. (*?x* rdf:type *?y*) appears in rdfs9, rdfp14bx and rdfp16, an alpha node sharing mechanism is adopted enabling common condition elements to be shared among rules. This mechanism constructs only one rather than n alpha nodes for a common condition shared by n rules, therefore the size of alpha memory and the number of match operations for this condition are reduced to 1/n. Note that the adoption of a node sharing mechanism adopted is not a novel idea and some other RETE engines, e.g. Drools[11],

[11] http://www.jboss.org/drools

implement some similar alpha node sharing optimizations to reduce the alpha network. Then an initial fact matching is performed against the shared alpha network after its construction and matched facts are stored in the corresponding alpha memory. This enables analyses to be performed on the matched facts collecting some information about the ontology that is originally hard to collect before RETE execution, e.g. the number of facts matching to a particular condition and the join selectivity factor between two joining conditions, and so on. As already discussed in the section 2.2.2.3 this information can help construct an optimized and customized beta network for the particular ontology (in particular join sequence). In this research the number of matched triples for each condition is gathered. As will be described in the later sections, it is used to optimize the join sequence of conditions.

There are two merits to collect information at this stage. Firstly, as mentioned earlier some information that is hard to collect before RETE execution can be easily collected at this stage. The second is the initial fact matching facilitates the collection of some information, e.g. the number of facts matching a particular condition, without traversing the ontology, saving some time.

An example rule set and ontology are used to exemplify the algorithm. Given an entailment rule set *R* with two rules rdfp14bx and rdfp15 (in Jena rule format) as

[rdfs9: (*?v* rdfs:subClassOf *?w*), (*?u* rdf:type *?v*) → (*?u* rdf:type *?w*)]

[rdfp15: (*?v* owl:someValuesFrom *?w*), (*?v* owl:onProperty *?p*), (*?u ?p ?x*), (*?x* rdf:type *?w*) → (*?u* rdf:type *?v*)]

and an ontology snippet *O* (in N-triple format) as

ex:Car rdf:type rdfs:Class. T1

ex:Car rdfs:subClassOf ex:Vehicle. T2

ex:Fiat rdfs:subClassOf ex:Car T3

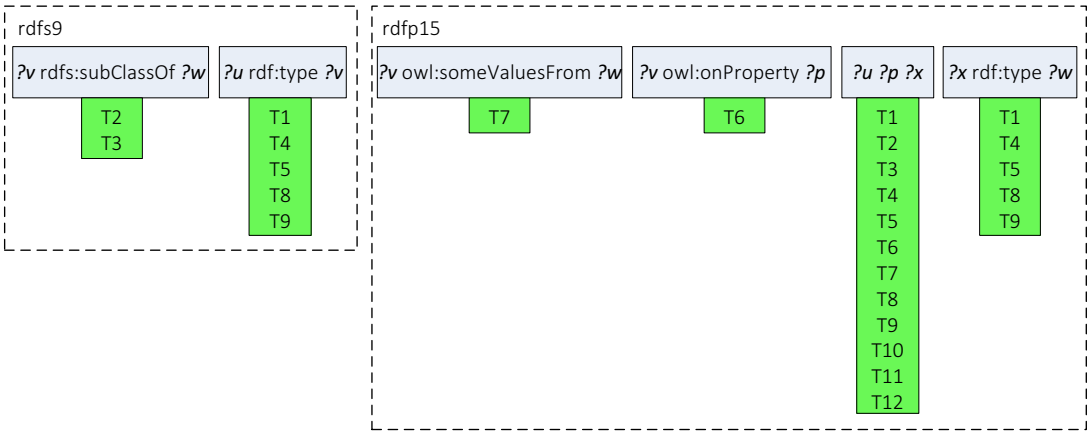
ex:Engine rdf:type rdfs:Class. T4

ex:hasEngineRestriction rdf:type owl:Restriction. T5

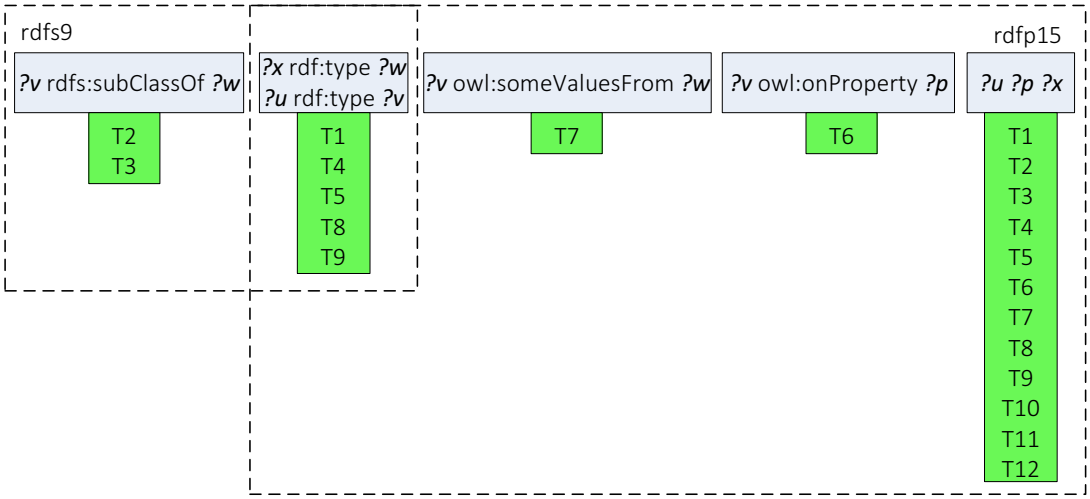
ex:hasEngineRestriction owl:onProperty ex:hasComponent.	T6
ex:hasEngineRestriction owl:someValuesFrom ex:Engine.	T7
ex:myCar rdf:type ex:Car.	T8
ex:azrTurbo rdf:type ex:Engine.	T9
ex:myCar ex:hasComponent ex:azrTurbo.	T10
ex:myCar ex:hasComponent ex:alcon	T11
ex:myCar ex:hasComponent ex:energyMX1	T12

ex:myCar rdf:type ex:hasEngineRestriction	I13
ex:myCar rdf:type ex:Vehicle	I14

whose TBox states any car needs to have a component as an engine and car is a subclass of vehicle. The ABox contains two individuals ex:myCar and ex:azrTurbo. Facts T1 to T10 are asserted facts, and the facts I13 and I14 are inferred facts that can be deduced from the inserted ontology according to the rule rdfs9 and rdfp15. Figure 3-5 illustrates the alpha network of the example rules after the initial fact matching. Both a non-shared and a shared alpha network are given to illustrate the differences. In Figure 3-5a a non-shared alpha network is constructed following normal RETE while in Figure 3-5b a shared alpha network is built following the *two-phase RETE* algorithm. The common node, i.e. (*?u* rdf:type *?v*), is shared between rdfs9 and rdfp15, leading to only one node constructed for it. Note that the condition sequence of rule rdfp15 has been changed ((*?u* rdf:type *?v*) is lifted to the front of the rule) in the shared alpha network for better illustration, and it does not affect its join sequence.



(a) A non-shared alpha network



(b) A shared alpha network

Figure 3-5: A shared alpha network v.s. a non-shared alpha network.

Matched facts for each condition are stored in the corresponding alpha memory after the initial matching. The number of matched facts for each condition is therefore collected for each condition. Results are shown in Table 3-1. The number of matched facts does not change for non-shared conditions, but the number of matched facts for the common condition is reduced to half as it is shared by both rules. This number will further reduce if more rules share this condition.

Table 3-1: Number of matched facts for each condition

Condition		Shared	Non-shared
rdfs9	(*?v* rdfs:subClassOf *?w*)	2	2
	(*?u* rdf:type *?v*)	2.5 (shared by both rules)	5
rdfp15	(*?v* owl:someValuesFrom *?w*)	1	1
	(*?v* owl:onProperty *?w*)	1	1
	(*?u ?p ?x*)	10	10
	(*?u* rdf:type *?v*)	2.5 (shared by both rules)	5

3.4.2.2 Second Phase

In general the second phase of this algorithm heuristically builds an optimized and customized beta network for the ontology, using the information collected at the first stage, and then matched facts (stored in alpha network) continue to populate and to join along the beta network firing rules. Two heuristics from the state of the art, i.e. the *most specific condition first* heuristic and the *pre-evaluation of join connectivity* heuristic, are used in this phase to optimize the join sequences (refer to section 2.2.2.3 for a detailed description of them). However the novelty here is the way they are applied. Firstly, they are applied taking the ontology to be reasoned into account, rather than being statically and directly applied considering only the rule set. Secondly unlike the previous automatic RETE optimization approaches that requires a priori execution [Scales 1986] in order to collect information to apply the join sequence optimizations, here the information collection and join sequence optimization are embedded into the RETE cycles and therefore the number of matched facts for each condition can be collected and used as its specificity (which was considered as a "mission impossible" by a previous work [Özacar et al 2007]).

As already discussed earlier in the background chapter, the *most specific condition first* heuristic, orders join sequences according to their specificity to avoid the long chain effect. In this research the number of matched facts for each condition is taken as an estimate of the specificity, which, according to section 2.2.2.3.1, is a straight forward way to determine specificity, but is hard for normal RETE as it cannot be known before execution. The more matched facts for a condition the less specific it is for the particular ontology. A corollary presents where the fewer matched facts a condition has, it is more specific. The join sequence is then reordered where conditions with more matched facts are moved later in the join sequence. This causes more discriminating joins to be performed first, thereby reducing the size of the beta network memory required and also the join operations to be performed.

For better illustration, the example given in section 3.4.2.1 is continued. The join sequence reordered after applying the most specific condition first heuristic should look like Figure 3-6.

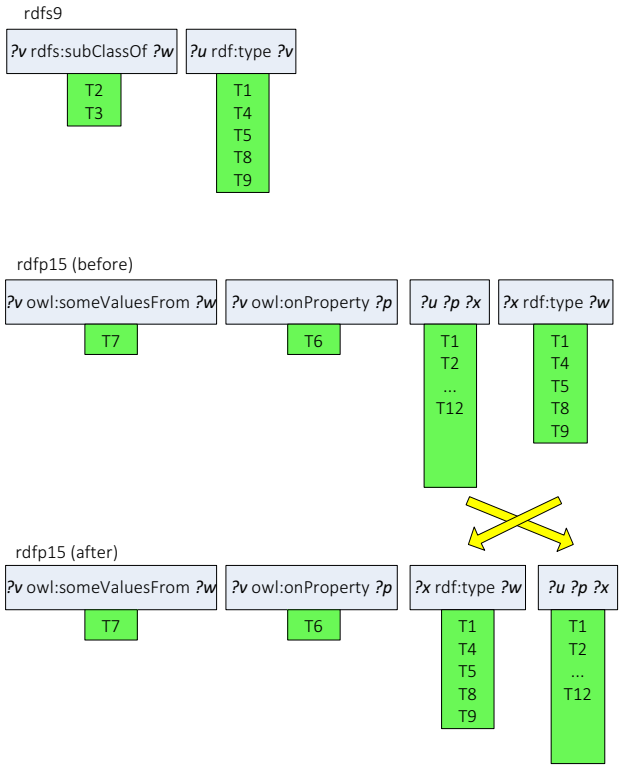


Figure 3-6: Join sequences after been reordered by the *most specific condition first* heuristic.

The pre-evaluation of the *join connectivity* heuristic is introduced after the *most specific condition first* heuristic as the second heuristic to ensure all joining conditions have variables in common so that Cartesian product joins are avoided. In brief it scans the entire join sequence from front to the end, and swaps the unconnected condition with the first connected condition behind it in the join sequence. As illustrated in Figure 3-7 C_1, C_2 … C_{n-1} are conditions having their connection checked (no matter if they are connected or not) and they form C_{pre}. C_n is found not connected to C_{pre}. Therefore this heuristic starts searching from C_{n+1} to the end for the *first* condition that is connected to C_{pre}, which is C_m in this case. Then C_m is then switched before C_n and the rest conditions (conditions behinds C_n) are moved one place toward the end of the join sequence (as if shown by the second joins sequence in Figure 3-7). As the join sequence has already been ordered by the most specific condition first heuristic, C_m is then the most specificity condition after C_n that connects to C_{pre}. If none is found to connect to C_{pre}, then leave C_n where it is, and continue to check the

next condition, e.g. C_{n+1} in this case. This tries to ensure the connectivity of the join sequence while avoids the damage to the join sequence where possible. As the rule rdfs9 and rdfp15 are already connected the application of this heuristic does not change the join sequence.

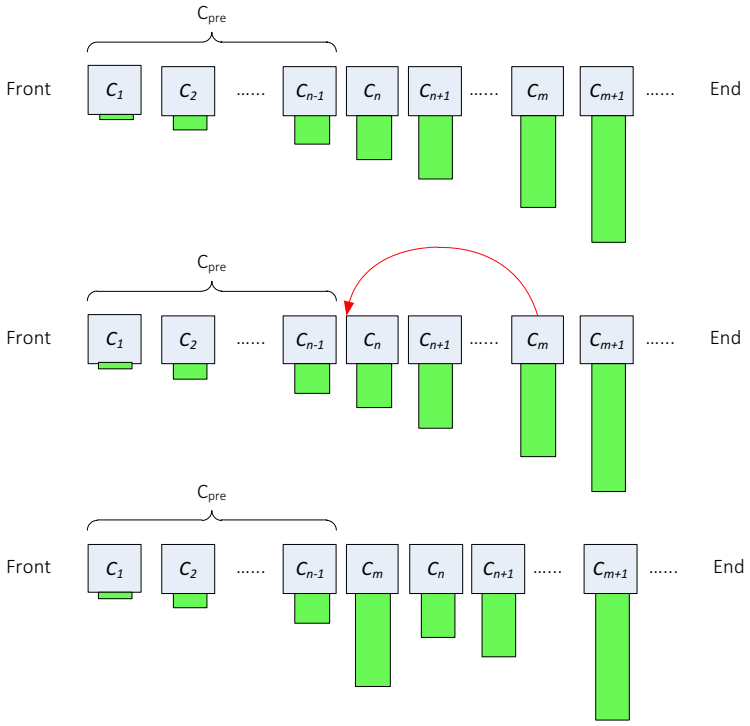
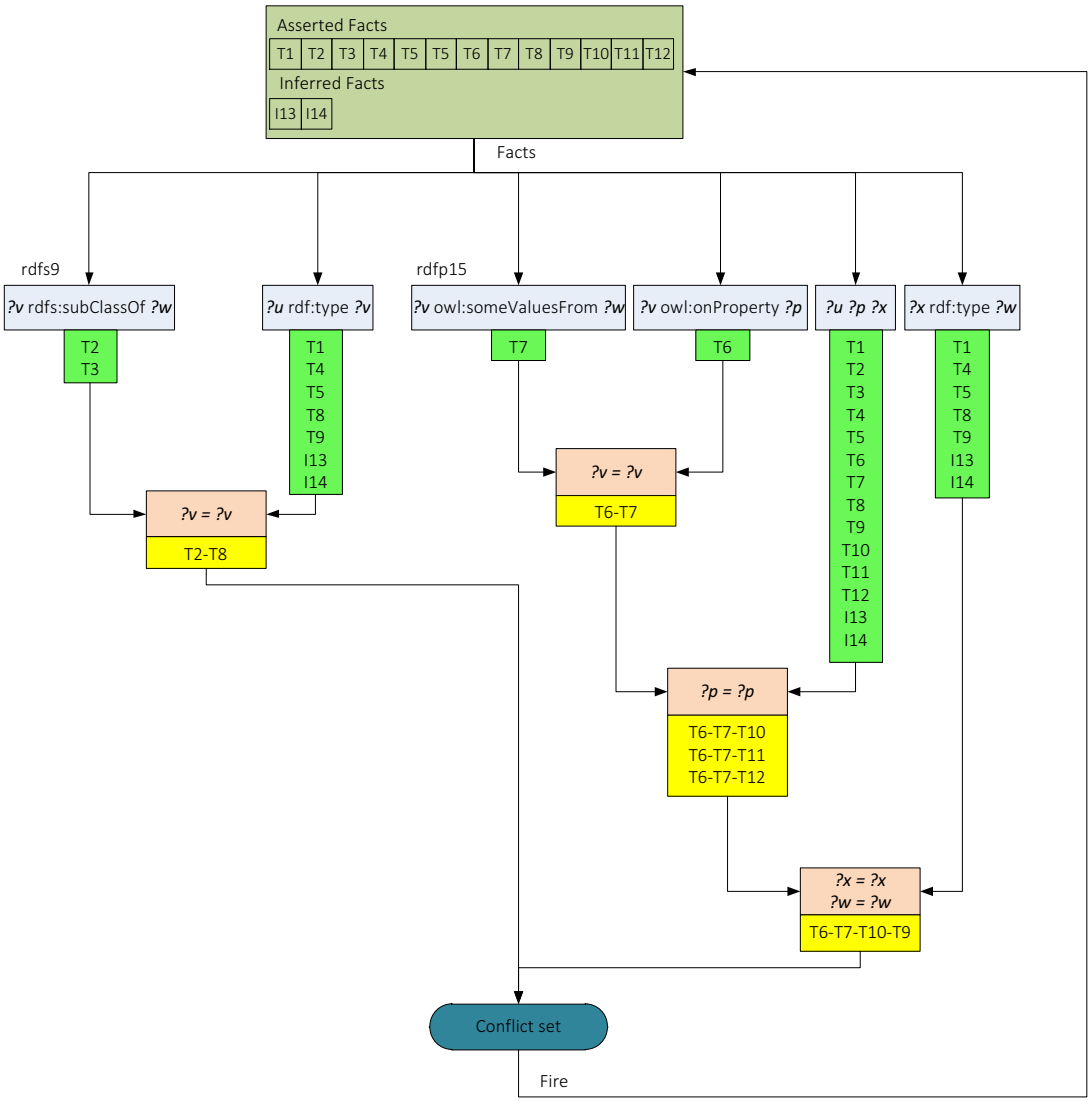
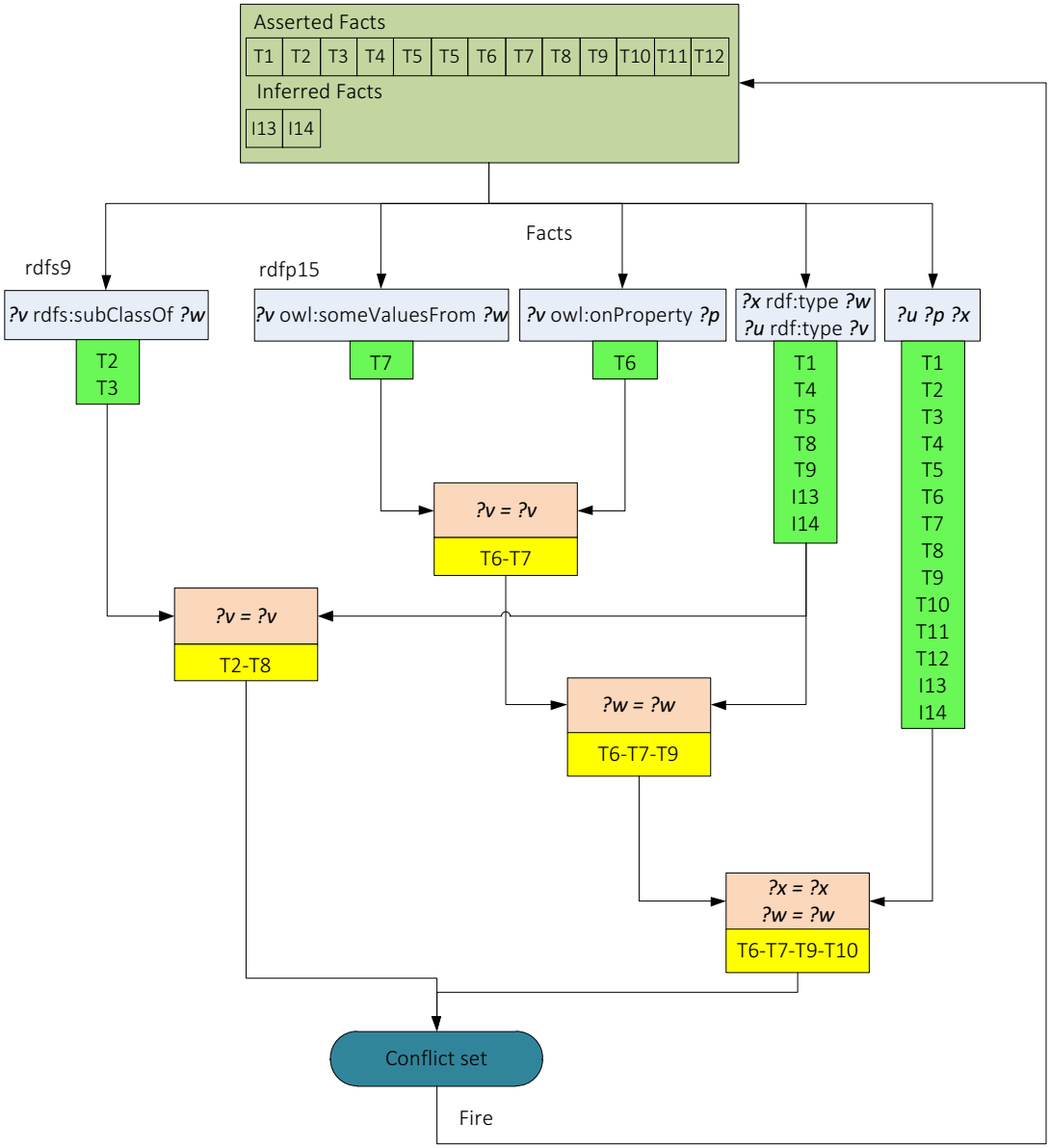


Figure 3-7: pre-evaluation of the join connectivity heuristic

Customized join sequences are fixed after both heuristics are applied and a customized beta network is then constructed. After the construction of the RETE network facts stored in alpha memory continue to pass down the beta network joining each other and firing rules as in normal RETE algorithm until no rules can be fired. Figure 3-8a and Figure 3-8b gives the RETE networks of both the original RETE algorithm and the *two-phase RETE* algorithm when no more rules are fired. Intermediate results generated by join operations are listed in the yellow box under the corresponding beta nodes.



(a) The RETE network constructed by the original RETE algorithm



(b) The RETE network constructed by the *two-phase RETE* algorithm.

Figure 3-8: RETE Network with facts after all RETE cycles.

As shown in the diagrams the *two-phase RETE* algorithm shares common alpha nodes, i.e. (?x rdf:type ?w) in the rule rdfp15 and the (?u rdf:type ?v) in rdfs9, therefore the memory is shared between them (as shown in Figure 3-8b). The customized join sequences enable less intermediate results to be generated. For example, only 4 intermediate results are generated in the network given in Figure 3-8b, however 6 intermediate results are generated in the original RETE network given in Figure 3-8a.

Discussion of the *Two-Phase RETE* Algorithm:

The *two-phase RETE* algorithm performs composition inside the RETE algorithm by constructing an optimized and customized RETE network for the particular OWL ontology. It is a fully automatic composition approach requiring no manual analysis or pre-analysis of either the rule set or the ontology and therefore can be applied to a different rule set or semantics without human intervention. Since composition is perfomed inside the RETE algorithm and the entire rule set is loaded, changes on the ontology can be reflected immediately in the reasoning without re-executing the entire composition. The two-phase RETE algorithm works on both ABox rules and TBox rules. Unlike the *selective rule loading* algorithm which can work on both RETE and resolution, the *two-phase RETE* algorithm is designed to work on RETE algorithm only. However it is clear that the functioning of the *two-phase RETE* algorithm does not rely on the particular rule set so it is semantic independent and can be applied to other semantics rather than pD*, e.g. OWL 2.

Two problems need to be further clarified. The first problem is whether the information collected in the first stage can be used to effectively optimize the RETE beta network. The second problem is the limitations of using the number of matched facts of a condition as its specificity to order join sequences.

In terms of the first problem, as the firing of rules may add inferred facts into the fact base and hence the RETE network, changing the number of matched facts for each condition, the number of matched facts collected in the first stage may not accurately represent the number of matched facts when RETE terminates. Therefore it appears that the number of matched facts collected at this stage is only accurate to construct an optimized RETE network for the first RETE cycle and this RETE network is not the optimal for the rest RETE cycles. However, it is also noticed from insights into the RETE network when reasoning over 19 ontologies (as described in the evaluation chapter) one by one that with most ontologies experimented upon, the majority of joins occur in the first RETE cycle: 15 of a total of 19 ontologies have an average of 75% joins performed in the first iteration (for the remaining 4 ontology this percentage is still above 50%). Furthermore an average of 83% inferred facts are generated in the first cycle. Hence it is appropriate to optimize the RETE network by applying heuristics based on information collected here.

For the second problem, it might not always be correct to deduce that a condition with 100 matched facts is more specific than a condition with 101 matched facts as these numbers are only collected from the initial match, therefore it is highly likely that the previous condition

(with 100 matched facts) may have more matched triples the later condition (with 101 matched facts) in the following RETE cycles. This can be partially solved by introducing more sophisticated mechanisms for specificity estimation, for example combining more types of information into specificity estimation such as the number of variables of condition elements, the cardinality of values to be joined and so on. They can all be gathered before or in the first phase. At the moment no other information is collected, but the approach taken is equally applicable and, as described later, the approach taken substantially reduces memory and reasoning time. As a matter of fact as other information can also be collected during the first phase enabling more sophisticated optimizations, e.g. to order join sequences according to join selectivity, to be applied to enable the construction of even more customized beta network.

3.4.3 Hybrid Algorithm

The above two composition algorithms compose at different levels: the *selective rule loading* algorithm composes at the rule set level while the *two-phase RETE* algorithm composes inside the RETE algorithm. Hence it is natural to think that if these two algorithms will complete each other. Based on this idea, the *hybrid* algorithm is designed by simply combining of the *selective rule loading* algorithm and the *two-phase RETE* algorithm: the *selective rule loading* algorithm is applied first constructing a selected rule set and then the *two-phase RETE* algorithm builds a customized and optimized RETE network using the selected rule set. The *hybrid* algorithm is inspired by the idea that unneeded rules are removed from the rule set on which the *two-phase RETE* algorithm is applied, therefore an even smaller RETE network is built reducing the memory and reasoning time.

Discussion of the *Hybrid* Algorithm:

The *hybrid* algorithm combines both algorithms and therefore it naturally combines the merits and drawbacks of both. The algorithm is not applicable to other reasoning algorithms as the *two-phase RETE* algorithm only applies to the RETE algorithm. The *hybrid* algorithm cannot be dynamically applied to a different rule set or semantics as pre-analysis is required for the *selective rule loading* algorithm. However the *hybrid* algorithm composes both at the rule set level and inside the RETE algorithm.

3.5 *Extending COROR to Support OWL 2 (Design Perspective)*

OWL 2 has been standardized and recommended in late 2009 by the W3C. As discussed earlier in the background chapter, OWL 2 RL is one of the OWL 2 sublanguages whose semantics are given as a set of entailment rules (the full OWL 2 RL rule set can be found in

[OWL 2 Profiles]). Therefore OWL 2 shows strong feasibility to be applied in rule-entailment reasoners. At a matter of fact OWL 2 is an extension of pD* semantics, as discussed in section 2.2.1.3. This section shows in an analytical way that although COROR is originally designed for the pD* subset of OWL-DL, it is by design extensible to OWL 2 without fundamental changes to the two composition algorithms, because of their semantics independent features.

As already discussed in the previous sections, the *two-phase RETE* algorithm concentrates on the construction of an optimized RETE network, which is independent of the semantics in use. Therefore it is applicable to OWL 2 without further change.

However in order to enable the *selective rule loading* algorithm to run on the OWL 2 rule set, the rule-construct dependency graphs need to be constructed for OWL 2. Figure 3-9 to Figure 3-14 list the rule-construct dependency graphs for OWL 2 RL entailment rule set, showing the applicability of the *selective rule loading* algorithm to OWL 2 RL without fundamentally changing the algorithm itself. Similarly core rules and core constructs are identified and emphasized in the same way as those in Figure 3-2 and Figure 3-3.

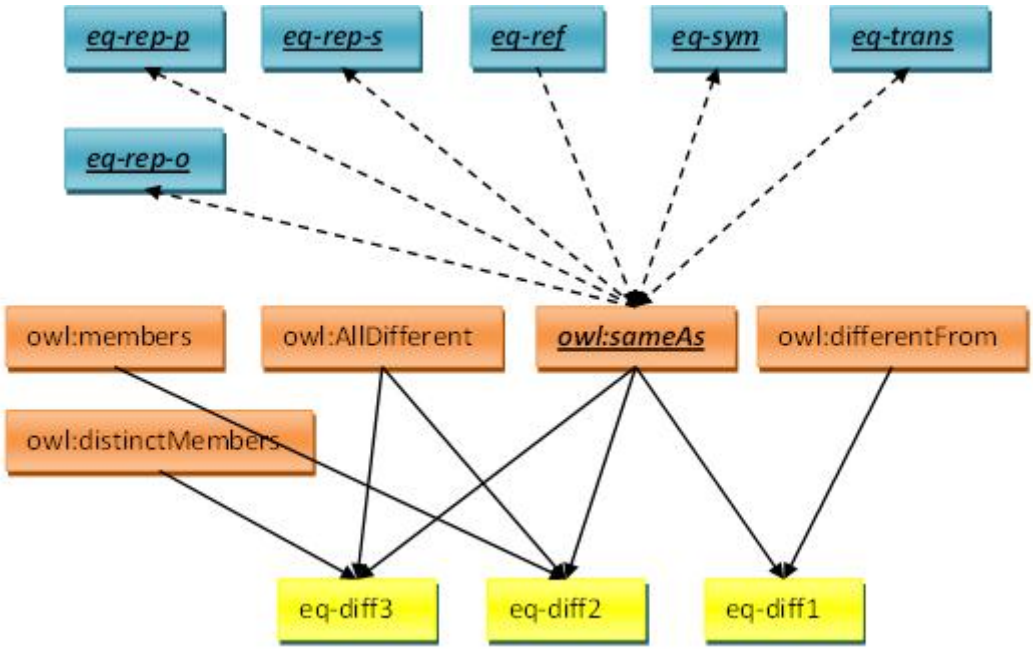


Figure 3-9: Rule-Construct dependency graph for OWL 2 RL entailments (semantics of equality).

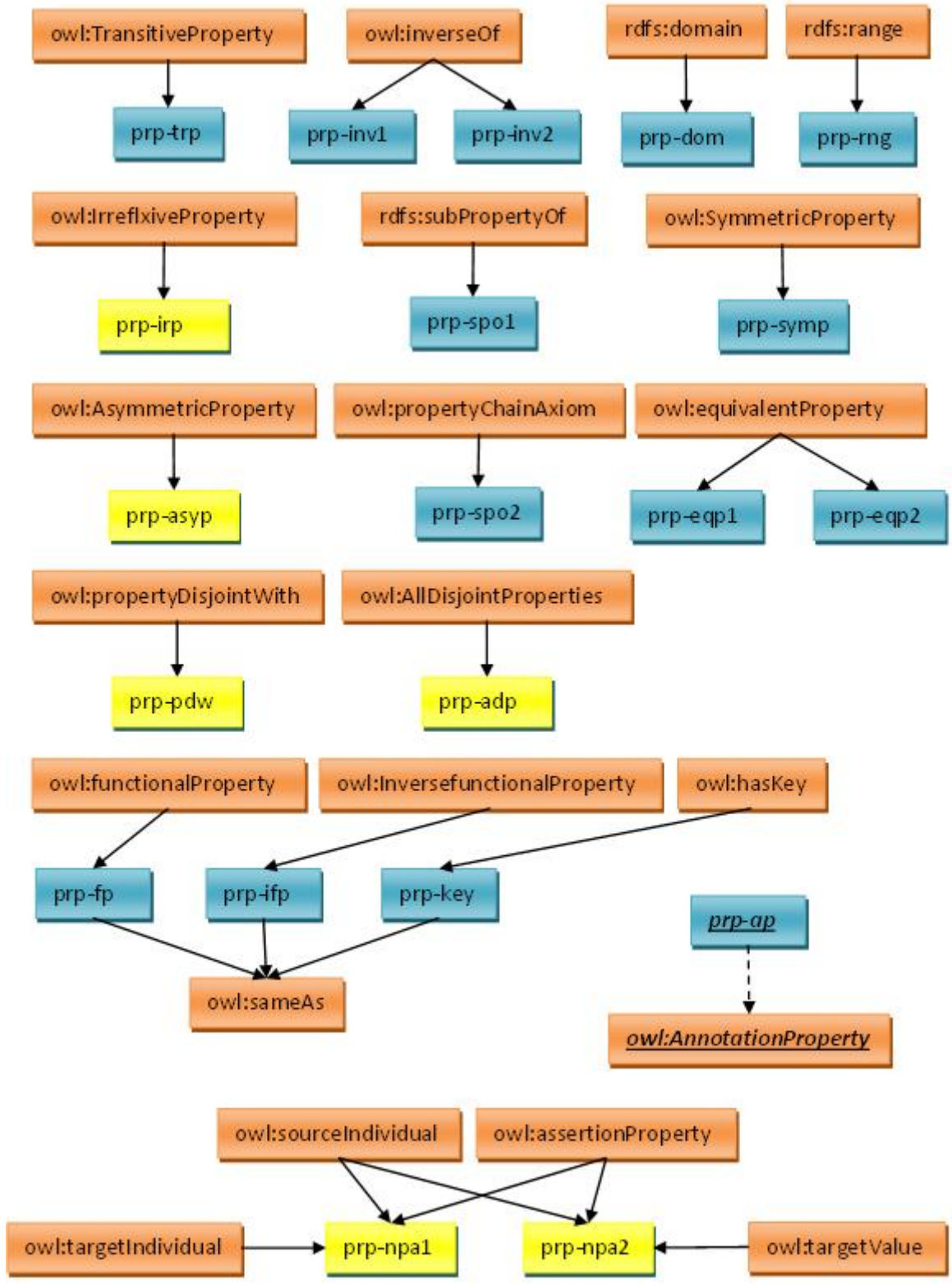


Figure 3-10: Rule-Construct dependency graph for OWL 2 RL entailments (Semantics of Axioms about Properties).

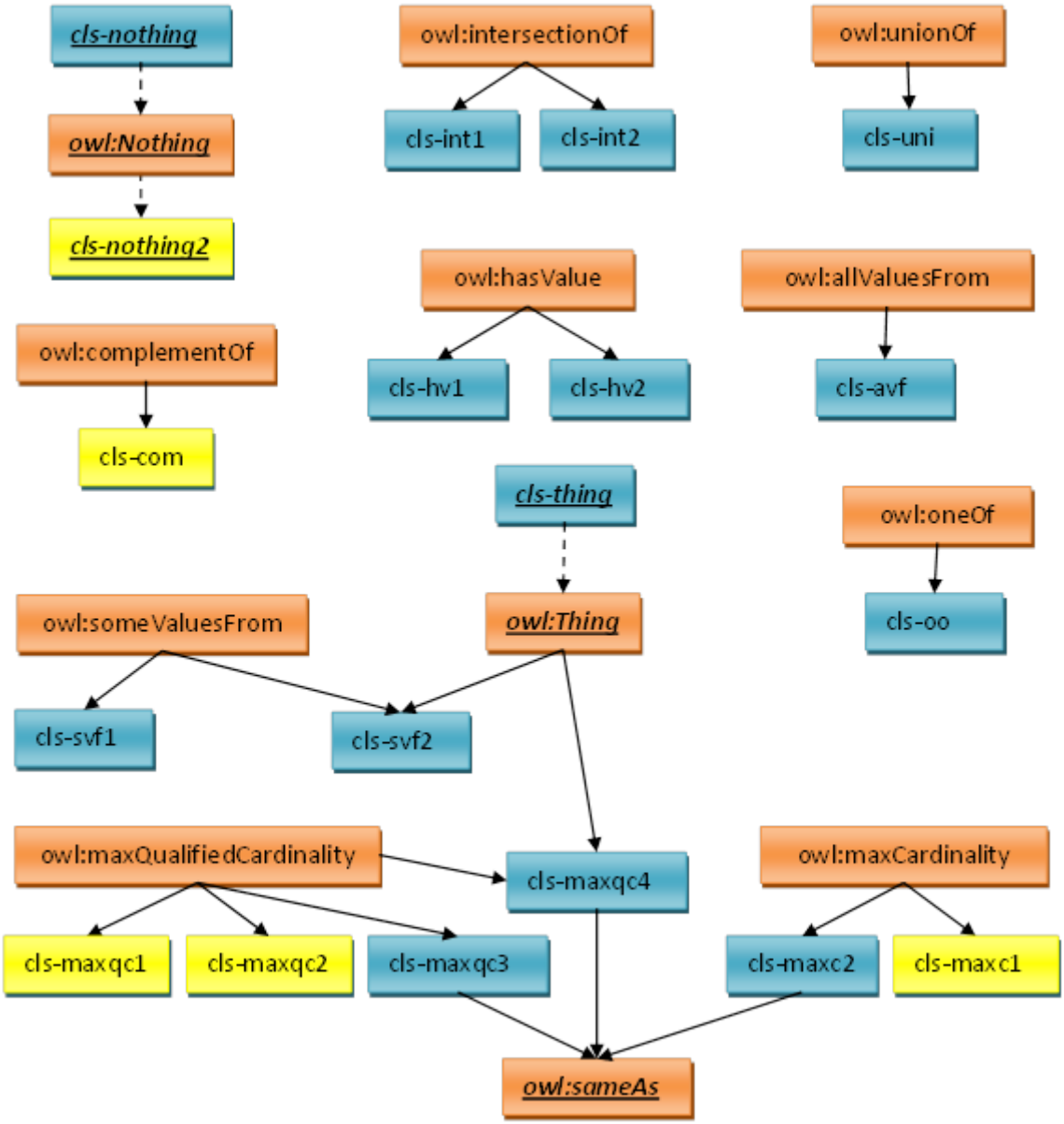


Figure 3-11: Rule-Construct dependency graph for OWL 2 RL entailments (Semantics of Classes).

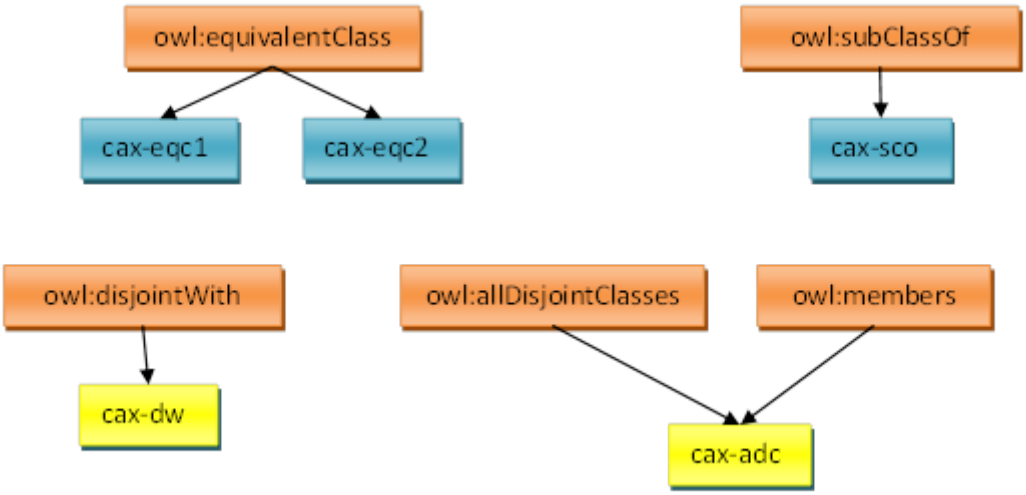


Figure 3-12: Rule-Construct dependency graph for OWL 2 RL entailments (Semantics of Class Axioms)

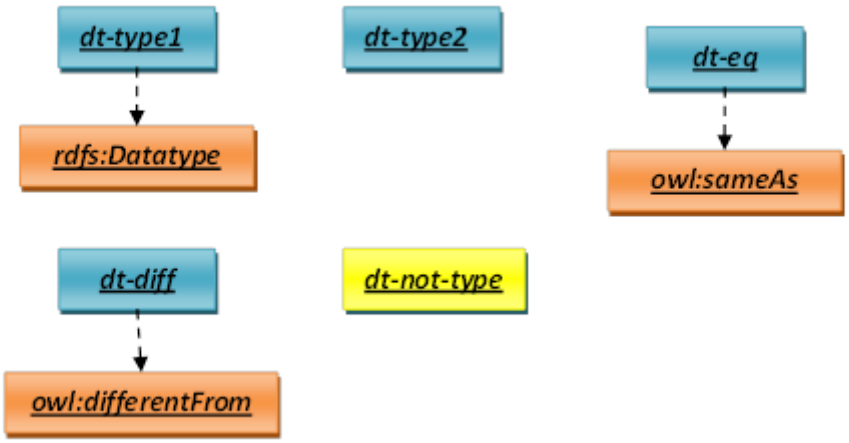


Figure 3-13: Rule-Construct dependency graph for OWL 2 RL entailments (Semantics of Datatypes)

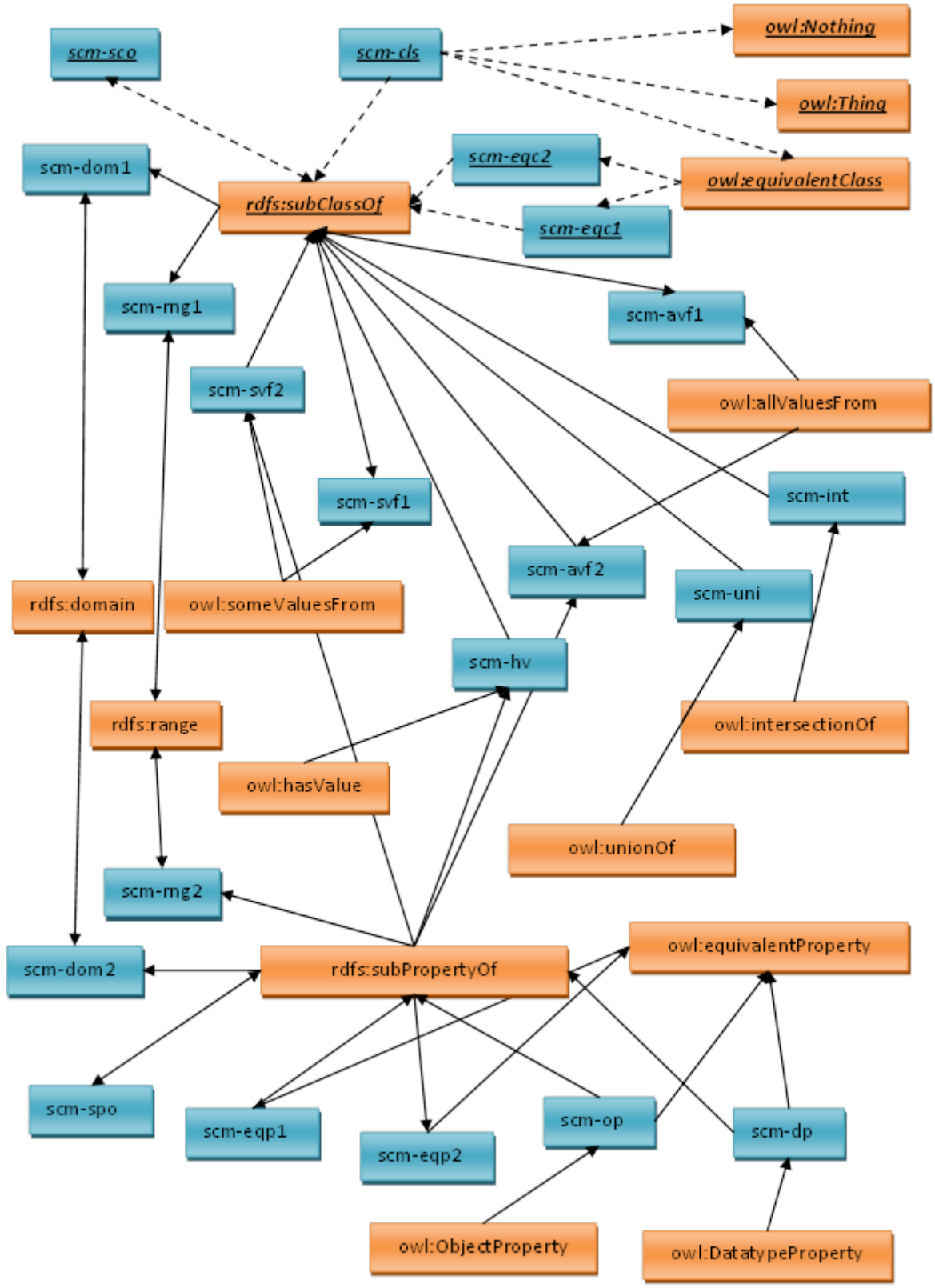


Figure 3-14: Rule-Construct dependency graph for OWL 2 RL entailments (Semantics of Schema Vocabulary)

As one might notice that in the OWL 2 rule-construct dependencies graphs a third type of node is used to represent consistency rules (yellow nodes). They are not included in either Figure 3-2 or Figure 3-3 since no consistency rules are modelled for the pD* semantics. Two OWL 2 constructs, namely owl:onProperty and owl:onClass, are identified as auxiliary constructs; rdf:type and owl:Class are considered as basic constructs. Therefore they do not appear in dependency graphs. Similarly core constructs and core rules are identified by the following dependencies. Core constructs include owl:sameAs, owl:AnnotationProperty, owl:Nothing, owl:Thing, rdfs:Datatype, owl:sameAs, owl:differentFrom, rdfs:subClassOf and owl:equivalentClass; core rules include eq-rep-p, eq-rep-s, eq-ref, eq-sym, eq-trans, eq-rep-o, prp-op, cls-nothing, cls-nothing2, cls-thing, dt-type1, dt-type2, dt-eg, dt-diff, dt-not-type, scm-sco, scm-cls, scm-eqc2 and scm-eqc1. Core rules, core constructs and their dependencies are emphasized in similar way to those in Figure 3-2 or Figure 3-3.

Some improvements can be performed to increase the accuracy of selecting rules using dependencies. For example, rather than constructing two premises (one for owl:someValuesFrom and the other for owl:Thing) for the rule cls-svf2, which can cause its loading even if both constructs are included in the ontology but not in the same triple, e.g. cannot match (*?x* owl:someValuesFrom owl:Thing). This can be solved using a complex premise that combines both owl:someValuesFrom and owl:Thing to limit the existence of two constructs in one triple. Another example is that owl:maxCardinality is limited to 0 and 1 in some rules of OWL 2 RL entailments, e.g. cls-maxc1, cls-maxc2 and so on; however the value is not checked for number restriction premises in the current dependency graphs, such that the above rules can be loaded without considering the value of the number restrictions. A complex premise can be designed to impose a value condition that the value of the number restriction can only be 1 or 0.

Although the implementation was not created for OWL 2, it would be straightforward to implement an OWL 2 conformant *selective rule loading* algorithm can be implemented after design analysis as presented above. The implementation of an OWL 2 conformant *two-phase RETE* algorithm would also be quite straightforward since it is independent of the semantics and the rule set.

3.6 *Summary*

The analysis of the previous composition algorithms and resource-constrained OWL reasoners has pointed out some aspects the automatic reasoner composition research can explore. Two ideas for composition are then inspired from this analysis. They perform reasoner composition on both the rule set and inside the RETE algorithm.

This chapter presents the design of a composable rule-entailment OWL reasoner, COROR, which performs reasoner composition based on one the required expressivity of the ontology to be reasoned. Two novel reasoner composition algorithms, i.e. the *selective rule loading* algorithm and the *two phase-RETE* algorithm, are designed following the above two ideas to perform composition at both the rule set level and inside the RETE algorithm. Hence COROR instances can be automatically composed according to distinct ontologies.

The *selective rule loading* algorithm builds a selected rule set according to the OWL constructs included in the ontology to be reasoned. Some previous work has also been designed to work on a customized rule set, e.g. dynamic rule generation. However the novelty of this algorithm is the use of rule-construct dependencies to analyse if a rule is to be used in the reasoning. In general the selective rule loading is performed following the dependencies between rules and constructs: a rule is loaded if all the OWL constructs from its l.h.s. (premises) are included by the ontology (valid), and the loading of this rule may cause the addition of constructs in its r.h.s. (consequences) into the ontology, leading to the loading of more rules. These chain-like dependency relationships are represented here using rule-construct dependencies graphs. They are used to guide the selective loading of rules in the s*elective rule loading* algorithm. Later discussion shows the *selective rule loading* algorithm is reasoning algorithm independent, i.e. can be applied to other some other algorithms other than RETE (e.g. resolution). It can be applied to another semantic, e.g. OWL 2 RL, however requires a priori analysis of the semantics for rule-construct dependencies. Also the requirement of a priori analysis to the semantics when applied to a different semanticsmakes this composition algorithm unable to be *dynamically* applied onto a different semantics. Furthermore, the loading of a selective rule set raises a problem that re-execution of the *selective rule loading* algorithm is required if new OWL constructs are added to the ontology at runtime.

The *two-phase RETE* algorithm introduces a novel interrupted RETE network construction approach that integrates the gathering of information required for applying existing join sequence optimization heuristics into the RETE network construction. Hence the

composition of a customized and optimized RETE network for the particular ontology to be reasoned can be performed at the RETE network construction phase without introducing extra a priori RETE execution. In general the *two-phase RETE* algorithm breaks the RETE network construction into two separate phases and interrupts them with an initial fact matching. In the first phase a shared alpha network is built using a node sharing heuristic such that common conditions share alpha nodes. Then the initial fact matching is performed against the alpha network and matched facts are stored in the corresponding alpha memory. Some information about the ontology that is otherwise hard to collect is then easily collected at this stage, e.g. number of facts matched to a specific condition, selectivity factor between two conditions and so on. At the moment only the number of matched facts for each condition is collected, however more types of information can be collected. Information used at this stage it is used to the next phase for building a customized beta network for the particular ontology. Two heuristics are introduced at this stage for building customized join sequences, i.e. most specific condition first and pre-evaluation of join connectivity. Rather being applied directly to the rule set (as the original optimization does), the most specific condition first is applied taking into consideration the information about the ontology collected at the first phase, such that a customized RETE join sequence is constructed for the particular ontology according to it, e.g. the less matched facts of a condition the more specific it is and it is pushed closer to the front of the join sequence, and to the end otherwise. Another heuristic is the pre-evaluation of join connectivity of the individual conditions in given rule. It is applied after the most specific condition first to check its connectivity. Unconnected conditions are swapped backward for the first connected condition. After the customized RETE network is constructed stored facts continue to propagate along it, joining each other and firing rules as ordinary RETE algorithm does until no more rules to fire.

As discussed later in section 3.4.2.2 the information collected in the first stage after the initial matching may not be as accurate as when RETE terminates, since more RETE cycles are needed after the beta network is constructed and new (inferred) facts will be deduced (from rule firing) and fed back to the fact base, hence matching conditions and storing in the corresponding memory. However as discussed earlier most fact matching/joining operations are performed in the initial matching, and most inferred facts are generated at this stage as well, therefore it is reasonable to use the information collected at this stage (after the initial matching) to optimize the RETE network. The two-phase RETE network is not algorithm independent since it can only be applied to RETE algorithm. However it is also semantic

independent since its functioning does not rely on any particular rule set.

A hybrid algorithm is designed by using the *selective rule loading* algorithm to construct a selected rule set based on which the *two-phase RETE* algorithm is applied. This algorithm tries to combine the rule-level composition and algorithm-level composition such that they can compensate each other. This algorithm is not algorithm independent because the use of *two-phase RETE* algorithm. However it is semantic independent because both two composition algorithms are semantic independent.

Although at the moment COROR is designed to reason over OWL 1 ontology using the pD* semantics it is shown that both composition algorithms are extensible to OWL 2 RL without fundamental changes. A later discussion in section 5.2.5 shows the composition algorithms can be extended to support OWL 2 from the implementation perspective.

This section is targeted at objective 2. As a matter of fact the design of COROR is considered as a part of the major contribution as identified in the introduction. Implementation of COROR, as another part of the objective 2, is presented in Chapter 5. The study of the performance impacts brought by composition algorithms, as targeted in the objective 3, is presented in Chapter 6. In the following sections the design of a reasoner selection process is presented. It approaches the research objective 4 identified in the introduction.

Chapter 4
RESP: An Automatic Reasoner Selection Process

4.1 *Introduction*

As discussed in previous chapters the ability to deduce implied knowledge from an ontology has attracted ever more applications from various domains to use OWL reasoners to solve problems that are sometimes hard to solve using traditional approaches, such as bridging heterogeneity in diverse environment, or introducing more intelligence into data processing, or to detect inconsistencies in a knowledge base and so on. More usages are presented in a survey of semantic applications as presented in section 2.3.2. On the other hand the ever increasing application of OWL reasoning techniques in diverse domains also stimulates the development of OWL reasoning techniques due to the distinct reasoning-related requirements imposed. For example, as discussed earlier in motivation some sensor network systems require OWL reasoning to run on sensors, while bioinformatics systems/ontologies often need to thoroughly discover knowledge implied in the ontology. These differences in requirements then can be represented as the different (reasoning-related) *application characteristics* (ACs) possessed by the distinct applications. In order to handle different requirements, different reasoning technologies/features/capabilities are required. For example, a reasoner needs to be able to at least run on resource-constrained devices in order to provide reasoning support on sensors, and preferably it has some optimizations to reduce the resource consumption of reasoning. Similarly in order to thoroughly reveal all implied knowledge in bioinformatics ontologies, a reasoner may need to be able to completely classify OWL-DL ontology. These different reasoning technologies/features/capabilities are then the *reasoner characteristics* (RCs) of OWL reasoners. The diversified ACs of applications then give rise to reasoners with different RCs being needed.

Existing reasoner selections rely largely on consultation between application developers and reasoner experts, which is straightforward and sufficient at the moment with the relatively small and simple set of ACs/RCs that currently exist. However, it is envisioned that the ever more widespread adoption of OWL reasoning into applications in different domains and the rapid development and emergence of new OWL reasoning technologies may cause such a consultation approach to become increasingly inadequate in the future. This can be embodied in two aspects. Firstly, as semantic applications grow more complicated and move beyond initial prototyping stages, these applications will be developed and extended by dedicated application developers with little or no knowledge of the intricacies of ontology reasoning. A direct impact of this aspect is that some ACs are expressed in domain specific languages which sometimes may be difficult for reasoner experts to interpret and map into reasoner requirements. For example in some bioinformatics systems, a reasoner expert might be presented with a requirement that the selected reasoner need to be able to have such ability as to:

"*The causative agent of stomach ulcers is the bacterium Helicobacter pylori is, or that each instance x of disease of type X with symptom y of type Y is always preceded by infection by z of species Z in all of its patients suffering from X*" (from [Keet et al 2007]).

It is clear that it would not be very easy for most reasoner experts to interpret the above AC expressed in domain-specific language into a requirement for some RCs. In fact the above requires having the selected reasoner to be able to deal with existing gaps and to find out new relationships and new gaps. Therefore there are risks in selecting reasoners in the future using existing approaches: either a considerable amount of effort is required before an agreement is reached or, what is worse, an appropriate reasoner is selected due to misunderstanding. Secondly existing approaches require that a reasoner expert is accessible to application developers, which will not always be the case. These inadequacies motivate an automated approach for helping application developers to limit efforts required by consultation or even to help them independently select a suitable reasoner for their semantic applications.

From some informal discussions by the author with semantic application developers, it is found that semantic application developers usually have some "shallow" knowledge on OWL reasoning, e.g. they may understand *conjunctive query*, *ontology*, *OWL*, and so on. However they would get confused at more detailed and specific reasoner technologies such as *DL*, *tableaux*, *materialization*, *RETE*, *entailment rules*, and so on. These complicated

reasoning-specific terms, however, may sometimes be raised in a reasoner selection process when reasoner experts try to discuss with application developers whether the selected reasoner is appropriate or not. However, on the other hand, application developers know well about what application characteristics need to be implemented on their application, and requirements posed by them may often be expressed using domain specific languages, as the example given above. The requirements expressed in domain specific languages often become hindrances impeding reasoner experts to understand the real needs of the application. In fact these gaps would become wider with the emergence of more new application characteristics and new reasoning technologies (reasoner characteristics).

To bridge the above identified gaps between reasoner experts and application developers, a good way could be to design an automatic reasoner selection process, where a large number of pre-identified candidate application characteristics expressed in domain specific languages and pre-identified connections from these application characteristics to reasoner characteristics are stored. Application developers then only need to identify their required application characteristics and input these identified application characteristics into the process. The process can automatically recommend a most appropriate reasoner according to the existing connections.

This chapter introduces RESP, a novel computer aided OWL REasoner Selection Process, to enable application developers with little knowledge of the intricacies of OWL reasoning to independently select an appropriate reasoner for their applications based only on the ACs. This process imitates the flow of thought in the existing consultation-based reasoner selection process however what is novel is it serializes expertise on computer: reasoners are abstracted as RCs and interplays between RCs and ACs are serialized as *connections*. Hence application developers only need to identify the ACs of their application and then the selection in RESP is one of automatic matchmaking between the identified ACs and the RCs of reasoners according to the connections. RESP enables the reuse of expertise and therefore RCs and connections need to be identified only once and then reused in the future selections. This reduces the effort required by the selection of an appropriate reasoner: application developers need not to know the complicated algorithms of OWL reasoning or to look for a reasoner expert every time a semantic application is to be developed, which can lower the barriers for a wider range of applications to adopt OWL reasoning technologies.

An overview of RESP can be found in section 4.2. In section 4.3 discussions of 11 different reasoner-related aspects of semantic applications are presented, from which example ACs

and example connections are derived. These aspects are identified from the survey on semantic applications as described in the related work. Note that the example ACs and example connections are only for demonstrating RESP and still at their early stage. Hence they are neither definitive nor complete and sometimes are simple therefore are lack of practical usage. Still the distillation of them was not trivial, which required careful reviews and analysis of more than 80 pieces of published literature and online documents. Section 4.4 describes the matchmaking performed in RESP for selecting an appropriate reasoner using an artificial use case. A summary is presented in section 4.3.13.

4.2 *Overview of RESP*

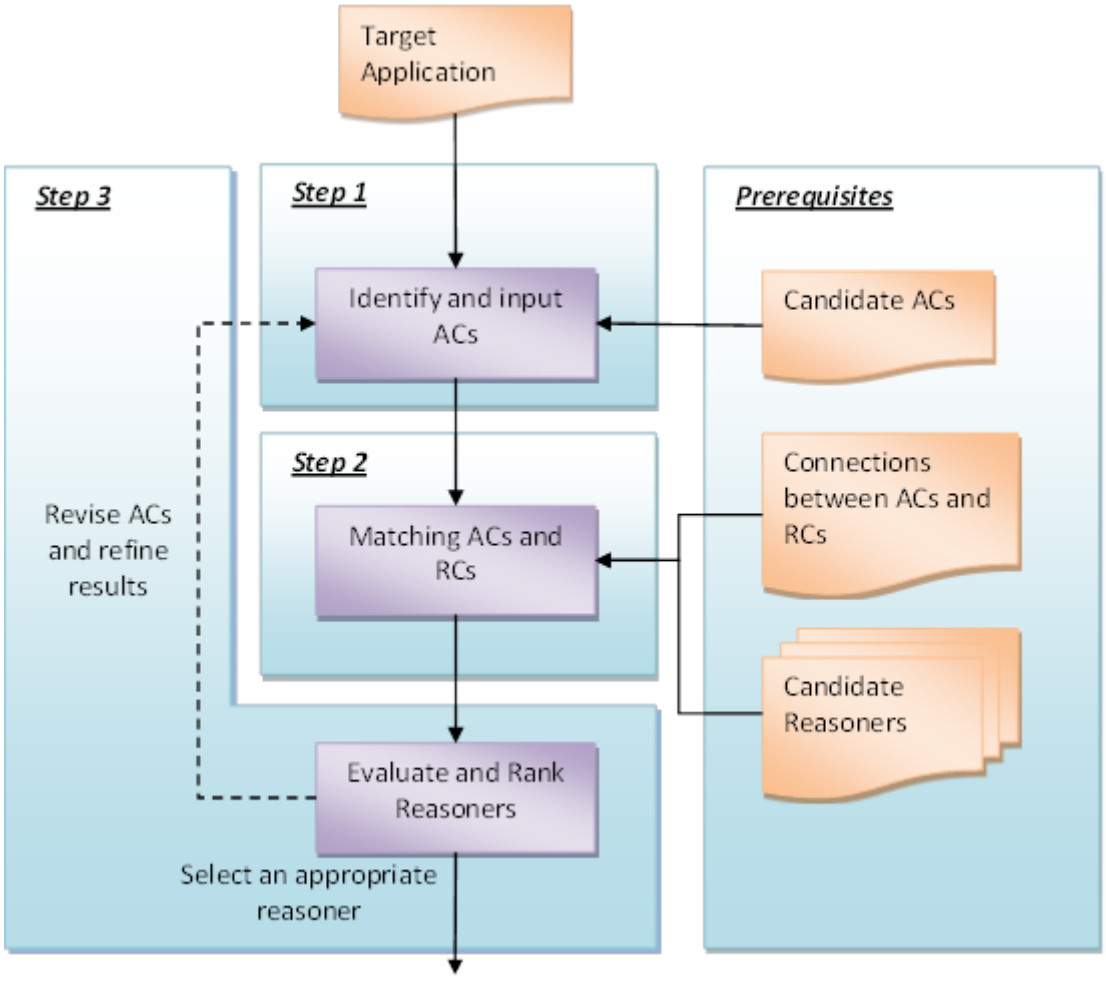


Figure 4-1: An overview of RESP

As illustrated in Figure 4-1, RESP consists of three steps:

1. In the first step, application developers identify the ACs for the target application and input them into RESP for selection.

2. In the second step, matchmaking is performed between the identified ACs and RCs of each *candidate reasoner* (reasoner to be selected), according to the pre-identified connections. The matchmaking result is given for each identified AC. At the moment the result can only be one of two values, namely *satisfied* or *not satisfied*. Results for all identified ACs are input into the next step for evaluating the satisfaction rate of each candidate reasoner for the input ACs.

3. In the third step, the *satisfaction rate* for each candidate reasoner is calculated using a straight forward approach by dividing the number of satisfied ACs (|*Satisfied_AC*|) by the number of selected AC (|*Selected_AC*|).

$$Satisfaction\_Rate = \frac{|Satisfied\_AC|}{|Selected\_AC|} \times 100\%$$

   This calculation is unique for this research and can directly represent the level of satisfaction of a candidate reasoner. For example, if five ACs are input ($|Satisfied\_AC|_R = 5$) and three are found to be satisfied by a candidate reasoner R ($|Selected\_AC|_R = 3$), then the satisfaction rate for R is 60%. The candidate reasoner with 100% satisfaction rate is then deemed the most appropriate reasoner for the given application. If none candidate reasoner is found to be the most appropriate reasoner, users can revise the input ACs according to the results loosening or tightening the ACs, and rerun RESP until an appropriate reasoner is selected.

Before RESP starts some prerequisites work are required to materialize expert knowledge that will be used in the selection into RESP (as illustrated in Figure 4-1). First, a set of *candidate* ACs needs to be constructed. From this set users identify ACs relevant to their applications in the first RESP step. The construction of candidate ACs requires expertise of domain experts. Second, candidate reasoners are registered with RESP as RCs by reasoner experts. RCs are derived from the survey of reasoner as presented in section 2.3.1.2. Third, connections between the candidate ACs and the RCs of candidate reasoners are analysed and modelled in RESP. This requires collaboration between domain experts and reasoner experts. Although a lot of effort may be required to accomplish the above three prerequisite

work in order to materialize expert knowledge into RESP, they are only one-off and once finished RESP can be reused in the subsequent selection for different applications within the domain.

Steps in RESP imitates the flow of thought in the existing consultation-based selection approach: application developers presents the requirements to reasoner experts (equal to RESP step 1), reasoner experts interpret the requirements into reasoning requirements and matchmaking reasoning requirements to reasoners based on their expertise on reasoning (equal to RESP step 2), reasoner experts recommend a most appropriate reasoner to application developers (equal to RESP step 3). However compared to the consultation-based approaches, what is novel of RESP is: RESP materializes expert knowledge used in the consultation-based process as ACs, RCs and connections in the process. There are many merits for doing this. First, it enables the reuse of expertise knowledge over time in the subsequent selection for other different applications without the attendance of a real reasoner expert. Second, materializing expert knowledge can prevent the selection of reasoners from being affected by geographical issue, e.g. application developers have the chance to run RESP anywhere in the world to perform reasoner selection without discussing with a real reasoner experts. Third, RESP hides the complicated interplay between semantic applications and reasoners from users such that users with little knowledge on ontology reasoning can use it.

Rather than being specific, RESP is designed to be a high level methodology to facilitate automatic reasoner selection, and therefore specific technical detail, e.g. the format of ACs/RCs, the matchmaking algorithm, the format that connections are authored, etc., are left unspecified until implementation stage. On one hand, this gives more flexibility to RESP implementers as RESP can be implemented to be domain-specific for a specific application domain. Therefore domain specific languages can be used to label candidate ACs allowing RESP users to have better understanding of the candidate ACs when they identify relevant ACs for their applications. Furthermore the matchmaking can be realized as general rule engines (connections as rules), or be hardcoded according to the preference of specific implementers. However on the other hand, problems can be raised: users need to specify their own format for materializing ACs, RCs and connections, and design their own algorithm for matchmaking, which could require extra efforts for implementing RESP.

In order to demonstrate RESP, the interplay between the surveyed semantic applications and reasoner characteristics has been investigated, and example candidate ACs and connections

have been derived. For clearer presentation, the interplay is discussed in the next section from 11 reasoning-related aspects. Note again, the investigation and derived ACs and connections are still at their early stage, and hence are neither definitive nor complete for practical usage. Still their development/distillation required careful reviews and analysis of more than 80 published pieces of literature and online documentations.

4.3 *Discussion of Interplay between Semantic Applications and RCs*

In this section the interplay between the surveyed semantic applications and RCs is discussed according to 11 reasoning-related aspects, namely: *frequently changing KB*, *terminology-centric reasoning*, *required semantics*, *query-related issues*, *rules*, *concrete domains*, *closed-world features*, *large KB*, *ontology manipulation*, *explanation of reasoning*, and *miscellaneous*, with each corresponding to a following subsection. The use of these reasoning-related aspects was motivated from reviewing the literature of the surveyed semantic applications where these reasoning-related aspects are the points where semantic applications are tied to OWL reasoners. Example candidate ACs and connections derived from each reasoning-related aspect are listed at the end of each discussion. Example candidate ACs and connections are also used for implementing a prototype tool for automatic reasoner selection using RESP, as will be depicted in the next chapter.

Several remarks are in order. Firstly, as mentioned before, the derived example candidate ACs and connections are only for demonstration and therefore they are not complete and definitive enough for practical usage. More usable candidate ACs and connections can be identified with the collaboration of domain experts and reasoner experts. Secondly, discussions are sometimes limited rather than in-depth and exhaustive. This is reasonable since it is the demonstration of RESP rather than having a thorough investigation of the interplay between semantic applications and reasoners, that is the major goal of this research. To limit the complexity of the discussions only three major types of reasoners are considered in the discussions, which are *DL-tableaux reasoners*, *Rule-entailment reasoners*, and *resolution-based reasoners*. Thirdly, for simplicity the performance aspects are dropped from the discussions. However it is obvious that the performance aspects will play an important role in deciding the selection of an appropriate reasoner for some semantic applications and so taking the performance of reasoners as an AC is part of the future work. Finally RCs and their values used in the discussion are selected from the state of the art survey of OWL reasoners as described in section 2.3.1.2.

The RCs and their values used in the discussions are listed in section 4.3.1. An RC may have

multiple different values and these values were gathered in the survey of reasoners. Discussions of the 11 reasoning-related aspects are separately presented in subsections from 4.3.2 to 4.3.12. Derived example candidate ACs and the correspondingly derived connections are presented at the end of each subsection using a table format. In the first column of the table derived example candidate ACs are listed and in the second column the corresponding connections between the AC and RCs are given. In order to be precise and concise, some mathematical symbols are used for describing connections and RCs and values are referred in connections as codes. A summary of all example candidate ACs and connections is given in section 4.3.13.

4.3.1 RCs used

Before discussions of the 11 reasoning-related aspects are presented, RCs and values used in the discussions are listed in Table 4-1. All RCs distilled from the survey of reasoners (as introduced in section 2.3.1.2) are used here in the discussions. However to avoid over-complex discussions of the interplay, the values of some RCs are restricted. Only three types of reasoners will be considered in the discussions, namely *DL-tableaux reasoners*, *Rule-entailment reasoners*, and *resolution-based reasoners*. The values of the RC *reasoning algorithm* are then correspondingly restricted to only the algorithms used by the above three reasoner types. Although discussions are restricted to three types of reasoners, still they are the major reasoner types and hence the discussions are of general sense. Values for two other RCs, namely *query support* and *rule support*, are also restricted to avoid too many values: for *query support* only *atomic query*, *SPARQL*, *nRQL* and *SeRQL* are kept and for *rule support* only *SWRL* and *Jena* are kept. However extension to support all values for these RCs does not require fundamental changes to existing connections since some counterpart values are already used in the discussions.

Table 4-1: A summary of values of the corresponding reasoner characteristics in the survey

Reasoner characteristic	**Values**		
Reasoning algorithm (ALGM)	DL-Tableaux (*tableaux*)	RETE (*rete*)	FOL prover (*fol*)
	Prolog (*prolog*)	Datalog (*datalog*)	
Reasoner type (TYPE)	DL-tableaux (*dl*)	Rule-entailment (*entailment*)	Resolution-based (*resolution*)
	Hybrid (*hybrid*)	Others (*others*)	
Reasoner expressivity (EXPR)			
Completeness (CPLT)	Yes (*yes*)	No (*no*)	
Reasoning tasks (TASK)	OWL entailment (*ent*)	Classification (*clsf*)	Realization (*real*)
	Concept satisfiability (*sat*)	Conjunctive query answering (*conj*)	KB consistency (*cons*)
Materialization (MTLZ)	Yes (*yes*)	No (*no*)	
Incremental reasoning (INCL)	Incremental classification (*classify*)	Incremental consistency checking (*consistency*)	Incremental materialization maintenance (*materialize*)
Query support (QUERY)	Atomic (*atomic*)	SPARQL (*sparql*)	nRQL (*nrql*)
	SeRQL (*serql*)		
Rule support (RULE)	SWRL (*swrl*)	Jena (*jena*)	
Closed-world features (CWA)	OWL (*owl*)	Rule (*rule*)	Query (*query*)
Concrete domain (CD)	XSD datatypes (*xsd*)	User-defined datatypes (*user*)	computation/comparison on datatypes (*comp*)
Database support (DB)	Native DB accessible (*access*)	Native DB reasoning (*reasoning*)	
Remote interface (RINF)	DIG (*dig*)	Self-defined (*self*)	
User access (ACCESS)	GUI (*gui*)	Command line (*cmd*)	
Explanation (EXPL)	Native explanation (*yes*)	No (*no*)	
Ontology manipulation (MANI)	OWLAPI (*owlapi*)	Jena (*jena*)	API (*api*)
Platforms (PLAT)	J2ME (*j2me*)	J2SE (*j2se*)	C++ (*cpp*)
	C# (*csharp*)	Prolog (*prolog*)	
OS (OS)	Windows (*win*)	Linux (*lin*)	MacOS (*mac*)
	Symbian (*sym*)	Android (*and*)	PalmOS (*pm*)
	SunSPOT (*sun*)	WinMobile (*wm*)	TinyOS (*tos*)

4.3.2 Aspect 1 - Frequently Changing Knowledge Bases

As discussed earlier in the survey of semantic applications, many applications such as semantic pub/sub systems [Halaschek-Wiener and Kolovski 2008, Haarslev and Möller 2003a, Halaschek-Wiener et al 2006] and semantic sensor network systems [Stuckenschmidt et al 2010] envision that their knowledge bases could be frequently updated by terminological or individual axioms. For examples, normally publications may be continuously received by the broker of a semantic publish/subscribe system and then combined into the knowledge base in order to match them against publications; sensor observations may be constantly generated by sensors and combined into the knowledge base for inference. As indicated by the survey of reasoners many existing OWL reasoners, especially DL-tableaux reasoners, are designed to handle static knowledge bases, namely once the knowledge base has been changed, the entire knowledge base needs to be re-reasoned, which is sometimes inefficient. However these applications often requires quick turnover and therefore the ability to efficiently reason over changing KBs turn out to be an important characteristic for the selected reasoners.

For DL-tableaux reasoners incremental reasoning techniques have been developed enabling KB consistency to be checked incrementally for ABox updates [Halaschek-Wiener et al 2006] and also terminology to be classified incrementally for TBox updates [Parsia et al 2006, Grau and Halaschek-Wiener 2010], which shows the potential for those DL-tableaux reasoners with these algorithms implemented to reason over updates efficiently.

Resolution-based reasoners can pre-compute and materialization some important reasoning tasks materialization to enhance runtime query performance (e.g. KAON2 materializes classification results), however this would require re-reasoning in order to maintain materialization when the knowledge base changes. An incremental materialization maintenance algorithm is devised to enable efficiently handling of TBox update in resolution-based reasoners, in particular those using Datalog engine [Volz et al 2005]. Work has been done in deductive database to handle incremental materialization maintenance for fact updates [Staudt et al 1996], but none has been found applied in resolution-based reasoners in the survey. Given the goal-based query-time reasoning feature of the resolution algorithm, resolution-based reasoners can perform localized reasoning and then can handle updates efficiently.

Rule-entailment reasoners use RETE to perform reasoning. According to the description of RETE given in the background knowledge, RETE inserts each newly added fact into the

RETE network where the fact is incrementally matched and joined to existing facts. Retraction is a similar process as insertion: the retracted fact is matched and joined in the RETE network. However rather than generating/caching intermediate results and inferring facts, RETE removes extant intermediate results and extant inferred facts for retraction. Since RETE view both TBox updates and ABox updates as facts, RETE has the intrinsic ability to handle both updates incrementally.

Derived ACs and connections for this aspect are given in Table 4-2. To help readers understand the derived connections, some general rules used to construct connection are explained here. In general a connection states the conditions that a candidate reasoner needs to satisfy in order to satisfy the ACs. For examples, the condition TYPE = *dl* states that the value of the RC *reasoner type* of the candidate reasoner needs to *DL-tableaux* (refer to Table 4-1 for codes); the condition classify ∈ INCL states that the value *incremental classification* needs to be included in the RC *incremental reasoning* of the candidate reasoner. Sometimes the ACs of the application are also considered in conditions. They are underlined in order to distinguish them from RCs and RC values. For example, the condition Required reasoning tasks ⊆ TASK states that the AC *reasoning tasks* required by the application (Required reasoning tasks) need to be a subset of the RC *reasoning tasks* supported candidate reasoner (TASK); the condition conjunctive queries are required state that conjunctive queries are required by the application. Conditions in a connection can be connected using two types of connectors: **and** and **or**. An **and** clause is satisfied only if all its composing conditions (clauses) are satisfied. An **or** clause is satisfied only if any of its composing conditions (clauses) are satisfied.

Table 4-2: Candidate ACs and Connections Derived from Frequently Changing Knowledge Bases

Derived AC	Connections
Frequent terminological update	(TYPE = *dl* **and** *classify* ∈ INCL) **or** TYPE = *entailment* **or** (TYPE = *resolution* **and** MTLZ = *yes* **and** *materialize* ∈ INCL) **or** (TYPE = *resolution* **and** MTLZ = *no*)
Frequent instance update	(TYPE = *dl* **and** *consistency* ∈ INCL) **or** TYPE = *entailment* **or** (TYPE = *resolution* **and** MTLZ = *no*)

4.3.3 Aspect 2 - Required Semantics

The amount of semantics required may vary from semantic applications and usually the selected reasoner needs to cover the required semantics of the application. For example reasoning an ontology as expressive as the wine ontology will usually require a full-fledged reasoner that covers the entire OWL DL semantics to be selected, e.g. Pellet or FaCT++, but if an ontology falls into the $\mathcal{ALN}$ subset of OWL DL, it can be classified using a reasoner implementing relatively simple structural subsumption algorithm [Baader et al 2007]. This characteristic can also be used to assign an appropriate reasoner to applications whose ontology is within some specific OWL sublanguages. For example, some pD*-based rule-entailment reasoners such as OWLIM and BaseVISor can be used if the ontology uses OWL constructs within the pD* semantics; CEL classifies on DL EL++ into which many bioinformatics ontology fall; if the ontology fall into the DL-Lite subclass of OWL then some dedicated DL-Lite reasoners such as Owlgres and QuOnto can be used to perform efficient query answering services.

Derived AC and connection for this aspect are given below in Table 4-3.

Table 4-3: Candidate AC and Connection Derived from Required Semantics

Derived AC	Connections
Required Semantics	EXPR > Required semantics

4.3.4 Aspect 3 – Reasoning Tasks

The required reasoning tasks may vary from applications and hence the selected reasoner needs to perform reasoner tasks required by the application. One problem is often naturally raised when discussing reasoning tasks: the completeness of OWL-DL reasoning. Some applications focus on large dataset, such as sensor network systems, or context-aware systems. For these applications, discovering implied knowledge through OWL reasoning is often like extra points and hence complete OWL-DL reasoning is not necessary. Instead incomplete but more data-efficient reasoners are preferable, e.g. rule-entailment reasoners (still the chosen reasoner needs to provide the required reasoning tasks). However some other applications in particular bioinformatics/medical applications such as the Gene ontology and SNOMED ontology have their knowledge bases mostly populated by structured concepts and relations, and they usually expect exhaustive reasoning over the given ontology. For these applications, efficiency of reasoning and the completeness of capturing the iff semantics of OWL-DL turn out to be important for the selected reasoners.

All surveyed DL-tableaux reasoners including Pellet, Fact++, and RacerPro are designed for such requirements and can well satisfy these applications. Some resolution-based reasoners translate OWL ontology into datalog or prolog programs following the ontology-specific approach as described in related work (section 2.3.1.1.2). For examples, KAON2 can completely classify ontology within DL $\mathcal{SHIQ}$ subset of OWL-DL [Hustadt et al 2004a]. Thea and Bubo can completely handle DLP [Vassiliadis et al 2009, Volz et al 2003]. CEL can efficiently classify EL++ ontology, however it does not belong to any of the three reasoner types, namely DL-tableaux reasoners, rule-entailment reasoners, or resolution-based reasoners. Since entailment rules used in rule-entailment reasoner cannot fully capture the iff semantics of OWL-DL, all rule-entailment reasoners do not perform complete OWL-DL reasoning.

Derived ACs and connections are given in Table 4-4.

Table 4-4: Candidate AC and Connection Derived from Terminology-Centric Reasoning

Derived AC	Connections
Reasoning tasks	Required reasoning tasks ⊆ TASK
Completely derive all implied knowledge	CPLT = *yes* **and** EXPR > Required Semantics **and** (TYPE != *entailment*)

4.3.5 Aspect 4 - Query

How queries can be issued may vary from applications. Some applications/ontologies require to pose complex queries in query languages such as SPARQL [Russomanno et al 2005, Compton et al 2009a, Eid et al 2007, Kim et al 2008, Compton et al 2009b] while for some others application posing atomic queries though an API is sufficient [Keeney et al 2008]. However different reasoner implementations have different capabilities in answering queries. Many state of the art reasoners such as Pellet (latest Ortiz API), KAON2, RacerPro, Jena (with ARQ), OWLIM and so on support conjunctive queries. They can be selected for the applications requiring complex queries. However some reasoners, e.g. CEL, FaCT++ and Jena (core), only allow queries to be posed either using pre-defined directives in command line or through an API. Hence they are not suitable for applications which need to put complex queries.

In addition the syntax and functionality differences in different query languages can also affect the selection of reasoners. SPARQL uses a RDF-based triple syntax. SPARQL is

(partly) supported by many of the state of the art reasoners, such as Pellet, KAON2, Jena (with ARQ) and RacerPro. The nRQL is an axiom-based ABox conjunctive query language specifically designed for RacerPro. OWLIM supports SeRQL, a RDF query language. KAON2 enables queries to be formulated using F-logic. Bossam uses Buchingae [Jang and Sohn 2004]. Thea supports queries to be authored using Prolog rules. There are some query languages, such as C-SPARQL [Barbieri et al 2010b], that are implemented but however are not yet incorporated by state of the art reasoners, and so they are not discussed here.

Derived ACs and connections for this aspect are given in Table 4-5.

Table 4-5: Candidate ACs and Connections Derived from Query-Related

Derived AC	Connections
Queries	(Atomic queries are required **and** QUERY $\neq \varnothing$) **or** (Conjunctive queries are required **and** QUERY $\cap$ {*sparql, serql, nrql*} $\neq \varnothing$)

4.3.6 Aspect 5 - Rules

Rules are widely used in some semantic applications to perform tasks such as fusing sensor readings, handling context information or transferring partitive properties in medical informatics application based on application specific semantics [Calder et al 2010, Sheth et al 2008, Compton et al 2009a, Brennan et al 2009, Rector 2002, Ejigu et al 2007]. Although varying in the syntax and expressivity, rule-based reasoners have the intrinsic capability to model and process rules. Many tableaux reasoners are also extended to support rules, e.g. Pellet and RacerPro partly supports SWRL. However some of the state of the art reasoners still lack of support for rules. For example there is no evidence that FaCT++ can process rules and therefore it is not appropriate to be selected for applications using rules.

The expressivity and syntax differences among rule languages can also affect the selection of reasoners to some extent. For example if application developers prefer to uses Jena rules to model domain knowledge, Jena would be preferable than the other rule-based reasoners such as Bossam. Other examples include that: as SWRL does not support negation as failure (NaF) a reasoner supporting only SWRL would not be appropriate for applications requiring NaF; Jena rules is triple-based and therefore does not express n-arity predicates; and so on. Considering this discussion is only for identifying example ACs and connections for demonstrating RESP rather than insight and complete discussion all interplay between different requirements on rules from applications and different rules languages, it is deemed

that the requirement for rules is satisfied if the selected reasoner has some types of rule support or the reasoner is one of the rule-based reasoners.

Derived AC and connection are given in Table 4-6.

Table 4-6: Candidate AC and Connection Derived from Rules

Derived AC	Connections
Rules required	RULE ∩ {*swrl*, *jena*} ≠ ∅ **or** TYPE ∩ {*entailment*, *resolution*} ≠ ∅

4.3.7 Aspect 6 - Concrete Domains

It is generally accepted that many real-world applications such as sensor network systems or context-aware systems need to handle some information such as temperature, geo-graphical location, time, moisture, speed and so on which often needs to be modelled as concrete objects such as string, numeric numbers, and time and so on. OWL[12] has limited capability to model and handle concrete domains: concrete objects of some XSD datatypes can serve as values of data value properties. However OWL lacks abilities to (1) express comparisons or computations between concrete objects, e.g. the convention between Celsius and Fahrenheit, and (2) model arbitrary user-defined datatypes, e.g. human age is from 0 to 150. These two has been generally accepted to be of important for many practical applications.

Due to this reason some semantic applications choose to handle concrete data outside OWL by interfacing an outside system with these abilities missing from OWL, e.g. a conventional publish/subscribe system [Keeney et al 2008, Keeney et al 2010] or a rule system [Sheth et al 2008, Agostini et al 2005] (many rule systems have a wide range of builtins for handling concrete objects) and so on. OWL is then used in such applications to model complex and non-concrete-data-related knowledge or information, e.g. concept hierarchies, user activities or human interests and so on. However, reasoning over concrete domains has been extensively studied in previous work for DL and OWL [Haarslev and Möller 2002, Lutz 1999, Hustadt et al 2004b, Haarslev and Möller 2003b, Pan 2004], and indeed the latest OWL 2 specification enables the definition of user-defined datatypes and datatype restrictions. Pellet supports user-defined datatypes to be embedded in OWL ontology using added OWL constructs such as owl:onDataRange, owl:datatypeComplementOf,

[12] OWL 2 has support for user-defined datatypes and datatype restriction. However given that this thesis is based on OWL 1, if not specified, all the terms "OWL" referred in this thesis are limited to OWL 1.

xsd:minInclusive and so on. Furthermore Pellet supports part of SWRL including mathematical/string builtins and hence it also enables limited comparisons and computation on concrete objects (except for date, time and duration) to be modelled and processed in SWRL rules. Jena also supports user-defined datatypes and it has a range of pre-defined builtins for handling concrete objects. Furthermore it enables user-defined builtins to be constructed and called in rules. Hence potentially enables arbitrary computation to be modelled as user-defined builtins. Bossam allows Java methods to be called as builtins in rules as well.

Derived ACs and connections are given in Table 4-7.

Table 4-7: Candidate ACs and Connections Derived from Concrete Domains

Derived AC	Connections
Concrete domains	(XSD datatypes are required **and** *xsd* ∈ CD) **or** (User-defined datatypes are required **and** *user* ∈ CD) **or** (Comparison and computation are required **and** (*comp* ∈ CD **or** RULE ≠ ∅))

4.3.8 Aspect 7 - Closed-World Features

Some systems, e.g. database-based systems, assume the known knowledge of this system is a complete modelling of the domain, and missing knowledge (the knowledge fails to derive) is simply regarded as *not true*. These systems are deemed to follow a closed world assumption (CWA). However OWL assumes an open world assumption (OWA) where the knowledge base is only a subset of domain and all missing knowledge is regarded as *unknown*. The difference between CWA and OWA can lead to different results in reasoning. A typical example could be owl:someValuesFrom. In standard OWL semantics no value for a property restricted by owl:someValuesFrom can lead to an anonymous object to be constructed: the object must be there however the only matter is it is unknown. However in CWA-based OWL semantics such as [Pellet ICV, Tao et al 2010] this would be interpreted as a breach of constraints: according to the knowledge base, there is no such object exists.

The standard OWL semantics are useful in most cases but for some practical systems closed-world features are required for answering negated queries or for checking integrity constraints. For example, a flight system may want to query which two cities are not connected by a direct flight or a student record system may expect inconsistency to be reported when it is found no student number is assigned for a student rather than assigning

an anonymous object. Two approaches are taken to cope with the requirement of CWA. Firstly a dedicated reasoner can be constructed implementing (local) closed-world semantics for OWL. A lot of research in OWL has been devoted to enable (local) closed-world semantics in DL or OWL [Donini et al 1992, Katz and Parsia 2005, Motik et al 2006, Motik et al 2007, Tao et al 2010]. In fact Pellet ICV [Pellet ICV] is the only tool found in the survey of the state of the art reasoners supporting a closed-world flavoured OWL. A second approach is to interfacing an outside rule or query system supporting some closed-world features. For example SPARQL supports the use of constructs OPTIONAL and BOUND to achieve negation as failure [Nerode and Shore 1997], i.e. **not** P is assumed from failure to derive P. This enables users to pose negated query and check for integrity constraints without changing the closed-world semantics of OWL. Similarly some other query/rule languages such as nRQL (RacerPro query language, refer to [RacerPro Reference Manual v1.9.2]), SeRQL (Sesame query language, refer to [Sesame User Guide]) or Jena rules also support negation as failure.

Derived ACs and connections are given in Table 4-8.

Table 4-8: Candidate ACs and Connections Derived from Closed-World Features

Derived AC	Connections
Integrity constraints	CWA $\neq \varnothing$ **or** RULE $\neq \varnothing$ **or** QUERY $\cap$ {*sparql*, *nrql*, *serql*)
Closed-world queries (negated queries)	*query* $\in$ CWA **or** QUERY $\cap$ {*sparql*, *nrql*, *serql*) $\neq \varnothing$

4.3.9 Aspect 8 - Large Knowledge Base or Persistent Storage

Some applications often need to process a large knowledge base or need data to be persistently stored for offline access. For examples, some semantic context-aware systems require storing received context information locally for further analysis [Gu et al 2007, Boehm et al 2008]. This is also the case for many sensor network systems where sensor observations are maintained in a centralized database for analysis [Sheth et al 2008]. Database support in the selected reasoners is then a key characteristic for such applications. Many OWL (RDF) stores are available, such as Jena TDB [Jena TDB], BigOWLIM, AllegroGraph [Allegrograph 2011], KAON2, Oracle database 11g, Parliament [Kolas et al 2009], and PelletDB [PelletDB]. However some of them implement optimizations enabling more scalable (and may be more efficient) database reasoning. For example, KAON2 implements a virtual ontology technique that maintains only a view of the ontology in memory and the real data are still kept in database. Jena TDB provides only storage and

SPARQL querying for RDF dataset, but the entire OWL ontology still needs to be loaded and reasoned in memory. For the other OWL/RDF data stores their optimizations are not mentioned everywhere since they are commercialized. AlegroGraph supports storing data set in database with a reasoning service of the RDFS++ [RDFS++] subset of OWL; Parliament supports inference over a selected subset of OWL (equivalent classes and equivalent, inverse, symmetric, functional, inverse functional, and transitive properties); BigOWLIM supports the pD* subset of OWL. Some in-memory reasoners also gain access to databases by interfacing to a database-enabled OWL framework. For example, FaCT++ and CEL can be plugged into OWLAPI for which OWLDB [Henss et al 2009] is a de facto database backend. Similarly as Jena, they load the entire ontology into memory and do not have any specific optimizations for scalable database reasoning.

Derived ACs and connections are given in Table 4-9.

Table 4-9: Candidate ACs and Connections Derived from Large Knowledge Base or Persistent Storage

Derived AC	Connections
Database support	DB != ∅ **or** MANI ∩ {*owlapi, jena*} ≠ ∅

4.3.10 Aspect 9 – User/Application Manipulation of Ontology

Although not explicitly notified in many published paper or webpages, the ability to manipulate ontology/knowledge (e.g. changing/adding/deleting axioms) is important for many applications to combine changes into the ontology at runtime or to allow human users to alter the ontology. For examples, it is generally accepted that semantic sensor network systems or semantic context-aware systems usually need to combine sensor readings or context information into the knowledge base/ontology at runtime in order to using ontology/KB manipulation APIs; some semantic publish/subscribe systems may allow users to dynamically change their current situation or interests (modelled as complex concepts) [Agostini et al 2005, Luther et al 2008]; in some applications such as the Gene ontologies, human users may want to use the reasoner standalone to manipulate, reason over and query the ontology, thus requiring either a rich and full-functioned GUI or command line to be supported by the selected reasoner, or that the selected reasoner needs to be pluggable into some graphical ontology manipulation tools such as Protégé (though DIG).

According to the survey of reasoners there are three ways to enable ontology manipulation. Firstly some state of the art reasoners provide a rich set of native APIs/command line

derivatives/GUI interfaces. For examples, KAON2 has a rich set of APIs for ontology reading, parsing and manipulation; CEL has a set of command line derivatives to enable concept/role construction; RacerPro has a graphical interface allowing users to manipulate and reason over ontologies. A second approach is to plug reasoners into some semantic frameworks such that a rich set of ontology manipulation API is available. For examples, Pellet can be plugged into both OWLAPI and Jena; FaCT++ and CEL can work with OWLAPI. A third approach is through the DIG interface. It provides a standardized XML interface for OWL manipulation and query, i.e. a set of tell verbs for axiom assertions and a set of ask verbs for querying. Furthermore DIG compliant reasoners can be plugged into Protégé enabling users to manipulate, query and reason over ontologies using Protégé GUI interface. Many state of the art reasoners support DIG such as RacerPro, FaCT++, Pellet and CEL and so on.

Some other reasoners, such as Bossam, have very limited native ontology manipulation interfaces and are also not pluggable into any ontology frameworks. They are then not quite suitable for these applications.

Derived ACs and connections are given in Table 4-10.

Table 4-10: Candidate ACs and Connections Derived from Ontology Manipulation

Derived AC	Connections
Ontology manipulation	(API manipulation is required **and** MANI ≠ ∅) **or** (CMD manipulation is required **and** *cmd* ∈ ACCESS) **or** (GUI manipulation is required **and** (*gui* ∈ ACCESS or *dig* ∈ RINF))

4.3.11 Aspect 10 - Explanation of Reasoning and Ontology Debugging

Explanation of deductions and debugging are required by some applications like ontology engineering tools such as SWOOP [SWOOP], configuration management tools [Baader et al 2007, Shahri et al 2007], or bioinformatics applications [Keet et al 2007] for explaining reasoning results or for providing justification for modelling inconsistency. In general two approaches are taken by existing reasoners to perform reasoning explanation. Some reasoners implement native explanation components enabling justifications to be derived for inferences, such as Pellet, Jena and Bossam. A second approach is to plug into OWLAPI that implements a black box debugging mechanism [Kalyanpur et al 2006]. Through this, explanations can be generated for reasoners that do not natively support explanation, e.g.

FaCT++ and CEL.

Derived ACs and connections are given in Table 4-11.

Table 4-11: Candidate ACs and Connections Derived from Explanation of Reasoning and Ontology Debugging

Derived AC	Connections
Reasoning explanation	EXPL = *yes* **or** *owlapi* ∈ MANI

4.3.12 Aspect 11 - Miscellaneous

Some other application characteristics may also affect the selection of an appropriate reasoner. In some applications such as bioinformatics ontology. Some applications may need to remotely access reasoning services to achieve thin client. This requirement will need the implementation of some kinds of remote interfaces on the reasoner, e.g. DIG. Some other application characteristics include the requirement to run on a particular operating systems (Linux, Windows, MacOS), open sources, user support, price, and so on. Given that these ACs are not relevant to the reasoning capability of a reasoner and hence they are not discussed in detail in this thesis. From this aspect, three ACs are identified as examples.

Derived ACs and connections are given in Table 4-12.

Table 4-12: Candidate ACs and Connections Derived from Miscellaneous

Derived AC	Connections
Human access	ACCESS ≠ ∅ **or** *dig* ∈ RINF
Remote access	RINF ≠ ∅
Operating systems	Required os ⊆ OS

4.3.13 A Summary of Example Candidate ACs and Connections

A summary of the example Candidate ACs and connections can be found in Table 4-13.

Table 4-13: A Summary of Example ACs and Connections

Derived AC	Connections
Frequent terminological update	(TYPE = *dl* **and** *classify* ∈ INCL) **or** TYPE = *entailment* **or** (TYPE = *resolution* **and** MTLZ = *yes* **and** *materialize* ∈ INCL) **or** (TYPE = *resolution* **and** MTLZ = *no*)
Frequent instance update	(TYPE = *dl* **and** *consistency* ∈ INCL) **or** TYPE = *entailment* **or** (TYPE = *resolution* **and** MTLZ = *no*)
Required Semantics	EXPR > Required semantics
Reasoning tasks	Required reasoning tasks ⊆ TASK
Completely derive all implied knowledge	CPLT = *yes* **and** EXPR > Required Semantics **and** (TYPE != *entailment*)
Queries	(Atomic queries are required **and** QUERY ≠ ∅) **or** (Conjunctive queries are required **and** QUERY ∩ {*sparql*, *serql*, *nrql*} ≠ ∅)
Rules required	RULE ∩ {*swrl*, *jena*} ≠ ∅ **or** TYPE ∩ {*entailment*, *resolution*} ≠ ∅
Concrete domains	(XSD datatypes are required **and** *xsd* ∈ CD) **or** (User-defined datatypes are required **and** *user* ∈ CD) **or** (Comparison and computation are required **and** (*comp* ∈ CD **or** RULE ≠ ∅))
Integrity constraints	CWA ≠ ∅ **or** RULE ≠ ∅ **or** QUERY ∩ {*sparql*, *nrql*, *serql*)
Closed-world queries (negated queries)	*query* ∈ CWA **or** QUERY ∩ {*sparql*, *nrql*, *serql*) ≠ ∅
Database support	DB != ∅ **or** MANI ∩ {*owlapi*, *jena*} ≠ ∅
Ontology manipulation	(API manipulation is required **and** MANI ≠ ∅) **or** (CMD manipulation is required **and** *cmd* ∈ ACCESS) **or** (GUI manipulation is required **and** (*gui* ∈ ACCESS or *dig* ∈ RINF))
Reasoning explanation	EXPL = *yes* ***or*** *owlapi* ∈ MANI
Human access	ACCESS ≠ ∅ **or** *dig* ∈ RINF
Remote access	RINF ≠ ∅
OS	Required os ⊆ OS

4.4 *Matchmaking*

Users identify from the candidate ACs those relevant to their applications and input them into RESP. Candidate reasoners are registered in RESP as profiles of RCs. For example, COROR can be registered as the profile given in Table 4-14. Then the matchmaking process

is to check satisfiability of the selected ACs according to the connections identified in section 4.3.

Table 4-14: An example reasoner profile for COROR

Reasoner characteristic	Values
ALGM	*rete*
TYPE	*entailment*
EXPR	*pD**
CPLT	*no*
TASK	*ent*
MTLZ	*yes*
INCL	*no*
QUERY	*atomic*
RULE	*jena*
CWA	*rule*
CD	*xsd, comp*
DB	*no*
RINF	*no*
ACCESS	*no*
EXPL	*no*
MANI	*api*
PLAT	*j2me*
OS	*win ,lin, mac, sym, and, wm, sun*

A simple use case is given showing how matchmaking is performed in RESP to assist the reasoner selection for applications using an artificial demonstration scenario and the above identified candidate ACs and connections.

"*Bob wants to build a semantic sensor network management system where sensors (SunSPOT) are clustered with one sensor as the head of each cluster. In this system cluster heads are expected to perform correlation over failures within the cluster. This can (1) identify the root cause of a set of received failures within a window, and then (2) reduce the traffic in the sensor network. To perform this Bob wants to use OWL to model failure hierarchy and causal relationships between failures are modelled as rules, as listed below:*

(1) ***BatteryNearDepletion*** *is a root cause of itself.*

(2) ***NoNodeAvailable*** *from a node can be caused by the* ***BatteryNearDepletion*** *from another node on its route to destination (include destination).*

(3) ***NoNodeAvailable*** *from a node can be caused by the* ***NodeOut*** *from another node on its route to destination (include destination).*

(4) ***NodeOut*** *from a node can be caused by the* ***BatteryNearDepletion*** *from another node on its route to gateway (gateway not included).*

(5) ***NodeOut*** *from a node can be caused by the* ***NodeOut*** *from another node on its route to the gateway (gateway not included).*

The system also needs XSD datatype values to be supported and handled by the reasoner so that sensor observations can be modelled and processed."

By analysing the requirements, Bob finds that:

- the reasoner needs to support rules in order to model and process failure correlation rules;
- the reasoner needs to run on the J2ME CLDC 1.1 platform since it is the software running platform for Sun SPOT;
- the reasoner needs to support xsd datatypes and computation/comparison over xsd datatypes in order to model and process sensor observations.

He then goes to the candidate ACs and finds the corresponding candidate ACs:

- Rule required,
- Platform: J2ME
- Concrete domains: XSD datatypes

in RESP. He then inputs them into RESP and RESP will then perform the matchmaking attempting to find out the most appropriate reasoners according to the example connections and the profile for COROR:

$$\mathrm{RULE}_{\mathrm{COROR}} = \{jena\}$$

and

$$\mathrm{RULE}_{\mathrm{COROR}} \cap \{swrl, jena\} = \{jena\} \neq \varnothing.$$

Furthermore for COROR

$$\mathrm{TYPE}_{\mathrm{COROR}} = \{entailment\}$$

and

$$\mathrm{TYPE}_{\mathrm{COROR}} \cap \{entailment, resolution\} = \{entailment\} \neq \varnothing.$$

Hence the AC *rule required* is satisfied. It can be deduced in similar ways that the ACs

platform and *concrete domains* are also satisfied. The satisfaction rate for COROR is then computed following the formula given in section 4.2, as

$$Satisfaction\_Rate = \frac{3}{3} \times 100\% = 100\%$$

and then COROR is identified by RESP as the most appropriate reasoner for the above semantic sensor network management system since its 100% satisfaction rate. Using RESP, Bob needs not to search for and read technical documents about reasoners and how different reasoning technologies match different application characteristics such as which reasoner supports rule reasoning, which reasoners supports J2ME CLDC 1.1 and which reasoner supports computation and comparison XSD datatype and so on. RESP can automatically select a most appropriate reasoner for him through matchmaking even if he has little knowledge of OWL reasoning.

4.5 *Summary*

This chapter presents a computer aided reasoner selection process, RESP, which is designed to address a problem raised by the rapid development and complexity in both OWL reasoning technologies and semantic applications: without a process to select an appropriate reasoner, such rapid advancement in semantic applications and reasoning technologies will require more effort to be devoted by application developers and reasoner experts to select an appropriate reasoner for semantic applications. RESP allows application developers to select an appropriate OWL reasoner for their applications by inputting only ACs of which they should be familiar.

Discussions of the interplays between semantic applications and reasoners are presented in terms of 11 reasoning-related aspects. Example candidate ACs and connections are identified from the discussions and an artificial use case is presented using the identified ACs and connections to show the using of RESP to select an appropriate reasoner.

This section approaches the first half (design of RESP) of the research objective 4 as to

"*design and implement a reasoner selection process enabling automatic/semi-automatic reasoner selection for applications*."

A description of a prototype implementation of RESP can be found in the next chapter. It is implemented based on the identified ACs, RCs and connections discussed in this chapter, and it is targeted at the second half of the research objective 4 (implementation of RESP).

RESP has been identified as the minor contribution of this thesis.

Chapter 5
Implementation

5.1 *Introduction*

As discussed earlier in the introduction chapter, the increasing demand for intelligence in embedded devices calls for resource-constrained OWL reasoners [Kleemann and Sinner 2006, Brennan et al 2009, Koziuk et al, 2008]. In order to use fewer resources on resource-constrained devices, the idea of reasoner composition is proposed in this thesis enabling reasoners to adjust their reasoning capabilities/algorithms according to the characteristics of applications, such that unnecessary reasoning capabilities are not loaded, reducing the resource usage. Some (relatively static) reasoner composition mechanisms are already out there, such as to manually add/remove rule set or to generate translate ontology into rules according to pre-defined rule patterns. However, the highly dynamic nature of some embedded systems makes the static reasoner composition mechanisms insufficient, which then raises the problem of designing automatic reasoner composition mechanisms. Furthermore, as mentioned in the introduction chapter, the ever growing of complexity and volume of reasoner characteristics and application characteristics gradually makes the existing consultation-based insufficient and hence raises another problem of designing an automatic reasoner selection process.

These two problems are separately discussed in the Chapter 3 and Chapter 4, and two novel tools are designed. In Chapter 3 the design of COROR, an automatic composable rule-entailment reasoner for resource-constrained devices, is presented in correspond to the first problem. Two novel automatic reasoner composition approaches are introduced to automatically compose rule-entailment reasoners both at the rule set level and inside the RETE algorithm according to the ontology to be reasoned. To address the second problem as to construct an automatic reasoner selection process, Chapter 4 presents the design of RESP,

an automatic reasoner selection process that allows application developers to select an appropriate reasoner for their applications according to the application characteristics of their applications.

Prototype implementation of the above two tools are described in detail in this section. The implementation of COROR is discussed with respect to five major aspects, namely: the selection of an appropriate resource-constrained platform on which COROR will be implemented; the selection/construction of a resource-constrained rule-entailment reasoner based on which the designed composition algorithms are implemented; the implementation of the pD* entailment rules using Jena rule format; the implementation detail for the two novel composition algorithms; and finally the extension of COROR to support OWL 2 from the implementation perspective (please refer to section 3.5 for the discussion of extending COROR to support OWL 2 from the design perspective). A desktop prototype implementation of RESP, which is called Tool for Automatic Reasoner Selection (TARS), is also presented in the subsequent sections. Since it is implemented for demonstration and evaluation purposes rather than for practical use, the RCs, example candidate ACs, and connections as identified above in section 4.3 are used.

5.2 *COROR*

This section describes in detail how COROR is implemented.

5.2.1 Choosing a Platform

The Sun SPOT [SUN SPOT 2010] platform is chosen as the platform on which COROR is implemented. It is designed to encourage the development of new embedded applications and therefore everything is well integrated. It includes a sensor board with a resource-constrained hardware platform: a 180MHz 32-bit ARM920T core processor, 512K RAM and 4M Flash. Furthermore it has a well-integrated top-to-bottom Java software programming platform which is the de facto platform for resource-constrained devices such as sensors or medical devices and so on. It runs a Squawk Java Virtual Machine (JVM) that supports J2ME CLDC 1.1, which is a platform adopted by many very limited resource-constrained devices. Furthermore Sun SPOT is powered by batteries, and multiple Sun SPOT sensor boards can be networked via wireless communications, enabling a wireless sensor network to be constructed. Some other benefits of using Sun SPOT include that it is well integrated with Netbeans java Integrated Development Environment (IDE) facilitating java development and it comes with an emulator allowing applications to be debugged and tested on it before physically deployed to the real sensor board. All these show that Sun

SPOT matches the target platform of the resource-constrained composable reasoner research carried out in this thesis, and therefore it appears to be a perfect platform for implementing and testing the COROR composable resource-constrained reasoner.

Figure 5-1: Sun SPOT wireless sensor network development kit.

COROR is implemented on the Sun SPOT [SUN SPOT 2010] sensor board emulator with SDK v4.0 (blue). It is written in Java in Netbeans 6.5. COROR is implemented to be conformant with J2ME CLDC 1.1 since it is the running platform supported by Sun SPOT and also many other small devices with very limited resources. Since J2ME CLDC 1.1 is a subset of J2ME CDC and J2SE platform, this implementation can also run on these platforms.

5.2.2 Constructing a Resource-Constrained Rule-Entailment Reasoner

Considering (1) the large amount of effort required designing and developing a resource-

constrained rule-entailment OWL reasoner from scratch, and (2) the major objective of this research as to investigate the performance impacts brought by the application of the composition algorithms (refer to objective 3), it would be much easier to implement COROR by combining the two novel composition algorithms into an existing resource-constrained rule-entailment OWL reasoner rather than from scratch. Since an investigation showed that no proper off-the-shelf resource-constrained rule-entailment reasoner is available, migration was required to port a proper existing desktop rule-entailment reasoner to the target platform (Sun SPOT). This section presents the effort involved in the construction of the resource-constrained rule-entailment reasoner based on which COROR was implemented.

5.2.2.1 Selecting a Proper Rule-Entailment Reasoner

Note that several requirements were to be imposed on the selected reasoner. Firstly, the selected reasoner needs to be a *typical* rule-entailment reasoner using RETE algorithm and ontology-independent translation, since (1) rule-entailment reasoners are identified as the most suitable type of reasoner to carry out the reasoner composition research (refer back to the introduction section of Chapter 3), and (2) the use of a typical rule-entailment reasoner enables a grounding for the general applicability of the two novel reasoner composition algorithms. Secondly, the selected reasoner needs to be open source since it is highly likely that the implementation of the composition algorithms, in particular the *two-phase RETE* algorithm, would require changes to the original reasoning algorithm. Thirdly the availability of well written documentation and an active developers/user group are important for the author to use less time to investigate the intricacies of the reasoner.

Seven rule-entailment reasoners, including five state of the art desktop rule-entailment reasoners: OWL2Jess, OWLJessKB, BaseVISor, swiftOWLIM, and Jena (Although Jena is categorized as a hybrid reasoner, it has a well-built RETE engine which allows forward-chaining rule-entailment OWL reasoning to be performed.); and two mobile rule-entailment reasoners: MiRE4OWL and Bossam, are investigated for their suitability in accordance to the above requirements. Among these reasoners, Jena shows better suitability than the others in all the above mentioned aspects, since (1) Jena is open source, enabling modifications to be performed on the code, (2) it has a typical and well-implemented RETE engine, providing a good basis for the implementation of the two novel reasoner composition algorithms, and (3) it is written in Java (J2SE) and there is also an off-the-shelf reduced Jena framework for mobile devices available (i.e. μJena, which is written to run on J2ME CDC), which largely reduces the efforts to port it to a J2ME CLDC platform. In addition Jena (and

μJena) has a rich set of interface enabling ontology manipulation and accessing, and it is well supported and documented, further simplifying the implementation.

BaseVISor, swiftOWLIM and Bossam are closed source reasoners. Although MiRE4OWL is a mobile reasoner, it is not publically accessible and it is implemented using C++, and would involve a much more difficult migration. OWLJessKB and OWL2Jess are open source (OWL2Jess is only a XSLT document translating OWL to JESS program), but they both rely on the JESS general rule engine [JESS] and therefore there would be a need to migrate the JESS engine to J2ME platforms. After investigation, it was discovered that this migration would be much difficult than Jena.

5.2.2.2 Jena RETE Engine

Since Jena is selected as the reasoner to perform the migration, a study was carried out into the inside of Jena RETE engine. This section describes the Jena RETE engine and its implementation features.

As mentioned earlier Jena has a RETE engine. OWL ontology is viewed in this RETE engine as a *RDF graph*, i.e. a set of triples of the format

(subject predicate object).

Each triple is a fact for the RETE engine. A set of OWL entailment rules written in Jena rule format is used to match the RDF graph according to OWL semantics. Two types of elements can exist in the l.h.s. of a Jena rule: a condition element and a functor. A normal condition matches triples and it has the same format as a triple except that the subject, predicate and object can be variables. A functor has different purposes. It can either be a builtin that performs actions, such as: assigning anonymous nodes; performing literal checking; performing mathematical operations; and so on, or be an embedded structure for caching matched subgraphs. As will be found out in the section 5.2.3, since functors are only used as builtins in the used rule set, the usage as embedded structure is ignored here.

Two sample Jena rules are given for illustration. The rule rdfs4b infers that any non-literal subject is a RDFS resource. The functor notLiteral(*?w*) is responsible for checking if *?w* is a literal or not.

[rdfs4b: (*?v ?p ?w*), notLiteral(*?w*) → (*?w* rdf:type rdfs:Resource)].

The rule rdfp15 infers the class of a property value when the property is restricted by an

OWL someValuesFrom restriction:

[rdfp15: (*?v* owl:someValuesFrom *?w*), (*?v* owl:onProperty *?p*), (*?u ?p ?x*), (*?x* rdf:type *?w*) → (*?u* rdf:type *?v*)].

The left hand side of this rule consists of four normal conditions but no functors.

The Jena RETE engine has some features that make it an efficient OWL reasoner.

First each intermediate result is represented as an array of the same size with the number of variables in a particular rule, and each entry of the array corresponds to the value bound to a variable. For example the intermediate results generated for the rule rdfp15 (as illustrated above) is a five-entry array that looks like that represented in Figure 2.2.

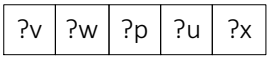


Figure 5-2: An example intermediate result for rule rdfp15

Each entry in this array corresponds to a variable in the rule and they are arranged in the same order as appeared in rule, as illustrated in Figure 5-2. This approach can speed up joins: joining two intermediate results turns out to be checking for the consistency of values in the same position of the corresponding intermediate values. For example, joining the first two conditions of the rule rdfp15, that is (*?v* owl:someValuesFrom *?w*) and (*?v* owl:onProperty *?p*), turns out to check the consistency of the value in the first entry, i.e. ?v, of the intermediate results from them. As will be discussed later on in section 5.2.4, this mechanism needs special attention since it hinders node sharing.

The second reason Jena is quite efficient is that triples are indexed according to the predicate used to speed up searching. The third reason is that as triple is the only type of fact, no type checking is needed for condition elements. However no other optimizations are applied to change the structure, construction process, or execution of the RETE network, making the Jena RETE engine a relatively typical and straight forward RETE engine for rule-entailment reasoner.

Figure 5-3 illustrates an example Jena RETE network for the rule rdfp15 as listed above. As

all facts are in triple format and therefore no type checking is needed for condition elements. Intermediate results are represented as n-array as described in Figure 5-2. Bound variables in intermediate results are coloured in green and yellow but unbound variables are coloured in grey with a cross in centre. Join operations are illustrated in each beta node as an equation(s) of the variable to be checked for consistency, e.g. in Beta Node 1 *?v=?v* means the variable *?v* is to be checked for consistency, and the corresponding bound variables are coloured in yellow and linked using dotted lines. The intermediate results come from the last beta node (in this example the beta node 3) pass the consistent checking for all variable bindings. It is then propagated to the functor node attached in the end, performing builtin operations such as number comparison, checking for variable bindings, etc. In this example no functors are exist therefore the functor set is empty. Intermediate results passing functor actions are then made to be the instantiations of the rule and are inserted into the conflict set waiting for firing.

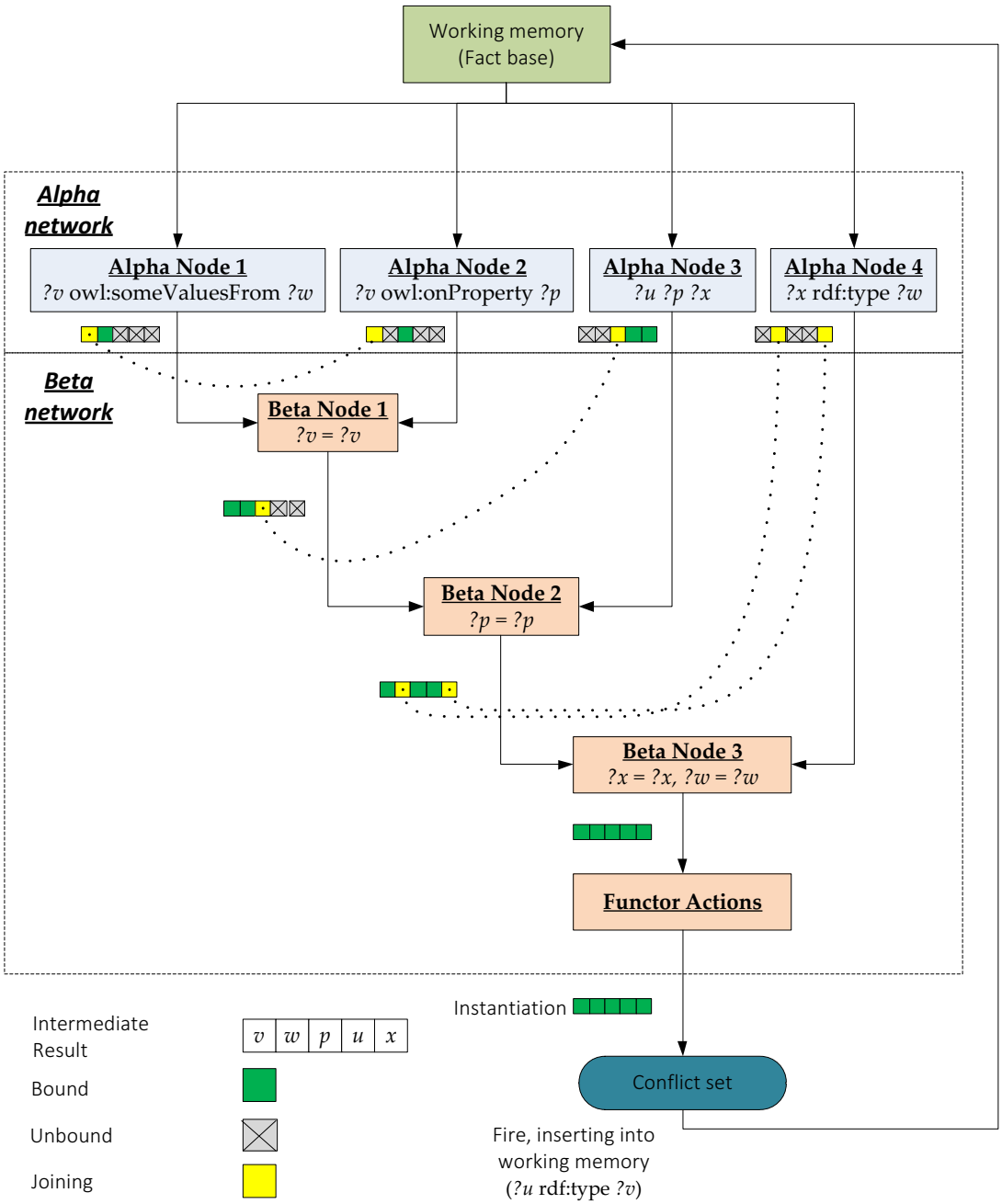


Figure 5-3: An example Jena RETE network and an illustration of join operations

The optimizations adopted in Jena RETE engine make it an efficient OWL reasoner. However their application does not change the structure of RETE network or execution process of the RETE algorithm, hence making the Jena RETE engine still a typical RETE engine compliant with the original RETE algorithm. Also considering its open-source nature, Jena RETE engine forms a good target for implementing COROR.

5.2.2.3 Implementing OWL Reasoning on µJena

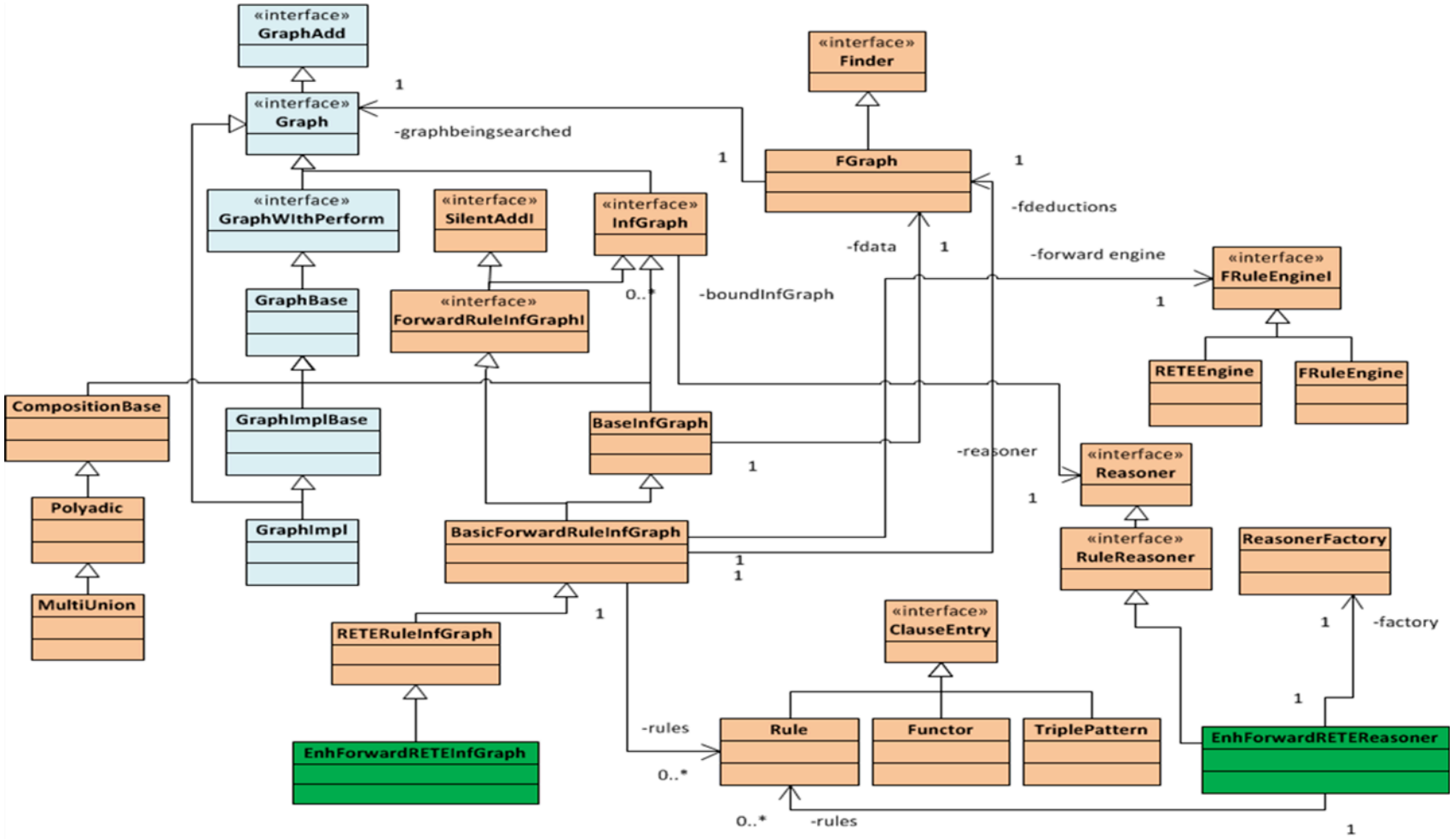
Based on the investigation carried out, Jena v2.5.5 and µJena v1.5 were thus selected to construct a resource-constrained rule-entailment OWL reasoner as a basis on which the implementation of COROR is carried out. However, a major problem that had to be addressed before µJena ran on Sun SPOT (J2ME CLDC platform) as a resource-constrained rule-entailment reasoner, was that µJena is only an ontology manipulation framework and the reasoning features of Jena were not included. Given the close connections between µJena and Jena, µJena was thus extended by the author to use the Jena RETE engine. The extension mainly focuses on the migration of classes from 12 packages of over 90 classes and over 10000 lines of code, which enlarged the original µJena source code by around 25% in size. Most classes are migrated from the corresponding classes in Jena and 9 class are introduced by the author in order for invoking forward chaining reasoning (*EnhForwardRETEInfGraph*, *EnhForwardRETEReasoner*), providing builtin support for operations in pD* rules (*LiteralStore*, *AssignAnon*, *IsDLiteral*, *IsPLiteral*), and some utils classes (*Character*, *Collection*, *NumberUtil*). Four major packages are *com.hp.hpl.jena.reasoner*, *com.hp.hpl.jena.reasoner.rulesys*, *com.hp.hpl.jena.reasoner.rulesys.builtins* and *com.hp.hpl.jena.reasoner.rulesys.impl,* where the major extension occurred. Since *graph* is the core structure in Jena (µJena) where ontology is stored and the other operations, including manipulation and reasoning, are performed, its extension is discussed in detail.

A class diagram is given in Figure 5-4 showing the extension made to µJena in order to support OWL reasoning. To avoid an over complicated diagram, not all introduced or migrated classes are included in the diagram. Only classes related to graph are shown. A full list of the introduced and migrated classes can be found in Appendix E.

Originally µJena only brings in six basic graph classes, as coloured in light blue in Figure 5-4. To enable OWL reasoning, the *InfGraph* interface, i.e. an interface extending the *Graph* interface with methods to call the RETE engine and query the inferred graph, and related classes are introduced into µJena, as coloured in orange. The *BaseInfGraph* class is a base level implementation of *InfGraph*. The *BasicForwardRuleInfGraph* class extends *BaseInfGraph* by rendering forward-chaining features. For example, it materializes all inference results in a deduction graph; furthermore, it maintains a simple forward rule chainer (*FRuleEngine*), and runs it each time a new triple is added. The *RETERuleInfGraph* class extends the *BasicForwardRuleInfGraph* by replacing the simple rule chainer with a RETE engine (*RETEEngine*). The *Reasoner* interface includes the methods that all reasoners

need to conform. The *RuleReasoner* interface extends *Reasoner* with two methods that rule-based reasoners need to conform, i.e. *getRules()* and *setRules()*. Jena does not have a dedicated RETE reasoner class. As a matter of fact, in order to use only RETE engine to perform OWL reasoning, Jena needs to configure the *GenericRuleReasoner* class using the FORWARD_RETE option. Other options include FORWARD (using a simple forward-chaining engine), BACKWARD (using a backward resolution engine) and HYBRID (using both the RETE engine and the backward resolution engine). However the introduction of the *GenericRuleReasoner* class requires the introduction of many more classes into μJena, such as forward-backward inference graph, backward chaining engines, and so on, and they are not used for this research. Therefore an *EnhForwardRETEReasoner* class and an *EnhForwardRETEInfGraph* are constructed to perform only forward-chaining RETE reasoning. It is based on *GenericRuleReasoner* by removing the options and Java code for the other reasoners and the java code for the forward RETE reasoner is kept. By doing this only *RETERuleInfGraph* is returned for ontology binding. Some other classes are also introduced such as *Rule*, *MultiUnion* and so on. Since they are auxilirary classes in order to support reasoning, they are not discussed here in detail.

The migration of the reasoning-related classes from Jena (J2SE platform) to μJena (J2ME CLDC 1.1 platform) involved a great deal of code refactoring, especially the replacement of J2SE conformant container classes with the corresponding CLDC 1.1 conformant container classes defined by μJena: *java.util.List* and its descendants such as *ArrayList* are replaced by *it.polimi.elet.contextaddict.microjena.util.List*. Similarly *java.util.Set* and its descendants such as *HashSet* are replaced by *it.polimi.elet.contextaddict.microjena.util.Set*, and *java.util.Map* and its descendants such as *HashMap* are replaced by *it.polimi.elet.contextaddict.microjena.util.Map*. The replacement container classes, namely *it.polimi.elet.contextaddict.microjena.util.List*, *it.polimi.elet.contextaddict.microjena.util.Set*, and *it.polimi.elet.contextaddict.microjena.util.Map*, are originally used in μJena and therefore their correct functioning is ensured. No other changes except for the replacement of container classes were performed on the migrated classes and hence the correctness of the migrated classes is ensured.


«interface»
GraphAdd
«interface»
Graph
«interface»
Finder
FGraph
-graphbeingsearched
1
«interface»
GraphWithPerform
«interface»
SilentAddI
«interface»
InfGraph
-fdata
-fdeductions
-forward engine
«interface»
FRuleEngineI
GraphBase
«interface»
ForwardRuleInfGraphI
0..*
-boundInfGraph
RETEEngine
FRuleEngine
CompositionBase
GraphImplBase
BaseInfGraph
-reasoner
«interface»
Reasoner
Polyadic
GraphImpl
BasicForwardRuleInfGraph
«interface»
RuleReasoner
ReasonerFactory
MultiUnion
RETERuleInfGraph
«interface»
ClauseEntry
-factory
-rules
Rule
Functor
TriplePattern
EnhForwardRETEReasoner
EnhForwardRETEInfGraph

For clarity the μJena version after being extended with reasoning capabilities is termed as *enhanced μJena*. Entailment is the key reasoning task for *enhanced μJena*. Conjunctive queries are not yet supported by *enhanced μJena* since *μJena* lacks support for it. Since adding such functionality is a pure implementation issue involving a large amount of code work, however, without (positively/negatively) affecting the composition algorithms (the evaluation of conjunctive queries will be performed in a different module separate from the reasoning algorithm), its addition can be taken as a future work. The *enhanced μJena* only supports a single-triple-based query mechanism through the API, i.e.

```
InfModel.listStatement(s, p, o),
```

where s, p and o correspond to the subject, predicate and object of the triple pattern being queried. Since COROR only extends the enhanced *μJena* by combining the two novel composition algorithms rather than adding more reasoning tasks or the conjunctive query answering ability, the entailment and single-triple-based query are respectively the key reasoning task and the only query mechanism of COROR.

However, some common reasoning tasks can be realized by querying the ontology with all entailments calculated (entailment closure) using the above single-triple-based query mechanism. For example: checking subsumption between two classes *C* and *D* can be reduced to querying the entailment closure with the triple (*C* rdfs:subClassOf *D)*; checking instantiation of *C* as querying with the triple (*?x* rdf:type *C)*, where *?x* is a variable (represented in Jena triple-based query as null); checking satisfiability of a class *C* as querying with the triple (*C* rdfs:subClassOf Nothing); instance checking *a:C* as querying with the triple (*a* rdf:type *C)*; and so on.

For some other reasoning tasks the reduction is non-trivial and requires some codework. For example checking for a type of P-clash can be reduced to querying the result ontology with the triple:

?x owl:differentFrom ?*y*,

and for every pair of (*?x, ?y*) in the results, checking for

?x owl:sameAs *?y*

Successful query (true is returned) then indicates a P-clash. Another example could be the

realization of an instance *a*. It requires finding the most specialized class C that *a*:C. This needs pairwise subsumption checking for all classes retrieved using (*a* rdf:type *?x*). Since the above reasoning tasks can be reduced to querying a fully entailed ontology, entailment is then the key reasoning task and its performance becomes the major factor determining the performance of these reasoning tasks. Considering their little relevance to the research question and the amount of codework required, these reasoning tasks are not implemented. However extending the *enhanced μJena* to support more reasoning tasks and complex queries will be considered in the future work to enable COROR for practical usage.

Manipulation of unreasoned ontologies is supported through the corresponding μJena APIs in either OWL style or triple style. Some common operations include add/delete statement(s), create (typed/plain) literal/property/resource and single-triple-based querying. For a reasoned ontology only three operations are supported, including triple-based addition, deletion, and searching. Addition is handled incrementally due to the use of the RETE algorithm and all subsequently inferred triples of the addition are inserted into the reasoned ontology. However since Jena RETE engine re-reasons the entire ontology for deletion, incremental deletion is not supported by the *enhanced μJena*, and therefore not available to COROR. This will be considered in the future research.

Some other features of the *enhanced μJena* are described in this paragraph. *μJena* only supports reading and parsing OWL ontology in the N-TRIPLE format[13], and hence so it is with the *enhanced μJena* and COROR. The *enhanced μJena* inherits the six XSD datatypes supported by *μJena*, which are xsd:float, xsd:double, xsd:int, xsd:long, xsd:integer, xsd:boolean, and xsd:string. The validation of datatype values is performed in *μJena* at the ontology loading time for encountered literal node of the above XSD datatypes in the ontology. Jena rules also provide a set of builtins to process string/number values (concrete domain objects) and to check for the bound/unbound of a variable (closed-world feature). Furthermore *μJena* has a datatype registry enabling user-defined datatypes to be constructed and used.

To summarise, in this paragraph the reasoner characteristics of the *enhanced μJena* are presented. The *enhanced μJena* extends the original *μJena* with the Jena RETE engine and the relevant classes, making it a resource-constrained rule-entailment OWL reasoner for J2ME CLDC 1.1 devices. Entailment and single-triple-based queries are respectively the

[13] http://www.w3.org/TR/rdf-testcases/#ntriples

key reasoning task and query mechanism of *enhanced μJena*. Some other reasoning tasks such as subsumption, instantiation, satisfiability and instance checking can be achieved by directly posing single-triple-based queries on the fully entailed ontology. Six XSD datatypes are supported and a datatype registry is available enabling users to define their own datatypes. Computations/comparisons of these datatypes are also supported through rule builtins. The migration of Jena rule handling classes enables Jena rules to be interpreted. This enables not only OWL inference rules but also user-specific rules to be modelled and handled in *enhanced μJena*. The ability to check if a rule variable is bound/unbound enables a closed-world taste in the supported Jena rules. A rich set of ontology manipulation APIs is provided by *μJena* to handle ontologies. Some reasoner characteristics are not yet supported since they are not relevant to the research question of this thesis and the significant codework involved. Explanation, conjunctive queries, and database are not yet supported.

5.2.3 Implementing the pD* Semantics

The *enhanced μJena* enables rules written in the Jena rule format to be loaded from a text file as streams. The following Java code snippet shows how rules are loaded and parsed. The variable `ruleSet` is a Java *String* pointing to the location of the rule file. Rules are stored in the reasoner as a list of *Rule* instances.

```
/* construct a buffered reader pointing to the rule file.*/
BufferedReader br = new BufferedReader(new InputStreamReader(
       this.getClass().getResourceAsStream(ruleSet)));

/* load and parse the rules.*/
List rules = Rule.parseRules(Rule.rulesParserFromReader(br));

/* construct a forward chaining RETE reasoner according to the loaded
rules.*/
Reasoner reasoner = new EnhForwardRETEReasoner(rules);
```

Therefore the pD* entailment rules needed to be implemented using Jena rules in order to load them in COROR. Since the pD* entailment rules are originally given in triple format (as given in Appendix C), this implementation was then a direct translation from pD* entailments to the Jena rules. For example, given G represents the ontology graph to be reasoned, the pD* entailment rule rdfp2 (as in Table 5-1)

Table 5-1: pD* entailment rule rdfp2.

Rule	If G Contains	Where	Then add to G
rdfp2	*p* type InverseFunctionalProperty *u p w*		*u* sameAs *v*

v p w

is directly translated into Jena rule as

[rdfp2: (*?p* rdf:type owl:InverseFunctionalProperty), (*?u ?p ?w*), (*?v ?p ?w*) -> (*?u* owl:sameAs *?v*)].

In general, the translation takes triple patterns in the "if G Contains" column as normal conditions in the l.h.s. of the Jena rule and conditions in the "where" column as functors. The triple patterns in the "Then add to G" is then transliterated as elements in the r.h.s. of the Jena rule. Variables and OWL constructs are preserved. However, a question mark (i.e. a *?*) is added to the front of each variable and the corresponding namespace is added in front of every OWL/RDFS construct, making them Jena rule conformant. Four built-in functors are constructed to check the conditions given in the "where" column: they are isPLiteral(), isDLiteral(), assignAnon(), and notLiteral(). Descriptions for each built-in are given in Table 5-2.

Table 5-2: Descriptions of built-in functors

Functor	**Description**
isPLiteral(?*l*)	Check if *l* is a plain literal.
isDLiteral(?*l, ?t*)	Check if *l* is a well-typed datatype literal. If it is the datatype of *l* is bound to *t*.
notLiteral(*?w*)	Check if *w* is not a literal (including plain literal and datatype literal).
assignAnon(?*l*, ?*b*)	Check if a literal *l* has not yet been assigned to an anonymous node. If yes a new anonymous node is assigned and is bound to ?*b*, otherwise the previously assigned anonymous node is retrieved and bound to ?*b*.

Functors are constructed by extending the *Functor* class. The actions of a functor is realized by rewritten the method *bodyCall()* declared in the *Functor* class.

In all 39 entailment rules are implemented, including 16 D* entailment rules and 23 P entailment rules (for a full set of pD* rules in Jena rule format please refer to Appendix C). Some modifications are made to the pD* entailment rule set while implementing them.

A first modification is that in order to reduce the number of rules the rule lg is combined with the rule rdfs1 and rdf2D forming 2 combined rules, i.e. lg-rdfs1 and lg-rdf2D:

[lg-rdfs1: (*?v ?p ?l*), assignAnon(*?l, ?b*), isPLiteral(*?l*) → (*?v ?p ?b*), (*?b* rdf:type rdfs:Literal)]

[lg-rdf2D: (*?v ?p ?l*), assignAnon(*?l, ?b*), isDLiteral(*?l, ?t*) → (*?v ?p ?b*), (*?b* rdf:type *?t*)]

Before the correctness of lg-rdfs1 and lg-rdf2D is discussed the definitions of the rule lg, rdfs1 and rdf2-D are presented (Table 5-3). Suppose that L represents a set of all literals, L_p represents plain literals and L_D^+ represents well-formed literals. The rule lg prescribes for every triple (*v p l*) in a RDF graph G, where is a literal to which a blank node has never been assigned, a new blank node, represented as b_l, is constructed and assigned to it. Otherwise the previously assigned blank node is assigned. The rule rdfs1 assumes that for any plain literal *l* in a triple (*v p l*) a new triple (b_l type Literal) is added. The rule rdf2-D generates a new triple (b_l type *a*) for well-formed datatype literal *l* in triple (*v p l*).

Table 5-3: Definitions of lg, rdfs1 and rdf2-D in pD* entailments

Rule	If G Contains	Where	Then add to G
lg	*v p l*	$l \in L$	*v p* b_l
rdfs1	*v p l*	$l \in L_P$	b_l type Literal
rdf2-D	*v p l*	$l = (s, a) \in L_D^+$	b_l type *a*

Here the correctness of lg-rdfs1 and lg-rdf2D is discussed. According to descriptions given in Table 5-3 the built-in assignAnon assign an anonymous node to all literals. Therefore the rule lg-rdfs1 assigns an anonymous node *b* to all literals in *L* and if the literal is a plain literal two triples, *v p b* and *b* type Literal, are added into the graph. Similarly the rule lg-rdf2D assigns a blank node, *b,* to all literals it encounters and if *l* is a well-typed literal of type *t*, two triples, *v p b*, *b* type *t*, are added into the RDF graph. The rule lg-rdfs1 and lg-rdf2D cover the rule rdfs1 and rdf2-D but part of the semantics in the rule lg is missing: the

triple *v p b* will not be added into the graph when *l* is an ill-typed datatype literal, otherwise will lead to a D-clash. However since it is presumed the ontology is clash-free and therefore the missing part of the semantics will not be needed for ontology reasoned under this presumption.

A second modification is the removal of the rule gl (as indicated in Table 5-4). The rule gl is a reverse application of the rule lg. Since in practice all blank nodes are allocated through assignAnon() by lg-rdfs1 and lg-rdf2D, therefore there must already exist (*v p l*) where *l* is a Literal (the condition (*?v ?p ?l*) needs to be matched before assignAnon() can assign a blank node b_l), therefore there is no need to reversely add (*v p l*) into the ontology graph.

Table 5-4: pD* entailment rule gl

Rule	If G Contains	Where	Then add to G
gl	$v\ p\ b_l$	$l \in L$	*v p l*

As discussed in section 5.2.2, consistency checking is not implemented in the *enhanced μJena* (and COROR) and hence XML-clash and P-clash are not detected. This is reasonable due to the significant codework involved and the little relevance of detecting these inconsistencies to the research question: detecting XML-clash does not require ontology reasoning; and as *μJena* does not natively support for XML, an XML processing component needs to be implemented or an third-party XML library needs to be incorporated into *enhanced μJena* in order to handle XML-clash, causing extra complexity in code and more resources required; checking for P-clash requires reasoning but as discussed in the previous section it can be reduced to querying a fully entailed ontology and therefore in *enhanced μJena* (and COROR) it is highly dependent on entailment computation which is the key reasoning task of *enhanced μJena* (and COROR). D-clash is limitedly supported by μJena since μJena checks validation for the six supported XSD datatypes at ontology loading time. In fact COROR is not the first resource-constrained reasoner that does not implement consistency checking: μOR also does not provide consistency support and SwiftOWLIM (v3.0.10) does not include pD* consistency rules. However to provide with full support of detecting XML-clash, D-clash and P-clash on COROR will be considered in the future in order to make COROR for more practical usage.

Considering COROR will work in resource-constrained environment, RDF/RDFS axiomatic triples and P axiomatic triples (can be found in [ter Horst 2005a]) are not included in this implementation to reduce the amount of inferred triples. This follows on from some of the other previous work where, for practical or efficiency reasons, axiomatic triples are sometimes removed. The work in [Hogan et al 2009] also removed axiomatic triples from the supported semantics to reduce the reasoning output so as to achieve web scale reasoning. OWL 2 RL does not include RDF/RDFS axiomatic triples and OWL axiomatic triples in order to avoid performance problems in practice

Rules are plain text encoded in a specific rule file, giving users more flexibility to modify the rule set and also authoring application-specific rules.

5.2.4 Implementing the Composition Algorithms

This section presents how the two composition algorithms, i.e. the *selective rule loading* algorithm and the *two-phase RETE* algorithm, are implemented in COROR. Four different composition modes are designed to allow the use of the corresponding composition algorithm. The *noncomposable* mode uses original Jena RETE engine (therefore no composition algorithms is applied). The three composable modes are the *selective rule loading* mode that uses the *selective rule loading* algorithm, the *two-phase RETE* mode that uses the *two-phase RETE* algorithm and the *hybrid* mode that uses the hybrid algorithm. Note that for clarity and brevity in the following text these COROR modes are respectively termed as COROR-noncomposable, COROR-selective, COROR-two-phase, and COROR-hybrid. Note, although modifications are made to enable the composition algorithms to work on Jena RETE engine, the capabilities to turn off the composition algorithms are included such that in the noncomposable mode the original Jena RETE engine is used.

The implementation of the *two-phase RETE* algorithm and the *selective rule loading* algorithm mainly happened in the RETE engine. It mainly involved modifications of the RETEEngine (the flow of RETE algorithm), RETEClauseFilter (alpha node) and RETEQueue (beta node). Three new classes were constructed in order to support condition node sharing, namely *RETESibling*, *RETEClauseFilterSharing*, and *IntermediateBindingVector*.

5.2.4.1 Selective Rule Loading Algorithm

The *selective rule loading* algorithm is implemented in the *RuleSetComposer* class in the package *ie.tcd.cs.nembes.microjenaenh.reasoner.rulesys.enh*. Figure 5-5 shows in general the components involved in order to construct a selective rule set. In brief it analyses the

ontology for OWL constructs and then uses the rule-construct graphs (coded in the form of mappings) to construct a selected rule set for the reasoner to initialize.

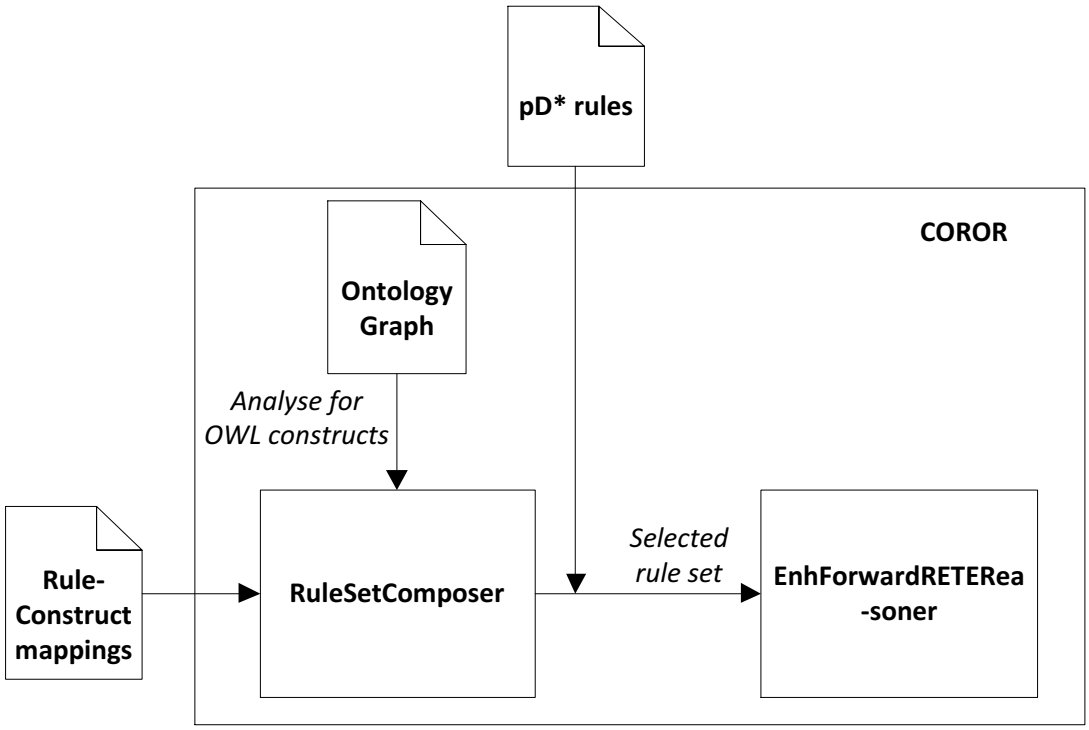


Figure 5-5: Implementation of the selective rule loading algorithm.

Analysing the ontology graph for the contained OWL constructs is implemented by enumerating all pD* expressivity constructs and querying the ontology for each of them by using the method `Model::contains()` to determine if a triple pattern is included in the ontology. For example, the code snippet

```
if(model.contains(null, OWL.inverseOf)) {
   ontSignature.add(owl+OWL.inverseOf.getLocalName());
}
```

checks if the ontology contains a triple pattern such as (x owl:inverseOf y) where x and y match arbitrary RDF nodes. Since owl:inverseOf will only appear in the predicate position if it is used to state the inverse of two properties, the existence of the above triple pattern then proves the existence of owl:inverseOf. All contained OWL constructs are stored in a list *ontSignature* for constructing a selective rule set.

The *rule-construct mappings* represent rule-construct dependency graphs coded in plain-text.

Each mapping represents the dependency relationships of a rule. The format of mapping is specified here by means of a BNF-like notation:

Mapping:=***rule-name***':'***semantic-level***':['***premises***']->['***consequences***']'

semantic-level:='rdfs'|'owl-lite'|'owl-dl'

premises:=''|***premise***{','***premise***}

consequences:=''|***consequence***{','***consequence***}

premise:= ***pD_expressivity_constructs***

consequence:=***pD_expressivity_constructs***

The field *rule-name* and *semantic-level* correspondingly represent the name of the rule and the semantic level into which this rule falls. They exist for each mapping. As the *rule-name* is used to construct the selective rule set, it needs to be exactly the same as the name of the rule in the rule set. The field semantic-level is for COROR to select rules according OWL sublanguages. The field *premises* and *consequences* are correspondingly the premises and consequences of the rule. They may contain no, one or multiple premises or consequences. A *premise* (*consequence*) is defined as any construct in the expressivity construct set of the pD* entailment rules. Some example rule-construct mappings are given:

rdfp13a:owl-lite:[owl:equivalentProperty]->[rdfs:subPropertyOf]

rdfp10:owl-lite:[rdf:Property,owl:sameAs]->[rdfs:subPropertyOf]

rdfp8bx:owl-lite:[owl:inverseOf]->[]

The loading and parsing of the rule-construct mappings is performed through RuleSetComposer::readRuleConstructs(). Each rule-construct mapping is stored in a *RuleSignature* java class instance. All rule-constructs mappings are stored in a map structure *ruleSignatures* with key as the *RuleSignature* instance for each rule and the value as a Boolean value indicating if the rule should be loaded. All values are initialized as *false*. The entire list of rule-construct mappings can be found in Appendix D and the original dependency graphs can be referred to in the section 3.4.1.

Figure 5-6 lists the algorithm for constructing a selective rule set. Before this code snippet starts, the field *ruleSignatures* is initialized with the set of rule-construct mappings, and the field *ontSignature* is initialized with OWL constructs included by the ontology, as discussed above. Then the algorithm checks according to the order of rules in *ruleSignatures* if all premises of a rule are contained by *ontSignature*. If so, set the corresponding value in *ruleSignatures* as *true* and add the consequences into *ontSignature* and re-start the selection from the front of *ruleSignatures*. Otherwise go on check the next rule in the sequence. This algorithm iteratively selects rules until all rules are checked and no new construct is added into *ontSignature*. The appropriate rule set is then constructed and loaded into the reasoner.

Note that not all selected rules are guaranteed to fire, as the presence of premises does not necessitate successful instantiation of the rule. However, unselected rules will definitely not fire even if they were loaded due to the absence of premises in the ontology.

```
boolean hasNewConstructs = false;
int s = ruleSignatures.size();
Set ruleSignaturesSet = ruleSignatures.entrySet();
do{
   hasNewConstructs = false;
   for(int i = 0; i < s; i++){
      Entry entry = (Entry)ruleSignaturesSet.get(i);
      if(entry.getValue().equals(Boolean.FALSE)){
         RuleSignature rSig = (RuleSignature)entry.getKey();

       // to check if all lhs constructs are included in ontology signature.
      // to mark the corresponding rule signature as true if it is contained.
         boolean containsAll = true;
         List lhsConstructs = rSig.getLHSConstructs();
         if(lhsConstructs != null){
            for(int j = 0; j < lhsConstructs.size(); j++){
               if(!ontSignature.contains(lhsConstructs.get(j))){
                  containsAll = false;
                  break;
               }
            }
         }
         if(containsAll){
            ruleSignatures.put(rSig, new Boolean(true));
         }

         // to handle rhs constructs
         List rhsConstructs = rSig.getRHSConstructs();
         if(rhsConstructs != null){
         for(int j = 0; j < rhsConstructs.size(); j++){
            Object rhsConstruct = rhsConstructs.get(j);
            if(!ontSignature.contains(rhsConstruct)){
               hasNewConstructs = true;
               ontSignature.add(rhsConstruct);
            }
         }
      }
   }
}while(hasNewConstructs);
```

Figure 5-6: Code snippet for the constructing a selective rule set.

5.2.4.2 Two-Phase RETE Algorithm

The *two-phase RETE* algorithm is implemented in the RETE engine, which is in the Java class *RETEEngine*. Before going into further detail about the implementation, a class diagram is given that shows in detail the classes involved in the Jena RETE algorithm. In this implementation all RETE algorithm related classes are included in the package *ie.tcd.cs.nembes.microjenaenh.reasoner.rulesys.impl*.

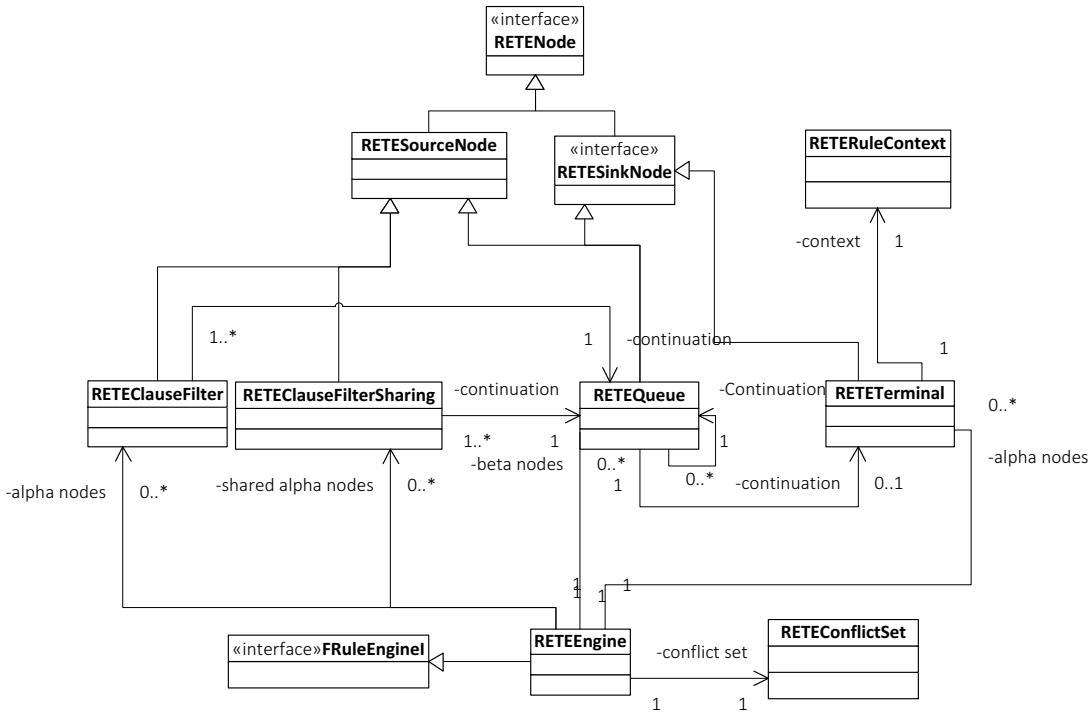


Figure 5-7: Classes related to the two-phase RETE algorithm implementation

The *RETEClauseFilter*, *RETEQueue*, *RETETerminal* and *RETEConflictSet* correspond to the alpha node, beta node and functor action node and conflict set in the RETE network given in Figure 5-3. The RETE network is connected through the *continuation* field in *RETEClauseFilter* and *RETEQueue*, it points to the next node in the RETE network. Figure 5-8 shows the code snippet for the *two-phase RETE* algorithm:

```
compileAlpha(rules, ignoreBrules);
conflictSet = new RETEConflictSet(
        new RETERuleContext(infGraph, this),
        isMonotonic);
findAndProcessAxioms();

 /*populate addspending with triples for initial matching*/
if (infGraph.getRawGraph() != null) {
    for (Iterator i = inserts.find(
              new TriplePattern(null, null, null));
              i.hasNext();) {
        addTriple((Triple)i.next(), false);
```

```
    }
}

preMatch();
applyHeuristics();
compileBeta();
crossJoinAll();
runAll();
```

Figure 5-8: Code snippet for two-phase RETE algorithm

A shared alpha network is first constructed (through `compileAlpha()`). The alpha node sharing heuristic is applied: for every non-sharable alpha node a *RETEClauseFilter* instance is constructed and for all sharable alpha nodes sharing a condition a *RETEClauseFilterSharing* instance is constructed. The implementation of this heuristic was hindered by some mechanisms used in the Jena RETE algorithm. Firstly, as discussed in section 5.2.2.2, the intermediate results used in Jena RETE algorithm vary across different rules. Therefore every time a fact matches sharable conditions from different rules, different intermediate results are generated, hindering the sharing. For example, the rule rdfs4b and rdfp15

[rdfs4b: (*?v ?p ?w*), notLiteral(*?w*) → (*?w* rdf:type rdfs:Resource)].

[rdfp15: (*?v* owl:someValuesFrom *?w*), (*?v* owl:onProperty *?p*), (*?u ?p ?x*), (*?x* rdf:type *?w*) → (*?u* rdf:type *?v*)].

have two sharable conditions, i.e. (*?v ?p ?w*) and (*?u ?p ?x*). However the intermediate results generated after they are matched to a same fact, say (ex:myCar rdf:type ex:Car), are different (as given in Figure 5-9). For the condition (*?u ?p ?x*) in rdfp15 an intermediate results of five elements are constructed. The matched fact only realizes three variables, i.e. *?p*, *?u* and *?x*, and the other two variables, i.e. *?v* and *?w*, are still blank. For the condition (*?v ?p ?w*) in rdfs4b a three element intermediate result is generated and all variables are realized. This mechanism speeds up join operations, as the consistency of variable binding is checked, by comparing variable bindings in the same position of two tokens. However it hinders the sharable conditions in different rules from sharing the alpha memory (since different intermediate results are generated).

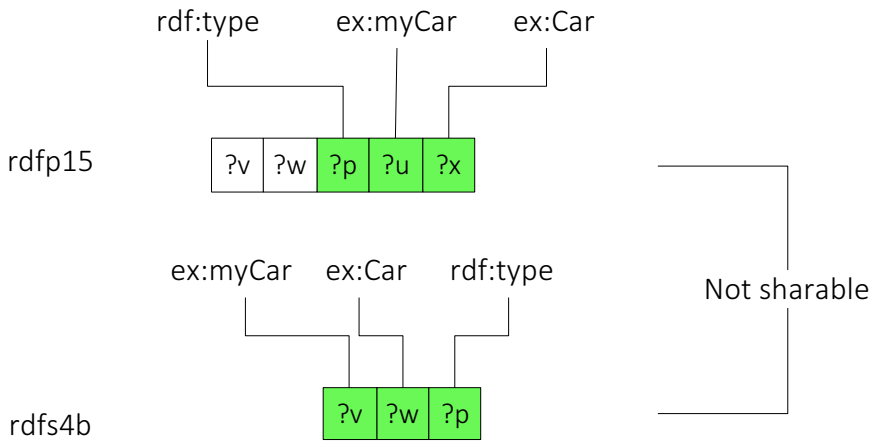


Figure 5-9: Intermediate results for the condition (*?v ?p ?w*) and (*?u ?p ?x*) in the rule rdfp15 and the rule rdfs4b.

A dual vector approach was designed by the author to separate the position information from value bindings such that condition-specific rather than rule-specific intermediate results can be constructed. The approach builds a *position vector* and a *value vector* correspondingly for storing the position information and bound values. The position information is unique for each sharable condition but the value vector can be shared among conditions. For example 4 different conditions are generated for the four conditions in rdfp15 under the dual vector approach. The value vectors for (*?v ?p ?w*) and (*?u ?p ?x*) are the same for the same matched fact and therefore facilitating the sharing of the alpha memory.

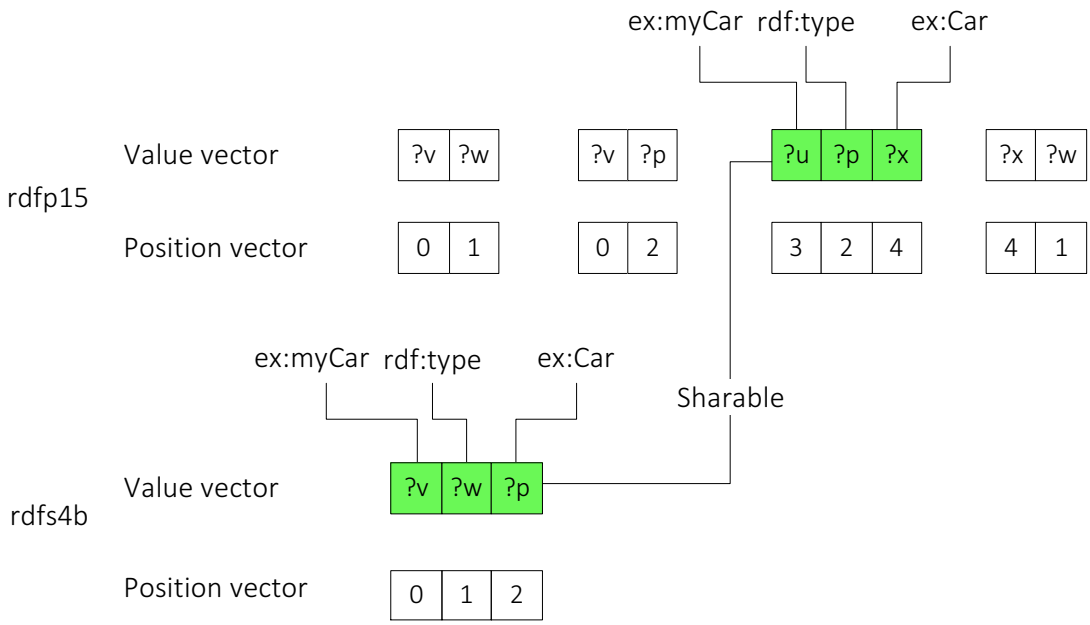


Figure 5-10: Intermediate results generated for rdfp15 and rdfs4b under the dual vector approach.

Dual vector separates the position information and value bindings, and hence it changes the join operation. Originally the join operation is comparing the value on the same position of the intermediate value from each input. Under the dual vector approach the join is then comparing the values with the same position information (if any). A new larger position vector and value vector are generated combining the two successfully joined intermediate results. The dual vector approach is implemented in *IntermediateBindingVector*.

A second problem was the different position vectors and different continuations among sharable conditions. To solve this problem a swap in/out approach is used: a class *RETESibling* is constructed for storing the position vectors and the continuations for shared conditions. Every time a particular condition is required (e.g. for performing join operation) it is restored into the *RETEClauseFilterSharing* node and the information stored in the corresponding *RETESibling* is restored as well. Therefore the *RETEClauseFilterSharing* node can work as the required condition.

On the construction of the shared alpha network, triples of the ontology graphs are then added to *addspending*, a cache of triples to be inserted into the RETE engine.

```
if (infGraph.getRawGraph() != null) {
    for (Iterator i = inserts.find(
                new TriplePattern(null, null, null));
                i.hasNext();) {
        addTriple((Triple)i.next(), false);
    }
}
```

The initial match is then performed (`preMatch()`). It removes triples from *addspending* and inject them into the RETE network (`inject()`) one by one, matching to the alpha network. Heuristics are applied by calling `applyHeuristics()`after the initial match: the information is collected according to which the most specific condition first heuristic is applied and the connectivity heuristic is applied after the application of the most specific condition first heuristic. Figure 5-11 shows the code snippet for the implementation of the most specific condition first heuristic in `applyHeuristics()`.

```
Iterator conditionIt = conditionList.iterator();
while(conditionIt.hasNext()){
   RETEClauseFilter condition = (RETEClauseFilter)conditionIt.next();

   if(newConditionList.size() == 0){
      newConditionList.add(condition);
      continue;
```

```
    }

    boolean inserted = false;
    int tripleNum = ((RETEQueue)condition.continuation).queue.size();
    for(int i = 0; i < newConditionList.size(); i++){
       if(tripleNum < ((RETEQueue)((RETEClauseFilter)newConditionList.get(i))
                                        .continuation).queue.size())){
          newConditionList.add(i, condition);
          inserted = true;
          break;
       }
    }

    if(inserted == false)
       newConditionList.add(condition);

}
```

Figure 5-11: Implementation for the most specific condition first heuristic.

Basically it checks the number of matched facts for the first condition in the existing join sequence (*conditionList*) and then sorts the condition in a new join sequence (*newConditionList*) in ascendant order. The new join sequence is constructed until all conditions in the join sequence are inserted into the new join sequence. The connectivity is then checked by calling

```
optimizeConnectivity(ruleId, newConditionList);
```

after the most specific condition first heuristic. It checks the ordered join sequence and rearranges the join sequence when non-connected conditions are detected.

The beta network is then constructed according to the new join sequences constructed (calling *compileBeta()*). After the construction of the entire RETE network, the initial fact matching resumes by calling *crossJoinAll()*. It goes through all join sequences and product joins the facts matched to first two conditions in the sequence (every matched fact from the first condition is joined to every matched fact from the second condition). Finally the *two-phase RETE* algorithm resumes the normal execution and call *runAll()* to iteratively calculate all inferences.

5.2.5 Extending COROR to Support OWL 2 (Implementation Perspective)

Analytical discussion in section 3.5 shows that both composition algorithms in COROR are semantically independent and therefore they are compatible with OWL 2 RL semantics from the design perspective. Rule-construct dependency graphs for OWL 2 RL rules are also given in that section providing a theoretical foundation for the selective loading algorithm to work on OWL 2 RL. However the application of OWL 2 RL entailments in COROR is still

hampered by two obstacles from the implementation perspective.

The first, and a minor, obstacle is the lack of support of OWL 2 ontology manipulation APIs in μJena. Therefore COROR cannot manipulate OWL 2 ontologies using OWL style operations. However as COROR views ontology as RDF graphs, it is able to manipulate OWL 2 ontologies using triple style operations, and the triple-based RETE engine can also reason over OWL 2 ontologies. A second, and the major, obstacle is the absence of a Jena compatible OWL 2 RL rule set. Early attempts to draft an OWL 2 RL rule set was impeded by the intensive and complex use of RDF list operations in OWL 2 RL semantics. A naïve solution would be to construct a built-in for each list operation. However this will require the construction of a large amount of complex built-ins, greatly complicating the rule set, and limiting the potential for node sharing capabilities and join sequence reordering. Other approaches to handle RDF list in OWL 2 RL reasoners include using ARQ, e.g. SPIN[14], or using customized tags for different RDF list operations and using dedicated list expansion rules, e.g. BaseVISor. However considering the small contribution to the result of this work by providing an OWL 2 RL rule set for COROR, this initiative was suspended.

5.3 *TARS: Tool for Automatic Reasoner Selection*

A desktop prototype implementation of RESP, called TARS (Tool for Automatic Reasoner Selection), is constructed to allow users with little background on ontology reasoning to select appropriate reasoner according to the application characteristics of their semantic applications. The implementation was built in Java using Netbeans 6.5, involving the construction of 4 Java packages, 22 Java classes and in total around 10600 lines of code (6500 of which were automatically generated by Netbeans GUI designer). A full list of the constructed packages and classes can be found in Figure 5-12. Netbeans is free and has an inbuilt drag-and-drop GUI designer, which facilitates fast prototyping of TARS and reduces the effort required on tweaking GUI components at java code level, enabling the author to concentrate on the implementation of application characteristics, reasoner characteristics and connections.

[14] http://www.topquadrant.com/products/SPIN.html

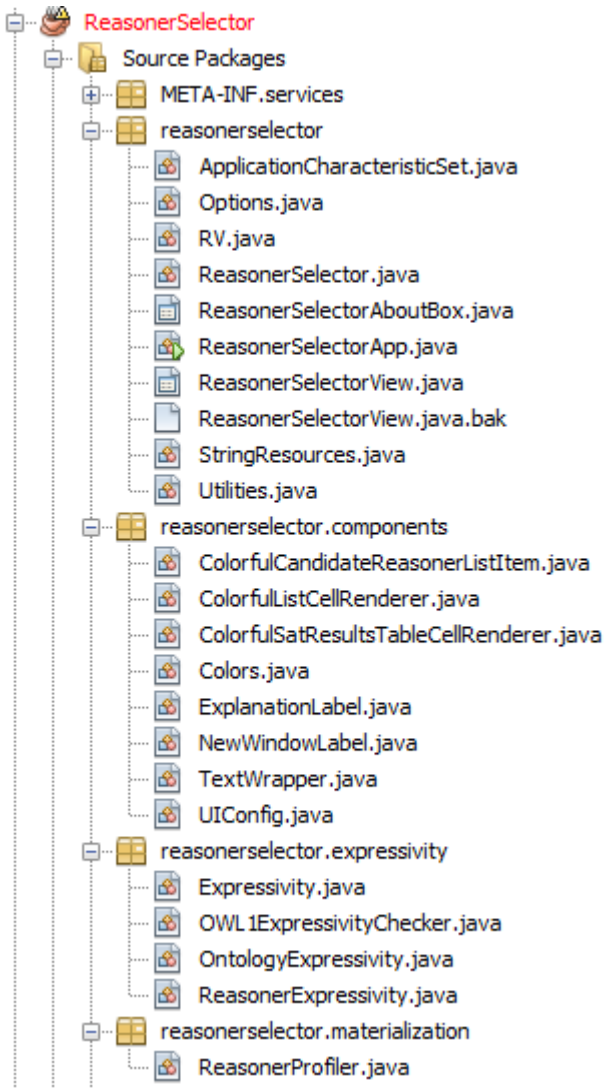


Figure 5-12: Packages and classes of TARS.

All reasoner characteristics distilled from the survey of reasoners, example candidate application characteristics and connections derived from the survey of semantic applications (can be found in section 4.3) are implemented in TARS. Five candidate reasoners are registered, namely FaCT++, KAON2, Pellet, Jena and COROR. They have diverse reasoner characteristics and therefore are suitable for a wide range of semantic applications. For example KAON2 supports efficient conjunctive query answering over a database which is essential for some context-aware applications, FaCT++ and Pellet support complete OWL-DL classification, while COROR runs efficiently on mobile platforms. Reasoner characteristics of candidate reasoners are stored as profiles locally as XML files. All of them are loaded into memory and parsed only once each execution (by the *ReasonerProfiler* class) when the selection process starts, and they remain in memory for the entire lifespan of the

execution. Figure 5-13 gives a snippet of the profile for FaCT++, showing how the RC reasoner expressivity is stored (The full profiles for all the five candidate reasoners can be found in the attached DVD).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<reasoner name="factpp">
       <algorithm>dl tableaux</algorithm>
       ...
       <!-- Entries for other RCs -->
       ...
       <expressivity>
              <dlexpressivity value="shion(d)">
                     <basedl value="al"/>
                     <negation value="yes"/>
                     <union value="yes"/>
                     <inverse value="yes"/>
                     <rolehierarchy value="yes"/>
                     <transitivity value="yes"/>
                     <nominal value="yes"/>
                     <cardinality value="yes"/>
                     <functionality value="yes"/>
              </dlexpressivity>
       </expressivity>
       ...
       <!-- Entries for other RCs -->
       ...
<reasoner>
```

Figure 5-13: A snippet of the XML-coded profile for FaCT++.

Note that the use of automatic composition algorithms in COROR gives rise to a new reasoner characteristic, namely *reasoner composition level*, which indicates if the reasoner has no, static (OWL/DL level), or automatic reasoner composition algorithms, as discussed in section 2.3.3. As will be shown in the evaluation chapter, the use of automatic reasoner composition approach can reduce the resource of OWL reasoning (for rule-entailment reasoners), hence enabling a new candidate application characteristic: *resource sensitive*. This application characteristic may hold for embedded systems that want to deploy OWL reasoning on resource-constrained devices [Kleemann and Sinner 2006, Brennan et al 2009, Koziuk et al, 2008]. According to the above discussion, it is simply deemed that this application characteristic is satisfied if automatic composition algorithms are implemented on the selected reasoner. A description of the application characteristic *resource sensitive* and its connections is given in **Table 5-5** and a description of the new reasoner characteristic *composition level* can be found in Table 5-6.

Table 5-5: The new application characteristic *resource sensitive* and its connections

Application Characteristic	**Connections**

Resource sensitive	*auto* ∈ CPSLvl

Table 5-6: The new reasoner characteristic *composition level* and its possible values

Reasoner Characteristic	**Values**		
Composition level (CPSLvl)	Rule (*rule*)	OWL/DL (*owl*)	Automatic (*auto*)

Three graphical interfaces are implemented, namely the *application characteristics selection* interface, the *reasoner registration* interface and the *reasoner selection results* interface. The *application characteristics selection* interface can be found in Figure 5-14, on which all candidate application characteristics are listed. In general, a user first analyses their application for relevant application characteristics, and then ticks the corresponding candidate application characteristics on the *application characteristics selection* interface. The user can start the RESP reasoner selection process by hitting the "*Perform Selection*" button.

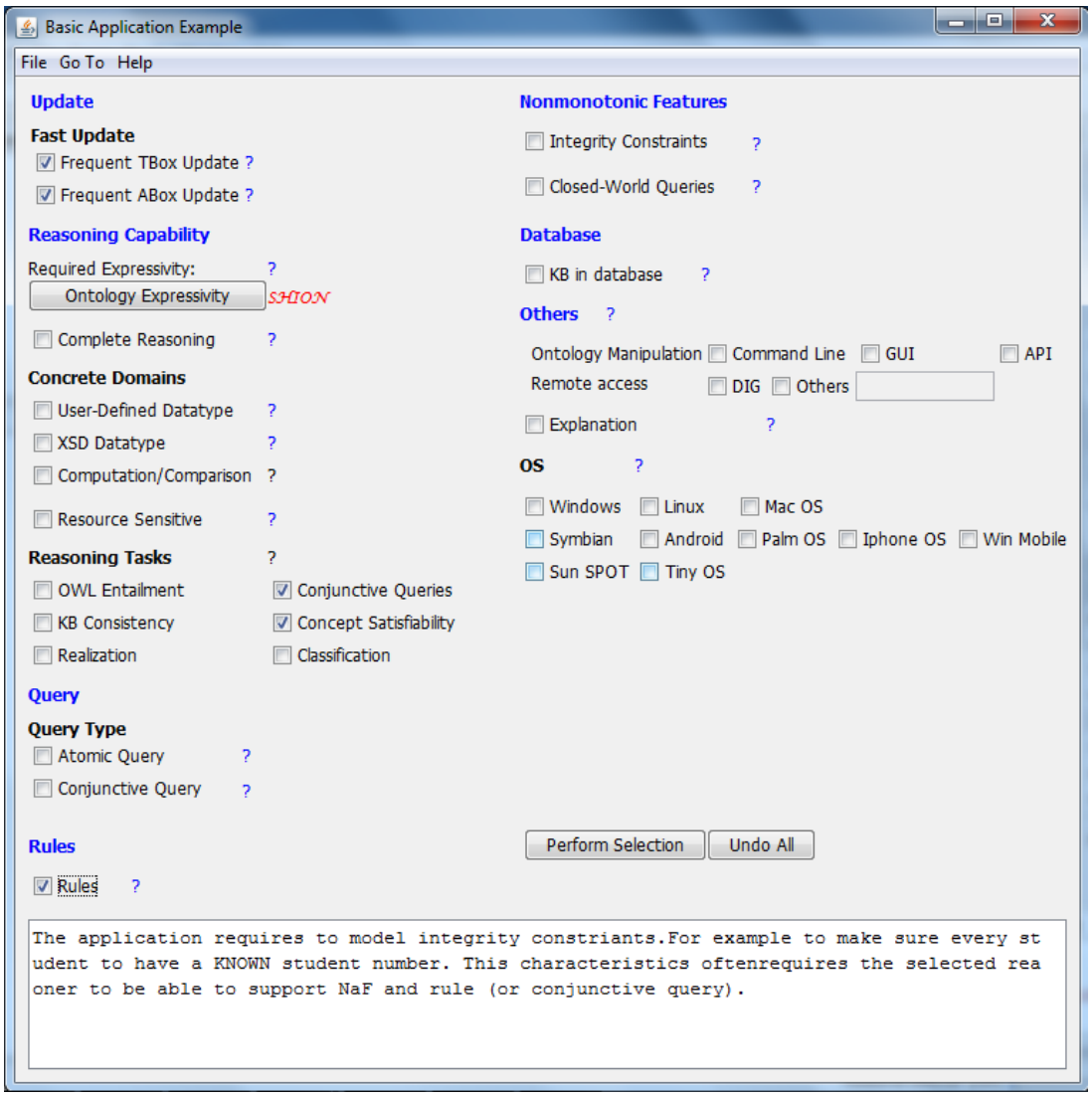


Figure 5-14: Application characteristics selection interface.

It worth mentioning that ontology expressivity is gathered automatically in TARS. The *OWL1ExpressivityChecker* class is responsible for ontology expressivity checking. In general it checks for OWL classes, axioms and properties using OWLAPI methods, and detected expressivities are indicated by a set of class fields such as *hasNegation*, *hasTransitive*, and so on. In order to check the expressivities for OWL classes, the *OWL1ExpressivityChecker* lists all named classes then checks all its super-classes, sub-classes, equivalent-classes and enumerations, setting the corresponding expressivity fields. Similarly In order to check the expressivities for class axioms, the *OWL1ExpressivityChecker* checks if they are *OWLSubClassAxiom*, *OWLEquivalentClassesAxiom*, or *OWLDisjointClassesAxiom* and then looks into the axioms to check the expressivities included. If a property axiom is found then the expressivity field

hasRoleHierarchy is set to be true. In order to check the expressivities for properties, the *OWL1ExpressivityChecker* checks if the property is transitive, symmetric, inverse, functional or inverse functional, and then sets the corresponding expressivity fields. Then the domain and range of the property is also checked. The combination of the expressivity fields is then the DL expressivity of the ontology. For example, if *hasNegation* and *hasUnion* are set to be true then the ontology has expressivity $\mathcal{ALC}$; if *hasTransitive* is set to be true then the expressivity $\mathcal{R}+$ is appended and the expressivity letter is set to be $\mathcal{S}$.

A user can view the hints for an application characteristic by clicking the question mark besides each application characteristic. The explanation is then displayed in the text field located in the bottom.

Selected application characteristics are stored in an *ApplicationCharacteristicSet* class instance. To reduce the complexity of the prototype implementation, each connection is hardwired as a Java method of the *ReasonerSelector* class. Connection methods follow a naming scheme, namely "evaluateXXX" with XXX representing the corresponding application characteristic. Connection methods take the *ApplicationCharacteristicSet* instance (which lists the required application characteristics) and a *ReasonerProfiler* instance (from which the reasoner characteristics for candidate reasoners can be accessed) as arguments, and returns an *ArrayList* of *SatisfactionLevel* instances with each entry in the list representing the satisfaction of a candidate reasoner to this application characteristic. Then the matchmaking is performed in the *ReasonerSelector* class by evaluating all connection methods against the *ApplicationCharacteristicSet* instance. Figure 5-15 gives the code snippet of connections for the application characteristic *integrity constraints*. If the application characteristic *integrity constraints* is not selected then it ceases the connections checking immediately. Otherwise it iterates all candidate reasoner profiles and checks for each candidate reasoner the value of the reasoner characteristic *native CWA support*.

```
ArrayList<SatisfactionLevel> retVal = new ArrayList<SatisfactionLevel>();

if(acs.getAppCharacteristicValue(RV.AC.integrityConstraints).equals(RV.NO))
   return retVal;

ArrayList<ReasonerProfiler.ReasonerProfile> profiles =
                                          profiler.getReasonerProfiles();

for(int i=0; i<profiles.size(); i++){
   ReasonerProfiler.ReasonerProfile profile = profiles.get(i);
   if(profile.nafInQuery.equals(RV.YES) || profile.nafInRule.equals(RV.YES))
      retVal.add(new SatisfactionLevel(profile,
                           SatisfactionLevel.satisfy));
   else
      retVal.add(new SatisfactionLevel(profile,
```

```
                                SatisfactionLevel.not_satisfy));
    }
return retVal;
```

Figure 5-15: Connections for AC integrity constraints.

Results are displayed in the *reasoner selection results* interface after the selection finished (as indicated in Figure 5-16). Traffic light notations are used to indicate the satisfaction percentage for each candidate reasoner. By clicking a candidate reasoner, a user can view the detail of the satisfaction for each application characteristic on the right hand side. Selected application characteristics are coloured in green or red, indicating their satisfaction or not. Unselected application characteristics are in grey. By selecting an application characteristic users can view the reason why it is satisfiable/not satisfiable.

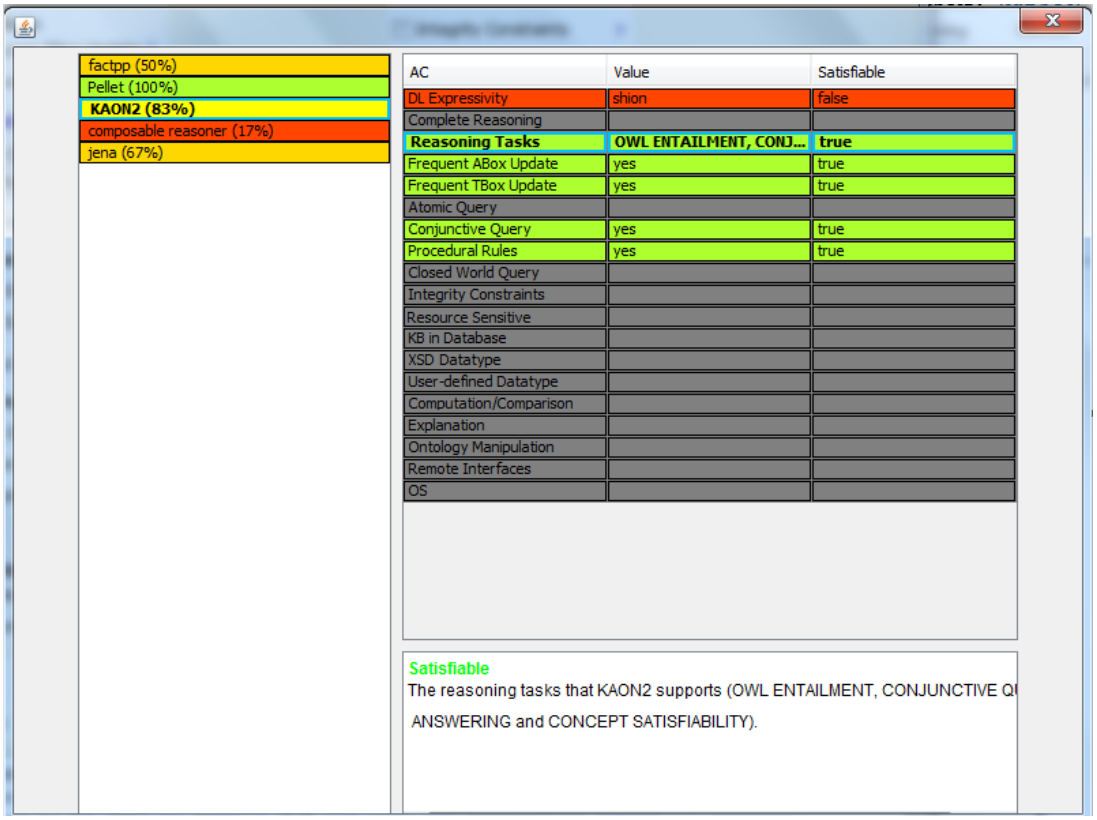


Figure 5-16: Reasoner selection results interface

Figure 5-17 shows the *reasoner registration* interface where new candidate reasoners can be registered. Reasoner characteristics are listed allowing users to input the corresponding reasoner characteristics for the reasoner.

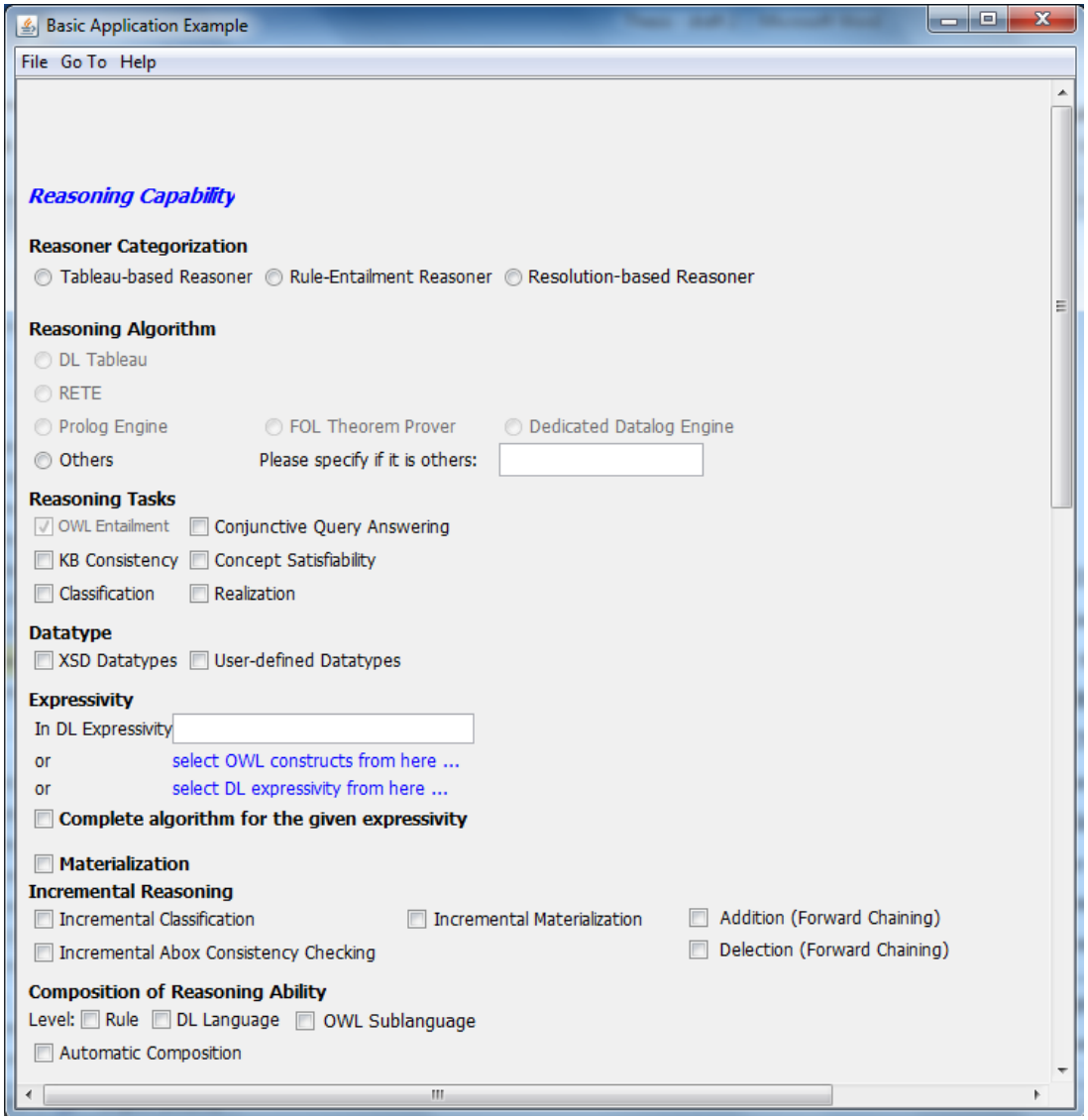


Figure 5-17: The user interface for registering candidate reasoners.

Reasoner expressivities are specified manually for the candidate reasoners to be registered. The expressivities can be specified either using DL or using OWL constructs, as separately illustrated in Figure 5-18 and Figure 5-19. Specifying reasoner expressivity using OWL constructs is mainly designed for non-DL reasoners such as rule-entailment reasoners or resolution-based reasoners using ontology independent translation. Four common expressivities, including RDFS, OWL-Lite, OWL-DL, and R-entailment, are given as "hotkeys" for them.

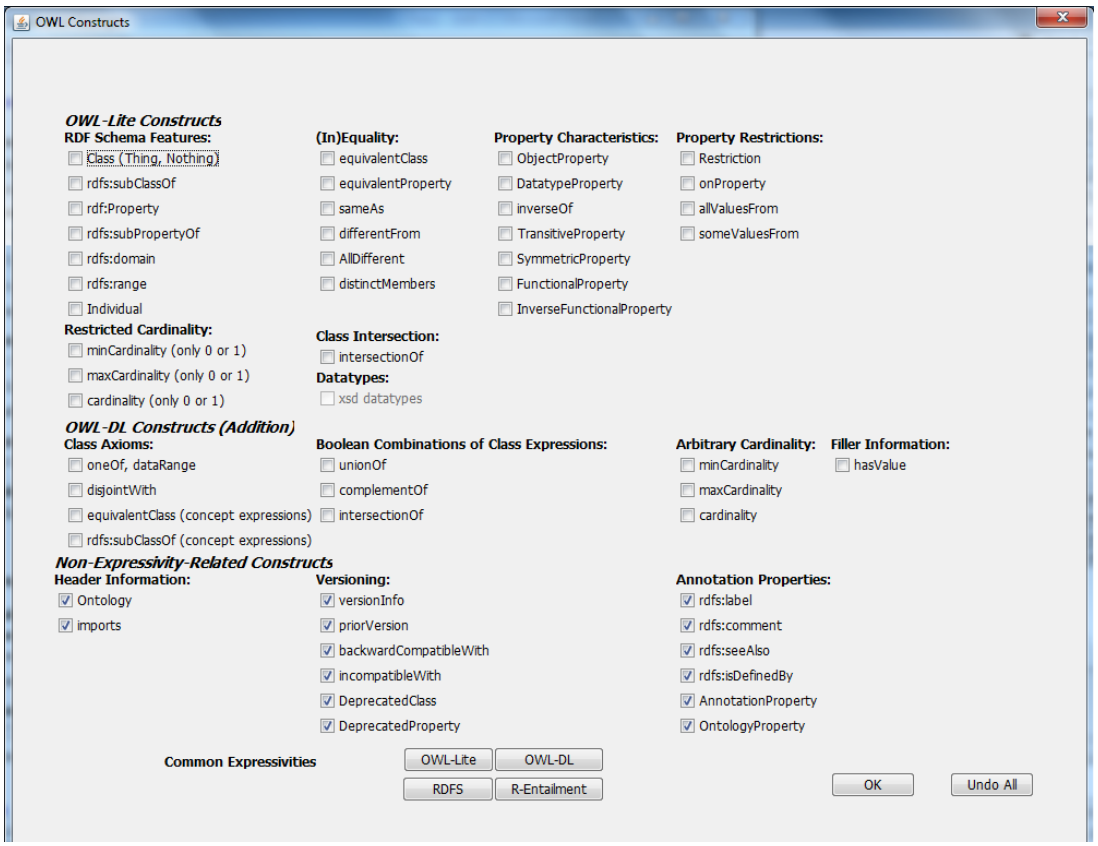


Figure 5-18: User interface for specifying reasoner expressivity using OWL constructs.

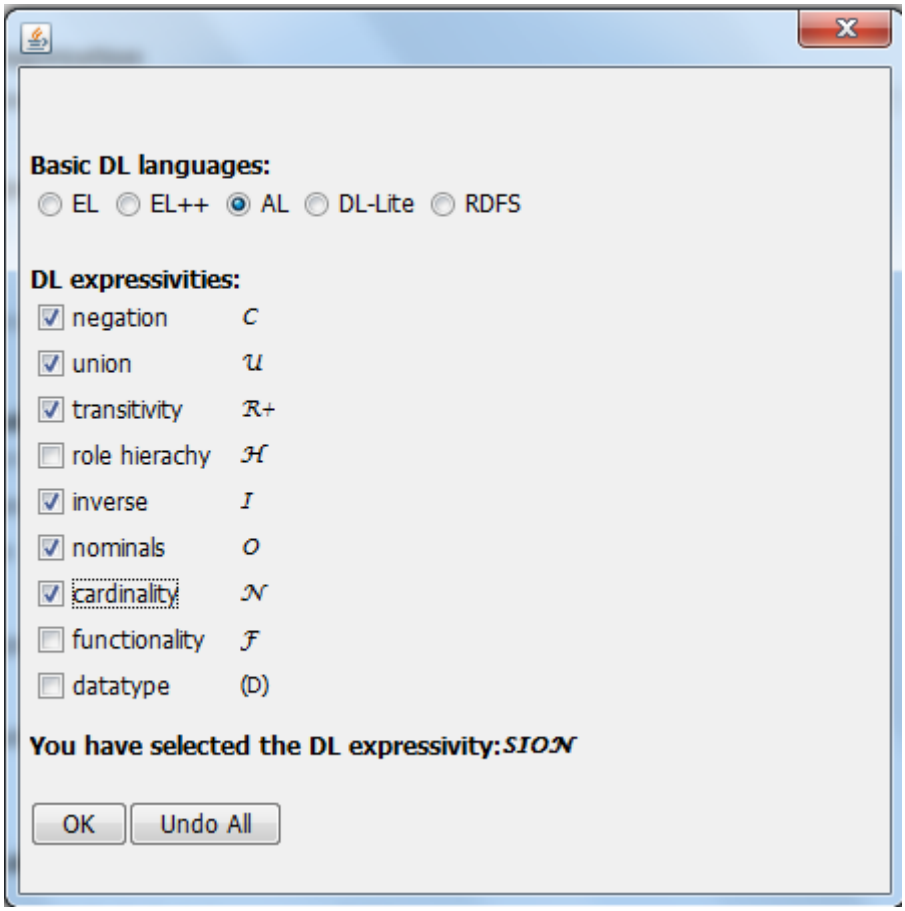


Figure 5-19: User interface for specify reasoner expressivity in DL.

5.4 *Summary*

This chapter presents the prototype implementations of COROR and RESP. The implementation of COROR is discussed with respect to five aspects, including the selection of a proper platform based on which COROR was implemented, the construction of a resource-constrained rule-entailment reasoner where the novel reasoner composition algorithms can be implemented, the implementation of the pD* entailment rules as Jena rule format, the implementation detail for the two novel composition algorithms, and finally the extension of COROR to support OWL 2 from the implementation perspective. Sun SPOT is selected as an appropriate platform on which the implementation is performed since its resource-constrained environment (J2ME CLDC 1.1) and well-integrated development tools. Several requirements are identified for finding a suitable resource-constrained rule-entailment reasoner to implement the composition algorithms. However a survey suggests that no off-the-shelf such reasoners exist. Therefore a resource-constrained rule-entailment reasoner is constructed by porting Jena RETE engine into μJena. The implementation of pD* semantics is simple translation from pD* entailment rules into Jena conformant rules.

Some modifications are performed to combine some rules. However it is shown that these modifications do not change the reasoning capability. Detail for the implementation of composition algorithms is then presented. The description of the implementation of the *selective rule loading* algorithm mainly concentrates on the implementation of the rule-construct graph using text-based mappings. Code snippets are also given to illustrate how a selective rule set is constructed according to the rule-construct mappings. Class diagrams are provided to illustrate the Jena RETE engine and how it is extended to integrate the *two-phase RETE* algorithms. Problems encountered during the implementation of the *two-phase RETE* algorithms are also discussed with solutions presented, including using the dual vector mechanism to solve sharing of intermediate results and using swappable conditions to support different sharable conditions share one common condition node. The process of the *two-phase RETE* algorithm is then illustrated in detail with code snippet presented.

The heavy use of RDF lists in OWL 2 RL semantics hampers the construction of a set of Jena compatible OWL 2 RL rules, which resulted in the lack of support of OWL 2 RL in COROR. However this was simply an implementation detail and extension would not contribute much to validate the contributions of this research with respect to the semantic independent feature of the composition algorithms. Therefore no actual implementation is performed to extend COROR to support OWL 2 RL but it remains one of the most important pieces of work needed to support the adoption of COROR in a user community.

A prototypical desktop implementation of RESP, TARS, is also presented in this chapter. Users can use it to perform automatic reasoner selection and to register new candidate reasoners. The candidate application characteristics, reasoner characteristics and connections implemented in TARS are based on the example candidate application characteristics, reasoner characteristics and connections derived in from the survey of reasoners (section 2.3.1.2) and the survey of semantic applications (section 4.3). However the use of COROR enables a new reasoner characteristic, i.e. *composition level*, and a new application characteristic, i.e. *resource sensitive*. This implementation is to some extent limited in terms of practical usage due to the use of example ACs and the hard-coding of AC/RC connections and the matching algorithm. However it is still sufficient for demonstration and evaluation purposes.

Section 3, 4 and 5 together complete the research objective 2 and objective 4. In the next chapter evaluations are carried out to evaluate the performance of COROR and usability of TARS), targeting the research objective 3 and 5.

Chapter 6
Evaluation

6.1 *Introduction*

In the previous chapters, solutions to the research question as to

"How an appropriate resource-constrained OWL reasoner can be automatically composed and be selected based on application characteristics."

are proposed, designed and implemented. In Chapter 3 the design of two novel automatic reasoner composition algorithms, i.e. the *selective rule loading* algorithm and the *two-phase RETE* algorithm, are presented in order to enable OWL reasoning to run better on resource-constrained environments. The two composition algorithms compose the reasoner both at the rule set level and inside the RETE algorithm. Considering that these two composition levels may complement each other, a *hybrid* algorithm is designed to combine both composition algorithms. Chapter 4 presents the design of an automatic reasoner selection process, RESP. It gathers application characteristics from applications and reasoner characteristics from candidate reasoners. Then the reasoner selection is performed through matchmaking between application characteristics and reasoner characteristics. Implementations of COROR and RESP (named as TARS) are given in Chapter 5.

Then in this chapter, these solutions are evaluated in order to investigate *to what extent* the application of these solutions can address the research question. As discussed in the motivation section in the introduction chapter, the very original motivation of having reasoner composition approach is to allow fewer resources to be used when applying OWL reasoning in resource-constrained environments. Hence a direct way to evaluate if the two designed composition algorithms can satisfy this aspect would be to study the change of OWL reasoning performance of a reasoner before and after the application of these two

composition algorithms, as informed in the objective 3.

The objective 3 can be further divided into two evaluation objectives. A first evaluation objective is the comparison of the reasoning performance of COROR when it is configured to use different composition algorithms, compared with no composition algorithm. This gives a direct impression on the performance change brought by the different composition algorithms. A second evaluation objective is to compare COROR (when composition algorithms are applied) with the state of the art (maybe resource-constrained) rule-entailment reasoners. This can show how a composable rule-entailment reasoner performs compared to the other rule-entailment reasoner implementations, and facilitates the identification of performance merits and pitfalls of COROR compared to other state of the art rule-entailment reasoners.

According to these two evaluation objectives, two experiments are designed and executed. An *intra-reasoner comparison* is designed that measures and compares the time/memory required by COROR to reason over the same set of ontologies when it is configured in different composition modes, i.e. COROR-noncomposable, COROR-selective, COROR-two-phase and COROR-hybrid. A set of 19 small or medium sized (no larger than 13000 triples) ontologies is selected for this experiment. The selection of time and memory required by reasoning as evaluation metrics can directly show the changes of performances across the systems. Other evaluation metrics do exist that are less appropriate for COROR.

An *inter-reasoner comparison* is designed to compare COROR-hybrid with some other state of the art (resource-constrained) rule-entailment reasoners, including Jena, Bossam, BaseVISor and OWLIM. COROR-hybrid is selected as it combines both composition algorithms and therefore can better represent a composable reasoner. Pellet is also included in this experiment, even though it is not necessarily directly comparable, as both process different sets of semantics. The inclusion of Pellet is intended only to give readers an intuition of the performance of COROR comparing to a full-fledged DL-tableaux reasoner. Results of the two experiments are presented and discussed in detail. More detail on the two experiments can be found in section 6.2. The correctness of the composable reasoning and the OWL semantics coverage of COROR are also discussed.

The major motivation for an automatic reasoner selection process is the large amount of difficulties to be involved in the future reasoner selection process: it may require a large amount of effort from both reasoner experts and application developers or even may lead to inappropriate reasoners to be selected. To solve this, RESP is proposed, designed and

implemented (as a prototype implementation TARS). In order to study *to what extent* the introduction of RESP can reduce the efforts involved in the reasoner selection process, the usability of TARS is then a good aspect to evaluate.

A usability experiment is then designed to evaluate the usability of TARS since it is an implementation of RESP. Two participant groups are used: an *application-aware group* consisting of 17 participants with strong background on developing semantic applications, and a *reasoning-aware group* consisting of 5 participants with strong background on ontology reasoning. A reasoner selection task is designed to require application-aware participants to use TARS to select an appropriate reasoner for the given application scenario, and a reasoner registration task is designed to require reasoning-aware participants to use TARS to register a candidate reasoner. Questionnaires are given to both groups to collect feedbacks.

Detail of the designs, execution results and discussion of the intra- and inter- reasoner comparison is presented in section 6.2. The usability test and results can be found in Section 6.3. Summary and key findings are presented in section 6.4.

6.2 *Performance Comparison and Investigation of COROR*

This section describes in detail the design, execution and results of the intra- and inter-reasoner comparisons that are designed to fulfil the research objective 3. The rationale and criteria for selecting the evaluation metrics are discussed first in section 6.2.1. Then in section 6.2.2 settings of the experiments and their executions are presented. Results and discussions for both experiments are separately given in 6.2.3 and 6.2.4. Section 6.2.5 discusses the correctness and the OWL semantics coverage of COROR.

6.2.1 Criteria of Selecting Performance Metrics

The *memory usage* and *reasoning time* required by COROR to fully compute the entailments of a given ontology are selected as the evaluation metrics to evaluate the performance of COROR. The underlying motivation for this selection is that the comparison and analysis of them is a direct and effective way to show the impact on reasoning performance. As a matter of fact, as indicated in the motivation in Chapter 1, a primary goal of COROR (and reasoning composition algorithms) is to reduce the resource usage such that OWL reasoning can be better applied to resource-constrained devices (e.g. sensor mote), which further encourages the use of memory usage and reasoning time as the evaluation metrics. It is not the first time that these metrics are used to in a performance evaluation. Many previous researchers have already employed (one or both of) them in reasoner related

experiments, such as Oracle database 11g [Wu et al 2008], SwiftOWLIM [SwiftOWLIM ver 2.9.1 SysDoc], MiRE4OWL [Kim et al 2010], μOR [Ali and Kiefer 2009] and so on.

Other evaluation metrics also exist for measuring the performance of OWL reasoners from other aspects, e.g. *execution time for DL reasoning tasks*, *conjunctive query answering time*, and *benchmark suites*. However, these metrics are excluded, as they are not suitable for the purpose of this evaluation, for the reasons explained in the following paragraphs.

Execution time for DL reasoning tasks. Besides entailment many DL reasoners also implement a different set of reasoning tasks such as subsumption, instantiation, realization, consistency checking and so on [Baader et al 2007]. Note that DL reasoners here refer to those reasoners using DL reasoning techniques to reason over OWL, including all DL-tableaux reasoners, some resolution-based reasoners such as KAON2, and some other reasoners such as CEL or QuOnto. Measuring the execution time for these DL reasoning tasks is widely adopted in the evaluations of DL reasoners, such as Pellet [Sirin et al 2005], CEL [Baader et al 2006], Minerva [Zhou et al 2006] and KAON2 [Motik 2008].

However as entailment is the most basic (and only explicitly implemented) reasoning task in COROR and the other reasoning tasks are not explicitly implemented but can be reduced to querying the entailment closure as discussed in section 5.2.2, therefore the performance of the reasoning tasks are highly dependent on the performance of entailment calculation. Hence measuring the performance of other DL reasoning tasks is omitted from the performance experiments of COROR.

Query Answering Time. Measuring the query answering time is also an often used evaluation metric for OWL reasoners, especially for those aiming at scalable ABox query answering. A number of reasoners, such as Pellet [Sirin et al 2005], RacerPro [RacerPro Release Notes v1.9.2], Miverva [Zhou et al 2006] and KAON2 [Motik and Sattler 2006], have used it in experiments to evaluate the scalability in terms of answering queries over a large ABox. This metric is also excluded from the performance experiment as the two designed composition algorithms do not extend μJena in this regard and no contribution is claimed in terms of query answering.

Benchmark Suites. Some benchmark suites, such as the Lehigh University Benchmark (LUBM) suite [Guo et al 2005] and the University Ontology Benchmark (UOBM) [Ma et al 2006], are designed to evaluate the scalability of a reasoner when handling a large ABox. They usually generate very large artificial ABox from a given TBox, which is often too large

to be used to evaluate the composition algorithms designed and implemented in this thesis where the context of resource-constrained devices is imposed. For example, LUBM generates synthetic data based on a university domain ontology; the smallest dataset of LUMB with only one university data, i.e. LUBM(1,0), still reaches 8.7MByte with 103K triples. Similarly the OpenRuleBench suite [Liang et al 2009] is a performance benchmarking tool designed to test rule engines from four aspects, including large join test, Datalog recursion, default negation, dynamic indexing tests and database interface tests. For each aspect, a set of tests is provided. The large join test appears to be relevant to this thesis where rule-entailment reasoner is used. However as the OpenRuleBench suite aims to evaluate how desktop rule engines perform at a web scale, the dataset is still too large for resource-constrained environments, e.g. the smallest dataset contains 50K facts.

There are other metrics that are used to evaluate particular reasoners but are not commonly used in general, such as the *ontology upload speed* [SwiftOWLIM ver 2.9.1 SysDoc, Kiryakov et al 2005] which measures the amount of triples a reasoner can process per second, *ontology deletion speed* [Kiryakov et al 2005] which represents the time a reasoner required to handle a deletion transaction, *completeness of query answer set* [Zhou et al 2006, Jang and Sohn 2004] which is the completeness (given in percentage) of a query answer set comparing with the goal standard, and so on. They are also excluded from this evaluation as they are too specific for a particular reasoner to fit into the objectives identified for this research.

6.2.2 Design and Execution

Different settings were used for the intra- and inter-reasoner comparisons. The intra-reasoner comparison was performed on a Sun SPOT emulator v4.0 blue[15] with the support of the Squawk JVM [Squawk JVM] which is CLDC 1.1 conformant. In this comparison COROR was configured as four different composition modes, i.e. COROR-noncomposable, COROR-selective, COROR-two-phase, and COROR-hybrid, and all of them need to reason over a same set of ontology (as will be listed later in this section). The memory usage and reasoning time for each mode to reason over each ontology is measured and compare against each other. A Sun SPOT sensor board has a 180MHz 32-bit ARM920T core processor with 512K RAM and 4M Flash. The Sun SPOT emulator was running on a desktop computer with Intel Dual Core CPU @ 2.4GHz, 3.25GB RAM and Windows XP professional version 2002 SP2 x86.

[15] http://www.sunspotworld.com/docs/index.html

State of the art rule-entailment reasoners are selected for comparison in the inter-reasoner comparison, including Bossam 0.9b45, Jena v2.6.3, BaseVISor v1.2.1, and swiftOWLIM v3.0.10. Their selection is motivated by the fact that they are state of the art rule-entailment reasoners using RETE algorithm and they have similar OWL expressivity as COROR. As discussed earlier, comparing them side by side with COROR can reflect the performance merits and pitfalls of COROR comparing to state of the art rule-entailment reasoners. MiRE4OWL and μOR were not accessible for usage in our experiments, so they are not included in the inter-reasoner comparison. Bossam is used in this evaluation as a resource-constrained rule-entailment reasoner. Even though Bossam supports J2ME CDC, due to its wide use of Java class *java.util.List* which is not included in CLDC 1.1, it cannot run on Sun SPOT. However, it proved time prohibitive to port the other reasoners onto the SunSPOT platform. As a result, inter-reasoner comparison was performed using the same desktop computer as described above, using a J2SE platform in Eclipse Helios with Java SE 6 Update 14 and maximum heap size as 128MB. Note, all J2ME CLDC 1.1 java code is backward compatible with J2SE (after 1.3) and so COROR runs on the desktop without modification. Jena was configured to use the RETE engine only (use *GenericRuleReasoner* and FORWARD_RETE mode) and it used the same pD* rule set as described in section 5.2.3. Pellet 1.5.1 was also included in this comparison in order to provide readers an intuition of the performance of COROR comparing to a full-fledged DL tableau reasoner.

Memory usage and reasoning time are measured using Java built-in time and memory methods. For measuring the reasoning time, the java method

```
System.currentTimeMillis()
```

was invoked both before and after the ontology reasoning is performed, that is

```
InfGraph.prepare().
```

The reasoning time is then the difference of the two measurements.

The memory usage that the composable reasoner requires to reason over an ontology is measured by subtracting the free memory from the total memory

```
Runtime.getRuntime().totalMemory() -
  Runtime.getRuntime().freeMemory()
```

after the

```
InfGraph.prepare().
```

is invoked. At this stage, all reasoning is finished and the RETE network reaches its maximum, populated with all asserted and inferred triples.

The reasoning time and memory usage are measured separately in different executions of COROR so no interference between them occurs. Each result of time/memory used in the evaluation is the average of 10 individual measurements to reduce the error in each measurement. Furthermore the method

```
System.gc()
```

is explicitly invoked 20 times before the memory measurements, releasing as much garbage memory as possible so interference from non-recycled garbage memory is reduced. A threshold of 30 minutes is set to avoid excessively long reasoning times, and manual termination is imposed for reasoning processes longer than this threshold.

In total 11 ontologies of small sizes and moderate expressivities are selected for the intra-reasoner comparison (as given in Table 6-1a), and eight more ontologies are used for the inter-reasoner comparison (as given in Table 6-1b). All ontologies used in this experiment can be found in the attached DVD of this thesis. They are selected for three reasons: (1) they model different domains, which, to some extent, is able to represent the diversity of the content of ontology that could be used in embedded devices, (2) they vary in expressivity so their usage avoids any unintentional bias where some OWL constructs are over- or under-used by some ontology designers in different application domains, and (3) they are well known and commonly used, and so are relatively free from errors.

Table 6-1: Ontologies used in intra-/inter-reasoner comparison experiments

(a) Eleven ontologies used in the intra-reasoner comparison.

Ontology	Expressivity	No. of cls/prop/indv	Size (triples)
teams	$\mathcal{ALCIN}$	9/3/3	87
owls-profile	$\mathcal{ALCHIOF(D)}$	54/68/13	116
Koala	$\mathcal{ALCON(D)}$	20/7/6	147
university	$\mathcal{SIOF(D)}$	30/12/4	169
Beer	$\mathcal{ALHI(D)}$	51/15/9	173
mindswapper	$\mathcal{ALCHIF(D)}$	49/73/126	437
Foaf	$\mathcal{ALCHIF(D)}$	17/69/0	503
mad_cows	$\mathcal{ALCHOIN(D)}$	54/17/13	521

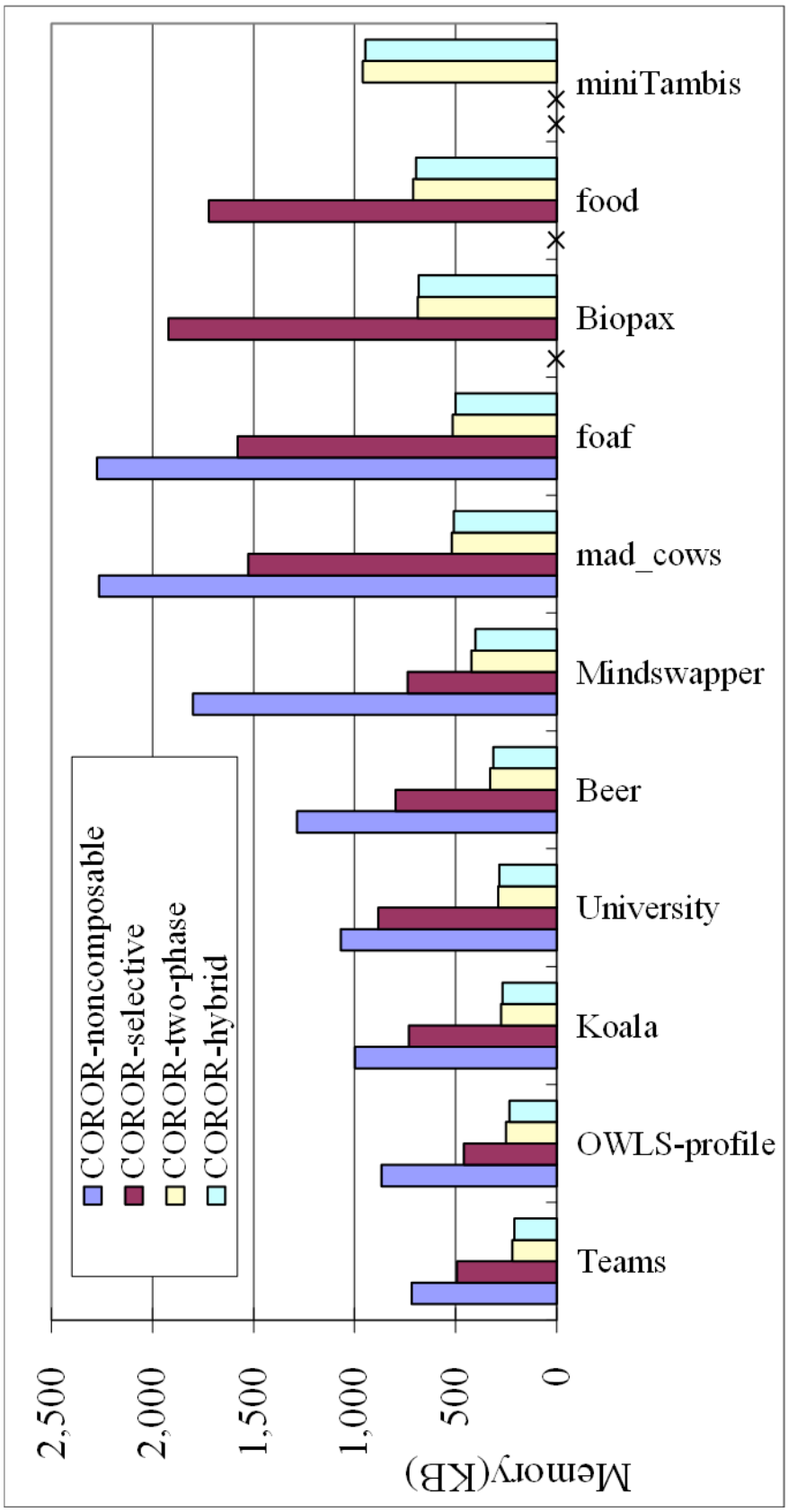
Biopax	$\mathcal{ALCHF}(\mathcal{D})$	28/50/0	633
Food	$\mathcal{ALCOF}$	65/10/57	924
mini-tambis	$\mathcal{ALCN}$	183/44/0	1080

(b) Eight more ontologies used in the inter-reasoner comparison.

Ontology	Expressivity	No. of cls/prop/indv	Size (triples)
amino-acid	$\mathcal{SHOF}(\mathcal{D})$	55/24/1	1465
atk-portal	$\mathcal{ALCHIOF}(\mathcal{D})$	169/147/75	1499
Wine	$\mathcal{SHOIN}(\mathcal{D})$	77/16/161	1833
Pizza	$\mathcal{ALCF}(\mathcal{D})$	87/30/0	1867
tambis-full	$\mathcal{SHIN}$	395/100/0	3884
Nato	$\mathcal{ALCF}(\mathcal{D})$	194/885/0	5924
Mged	$\mathcal{RDFS}(\mathcal{DL})$	437/21/1278	6284
Tap	$\mathcal{RDFS}(\mathcal{DL})$	5488/0/0	12085

6.2.3 Intra-Reasoner Comparison: Results and Discussions

The memory usage (byte) and reasoning time (millisecond) required by the four composition modes, i.e. COROR-noncomposable, COROR-selective, COROR-two-phase and COROR-hybrid, to reason over the selected ontology are listed in Table 6-2. For easier presentation the raw data is represented as bar charts separately in Figure 6-1a (memory) and Figure 6-1b (reasoning time). In the charts the ontologies are arranged ascendant by sizes. There are no results for some measurements, e.g. the time and memory usage for COROR-noncomposable to reason over biopax, food and miniTambis ontologies. This is because these tests exceeded the threshold and therefore manual termination is applied to the reasoning process. For these measurements "X" symbols are placed on the corresponding position on the charts to represent their lack of results.


miniTambis
food
Biopax
foaf
mad_cows
Mindswapper
Beer
University
Koala
OWLS-profile
Teams
COROR-noncomposable
COROR-selective
COROR-two-phase
COROR-hybrid
2,500
2,000
1,500
1,000
500
0
Memory(KB)

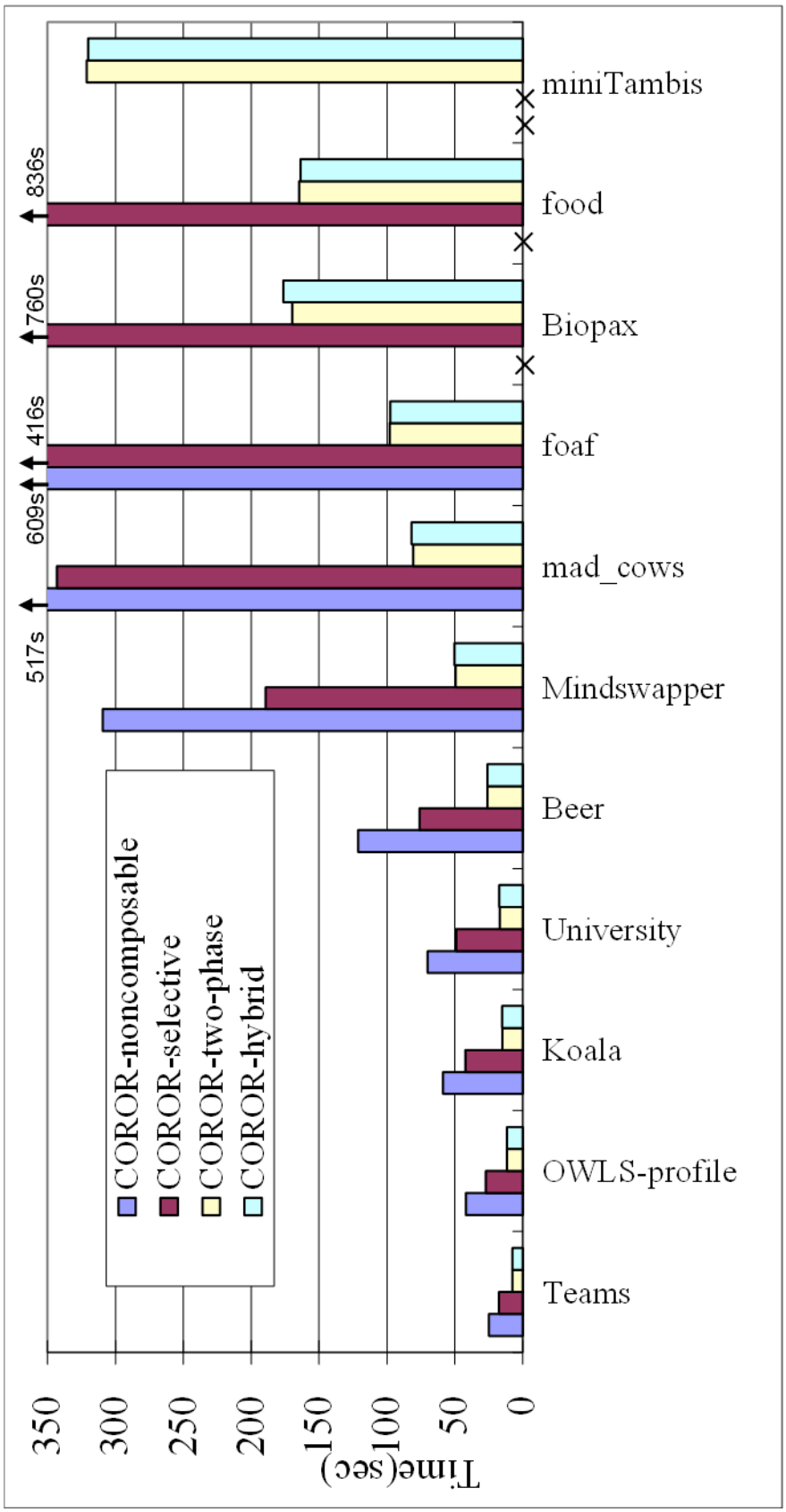

miniTambis
food
Biopax
foaf
mad_cows
Mindswapper
Beer
University
Koala
OWLS-profile
Teams
836s
760s
416s
609s
517s
CORORnoncomposable
CORORselective
CORORtwophase
CORORhybrid
Time(sec)
350
300
250
200
150
100
50
0

Table 6-2: Raw data for memory tests and time tests (memory in byte and time in millisecond).

Ontology	COROR-noncomposable		CORORselective		CORORtwo-phase		CORORhybrid	
	Memory (byte)	Time (ms)	Memory (byte)	Time (ms)	Memory (byte)	Time (ms)	Memory (byte)	Time (ms)
Teams	716,996	24,734	493,140	17,406	218,700	7,469	208,400	7,516
owls-profile	866,600	41,938	459,304	27,171	250,584	11,453	232,092	11,563
Koala	997,132	58,688	731,540	42,328	275,704	14,843	268,048	15,031
university	1,068,696	70,093	882,572	48,922	288,948	16,782	283,832	17,282
Beer	1,284,352	121,282	797,316	76,031	328,680	25,797	313,352	25,891
mindswapper	1,800,228	309,469	737,824	189,375	422,056	49,516	401,452	50,266
mad_cows	2,265,992	517,609	1,526,288	343,234	518,240	80,641	508,592	81,828
Foaf	2,276,176	609,218	1,578,896	416,125	513,656	98,125	500,044	97,547
Biopax	N/A	N/A	1,921,824	760,218	688,220	169,812	683,032	176,328
Food	N/A	N/A	1,720,956	836,344	710,924	164,781	695,852	163,672
miniTambis	N/A	N/A	N/A	N/A	959,644	321,297	946,816	320,094

As can be found in both the diagrams and the table, the three composable modes, i.e. COROR-selective, COROR-two-phase and COROR-hybrid, use much less memory and reasoning time to reason over the same ontology than COROR-noncomposable does (where only the original ported Jena RETE engine is used and no composition algorithms are applied). Furthermore for eight ontologies (which are teams, owls-profile, koala, university, beer, mindswappers, mad_cows, and foaf) COROR-hybrid requires less than 512KB memory (the RAM size of Sun SPOT). Six ontologies for COROR-two-phase and two ontologies for COROR-selective use less than 512KB memory, however, no ontology falls into this size for COROR-noncomposable, which indicates for a given memory bound COROR composable modes (in particular for COROR-hybrid) can reason over larger ontology than COROR-noncomposable. In addition using less memory than the physical RAM size means there is not a need for the ontologies to frequently page in and out, further improving the time performance and reducing the power consumption.

It is clear that the COROR-selective, COROR-two-phase and COROR-hybrid use less time and memory to fully compute the entailments for all tested ontologies than COROR-noncomposable. However the amount of saved memory/time varies largely from the three COROR composable modes. For example, COROR-noncomposable uses 1.8MB memory to reason over mindswappers; COROR-selective requires 738KB to reason over the same ontology (of which the memory reduction is around 1.0MB); but COROR-two-phase and COROR-hybrid correspondingly require only 422KB and 401KB (of which the memory reduction is around 1.4MB). Similar situations also apply to the time reduction. As can be found in Table 6-2 and Figure 6-1 COROR-hybrid generally has the most memory/time reduction among the three composable modes, and then comes the COROR-two-phase which usually requires slightly more memory/time usage than COROR-hybrid. COROR-selective usually has the least memory/time reduction among them all.

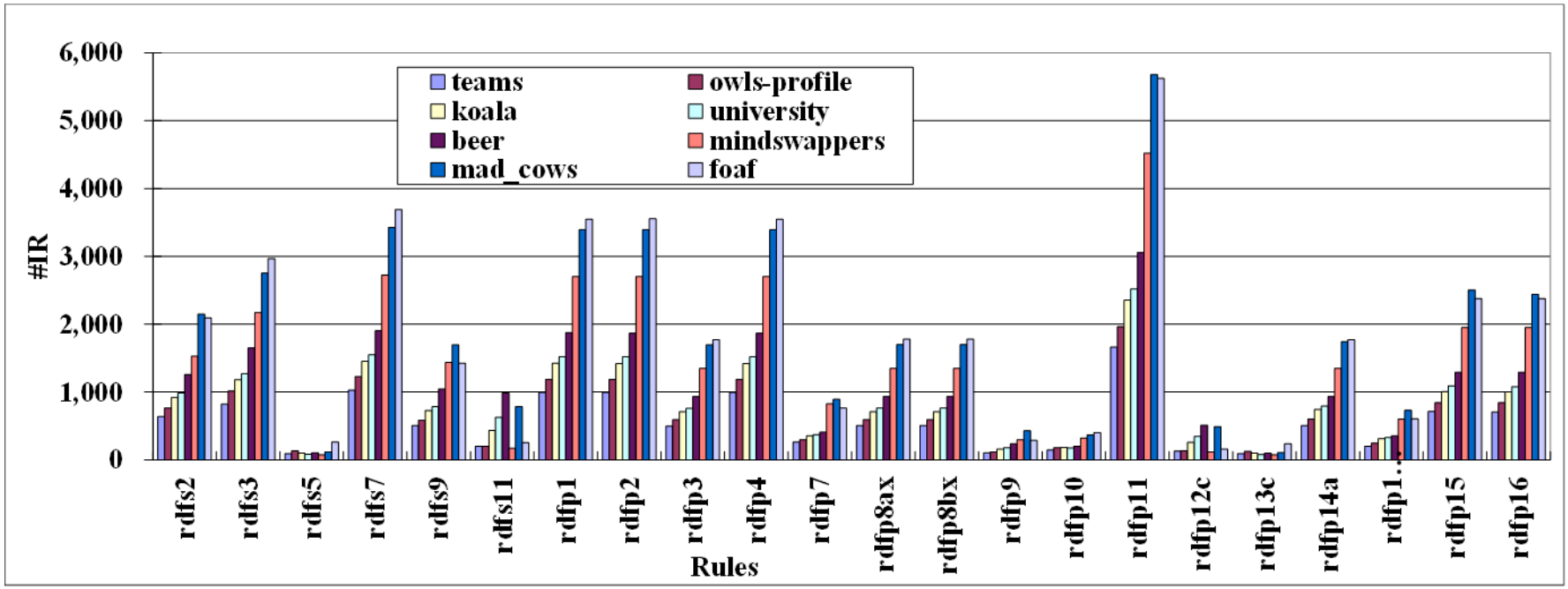
In the remaining parts of this subsection, in-depth analysis is performed to investigate the causes of memory/time reduction for all the three composable modes, based on which the differences in memory/time among composable modes are discussed and analysed. Since it is proven to be difficult to measure the exact memory usage or time consumption for a rule or only for a part of the RETE network using the approaches described in section 6.2.2, an approximation approach is used. It is clear from the introduction of the RETE algorithm in section 2.2.2.1, join operation and match operation are the two major operations separately performed in the beta and alpha network. Furthermore the cached intermediate result in alpha memory and beta memory is the major source for memory consumption in beta

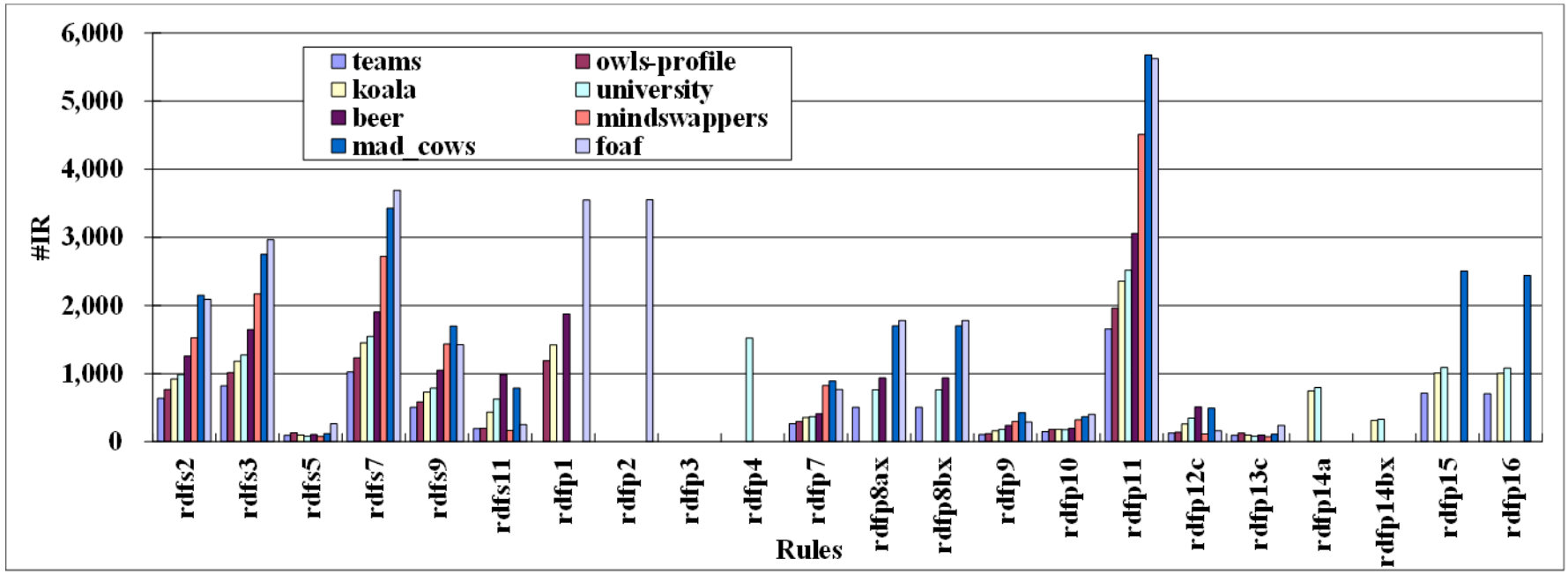
network. As discussed in section 2.2.2.3.1, join sequence optimization heuristics try to construct a better join sequences such that less join operations are required (so the less processing time is required) and also less intermediate results are generated, therefore in this evaluation the changes of *the number of joins* (#J) performed by a particular rule are used to represent the changes of *reasoning time in the beta network* of the rule. Similarly the changes of *number of matches* (#M) performed by a particular rule are used to represent the changes of *reasoning time in the alpha network* of the rule. Generally the greater the #M (#J) that are performed, the more time is required by the alpha network (beta network) of the rule (comparing to other rules). For memory, the changes of the *number of intermediate results* (#IR) generated in a rule are used to represent the changes of the *memory usage* required by the rule. The changes of *number of intermediate results generated in the alpha/beta network* of a rule ($\#IR_M$/$\#IR_J$ which are generated from successful matches/joins) are used to represent the changes of the *memory usage of the alpha/beta network* of the rule.

A major benefit of using the changes in #J/#M/#IR/$\#IR_M$/$\#IR_J$ to represent the changes of reasoning time/memory usage of the RETE network is that the #J/#M/#IR/$\#IR_M$/$\#IR_J$ can truly represent the performance of the algorithm independent of the specifications of the platform on which the RETE algorithm is running. As long as the algorithm does not change, the same results will always be produced regardless as to whether it is run on a powerful desktop machine or a resource-constrained sensor node. As a matter of fact this thesis is not the first research adopting these metrics. These metrics were adopted in many previous work to measure the performance of their RETE network [Miranker 1987, Wang and Hanson 1992, Ishida 1994].

6.2.3.1 Selective Rule Loading Algorithm

As discussed in the section 3.4.1 the rationale behind the *selective rule loading* algorithm is to remove unused rules from the rule set for a given ontology so that memory and processing power that would be allocated to them are saved. Figure 6-2a and Figure 6-2b separately show the number of intermediate results (#IR) generated by each rule when COROR-noncomposable and COROR-selective are used to reason over the eight selected ontologies. *Singleton rules*, i.e. rules with only one condition, are directly fired without caching intermediate results, and therefore they do not contribute to the #IR and they are excluded from both diagrams. The biopax, food and miniTambis are not included in the diagrams as manual termination was imposed for COROR-noncompoable leading to no results for them.


6,000
5,000
4,000
3,000
2,000
1,000
0
#IR
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
rdfs2
rdfs3
rdfs5
rdfs7
rdfs9
rdfs11
rdfp1
rdfp2
rdfp3
rdfp4
rdfp7
rdfp8ax
rdfp8bx
rdfp9
rdfp10
rdfp11
rdfp12c
rdfp13c
rdfp14a
rdfp1…
rdfp15
rdfp16
Rules


6,000
5,000
4,000
3,000
2,000
1,000
0
#IR
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
rdfs2
rdfs3
rdfs5
rdfs7
rdfs9
rdfs11
rdfp1
rdfp2
rdfp3
rdfp4
rdfp7
rdfp8ax
rdfp8bx
rdfp9
rdfp10
rdfp11
rdfp12c
rdfp13c
rdfp14a
rdfp14bx
rdfp15
rdfp16
Rules

For each tested ontology in Figure 6-2 there are some rules that have the same #IR for the two composition modes, however, the #IR drops to zero for some other rules. For example, for the team ontology, in total 7 rules drop down to zero when the *selective rule loading* algorithm is applied, i.e. rdfs12, rdfp1, rdfp2, rdfp3, rdfp4, rdfp14a, and rdfp14bx, however, the #IR remains the same for the other rules. This shows the *selective rule loading* algorithm generates a selective rule set, leading to no generated intermediate results for unloaded rules. This then raises a supplementary question that: is there any relationship between rules not selected and the memory reductions achieved. If such an indicative relationship exists, it may be possible to estimate the memory reduction that could be achieved without having to actually reason the ontology.

To investigate this question, the amount of memory reduction and the unselected rules for each ontology are listed both in absolute values (in byte) and relative values (in percentage), as given in Table 6-3. Note that the relevant memory reduction is computed as

$$\frac{M_{CN} - M_{CS}}{M_{CN}} \times 100\%$$

where M_{CN} is the memory usage for COROR noncomposable and M_{CS} is the memory usage for COROR selective.

As shown in the table, the absolute memory reduction does not show an obvious tendency with the number of unselected rules (shown in brackets). However, the relative memory reduction increases with the number of unloaded rules, from 17.42% for the university ontology with 4 unloaded rules to 59.01% for the mindswappers ontology 11 unloaded rules. One obvious relationship is the relative memory reduction increase with the number of unselected rules.

Table 6-3: Memory reduction achieved by the COROR-selective.

Ontology	**Memory reduction**		**Unloaded rules (number of unloaded rules)**
	Absolute (byte)	**Relative (percentage)**	
university	186124	17.42%	rdfs12, rdfp1, rdfp2, rdfp3 (**4**)
Koala	265592	26.64%	rdfs12, rdfp2, rdfp3, rdfp4, rdfp8ax, rdfp8bx (**6**)
Teams	223856	31.22%	rdfs12, rdfp1, rdfp2, rdfp3, rdfp4, rdfp14a, rdfp14bx (**7**)
mad_cows	697280	32.64%	rdfs12, rdfp1, rdfp2, rdfp3, rdfp4, rdfp14a, rdfp14bx (**7**)
Foaf	739704	30.63%	rdfs12, rdfp3, rdfp4, rdfp14a, rdfp14bx, rdfp15, rdfp16 (**7**)
Beer	487036	37.92%	rdfs12, rdfp2, rdfp3, rdfp4, rdfp14a, rdfp14bx,

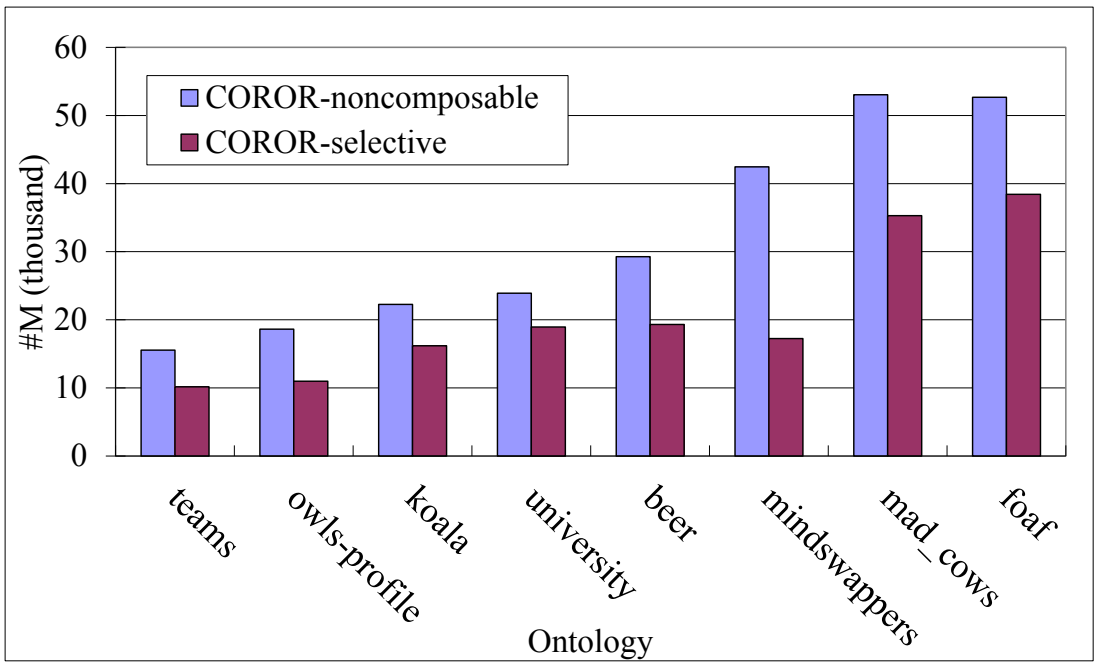
			rdfp15, rdfp16 (**8**)
owls-profile	407296	47.00%	rdfs12, rdfp2, rdfp3, rdfp4, rdfp8ax, rdfp8bx, rdfp14a, rdfp14bx, rdfp15, rdfp16 (**10**)
mindswappers	1062404	59.01%	rdfs12, rdfp1, rdfp2, rdfp3, rdfp4, rdfp8ax, rdfp8bx, rdfp14a, rdfp14bx, rdfp15, rdfp16 (**11**)

Further investigation into the percentage of #IR contributed by each rule when the COROR-noncomposable is used (as shown in Figure 6-3) indicates that all unselected rules occupy a relative constant percentage of the total #IR generation *regardless of the ontology reasoned*: the rule rdfp1 occupies 8.15% on average with standard deviation 0.23%; this percentage is 8.15%/0.24% for rdfp2, 4.07%/0.12% for rdfp3, 8.15%/0.23% for rdfp4, 4.09%/0.12% for rdfp8ax, 4.09%/0.12% for rdfp8bx, 4.15%/0.11% for rdfp14a, 1.67%/0.13% for rdfp14bx, 5.78%/0,19 for rdfp15 and 5.74%/0.18% for rdfp16. A possible use of this observation is to *heuristically estimate* the memory reduction for an ontology without having to perform reasoning. For example by running only the *selective rule loading* algorithm (without reasoning) against the university ontology, a set of four unselected rules are identified, which are rdfs12 (0%), rdfp1 (8.15%), rdfp2 (8.15%), and rdfp3 (4.07%); by adding up the corresponding percentage the estimated relative memory reduction is 20.37% for the university ontology (of which the measured memory reduction is 17.42%). Similarly, the estimated memory reduction is 28.55% for koala (of which the measured memory reduction is 26.64%), 34.34% for teams (31.22%), 34.34% for mad_cows (32.64%), 33.64% for foaf (30.63%), 37.75% for beer (37.92%), 43.57% for owls-profile (47.00%), 51.72% for mindswappers (59.01%).

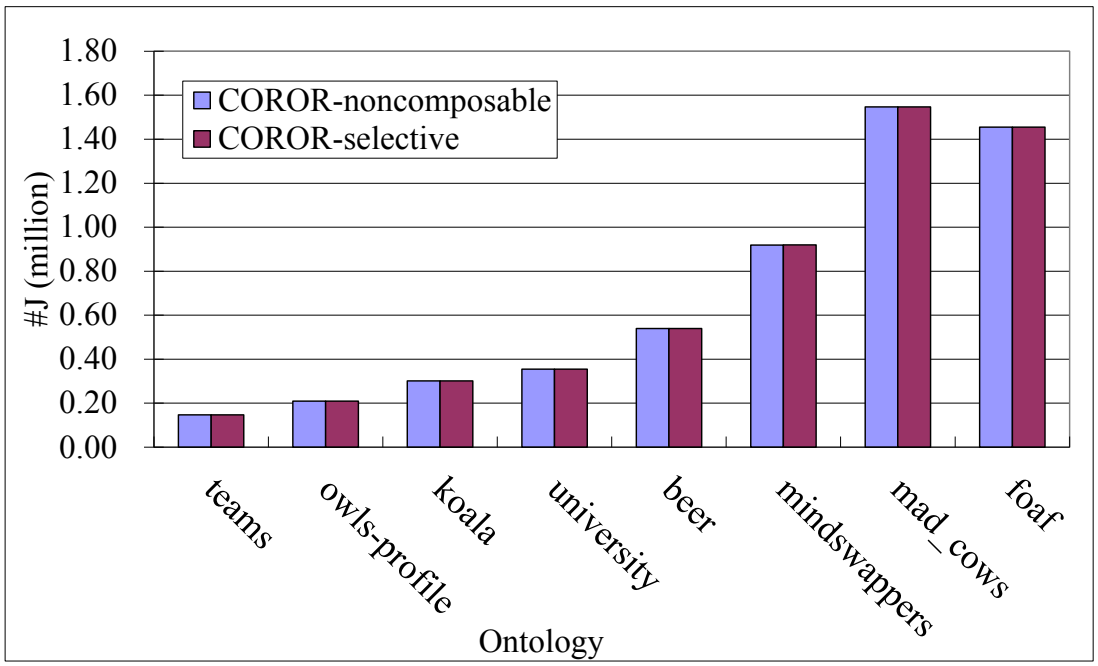
Considering the diversity of the tested ontologies, this approach may be generally applicable to other ontologies, but it has not yet been formally proven or verified in this thesis. Since it is a by-product of this research, its validity does not contribute to any of the evaluation objectives identified, i.e. comparing COROR-noncomposable and composable modes, and comparing between COROR and state of the art rule-entailment reasoners. A potential usage of this approach is to facilitate fast ontology engineering without performing reasoning in circumstances when a memory limitation is imposed.

16.00%
14.00%
12.00%
10.00%
8.00%
6.00%
4.00%
2.00%
0.00%
Percentage of #IR
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
rdfs2
rdfs3
rdfs5
rdfs7
rdfs9
rdfs11
rdfp1
rdfp2
rdfp3
rdfp4
rdfp7
rdfp8ax
rdfp8bx
rdfp9
rdfp10
rdfp11
rdfp12c
rdfp13c
rdfp14a
rdfp14bx
rdfp15
rdfp16
Rules

Figure 6-4a and Figure 6-4b respectively compare COROR-noncomposable and COROR-selective with respect to the number of matches (#M) and the number joins (#J) performed to reason over the eight selected ontologies. COROR-selective reduces #M for all tested ontologies (Figure 6-4a), however the #J remains the same for both modes (Figure 6-4b). This would suggest that all the time reduction is a result of the decrease of the #M.



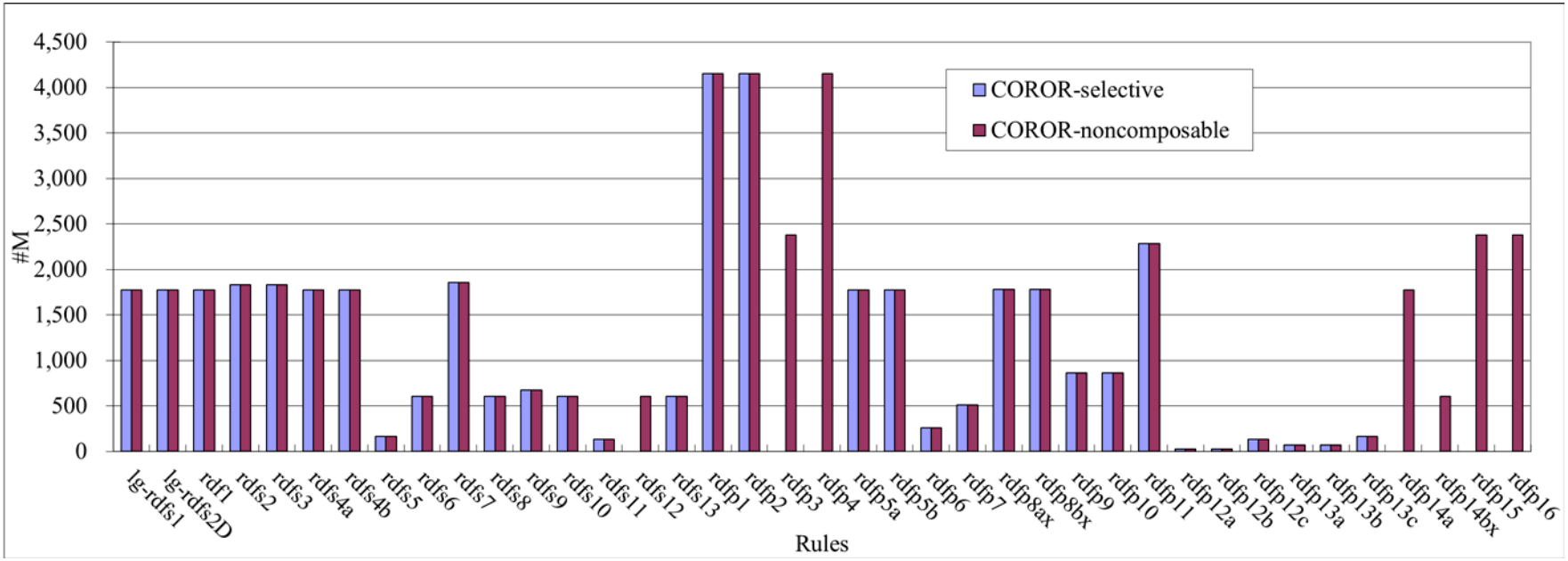
(a) Comparison between #M of COROR-noncomposable and COROR-selective

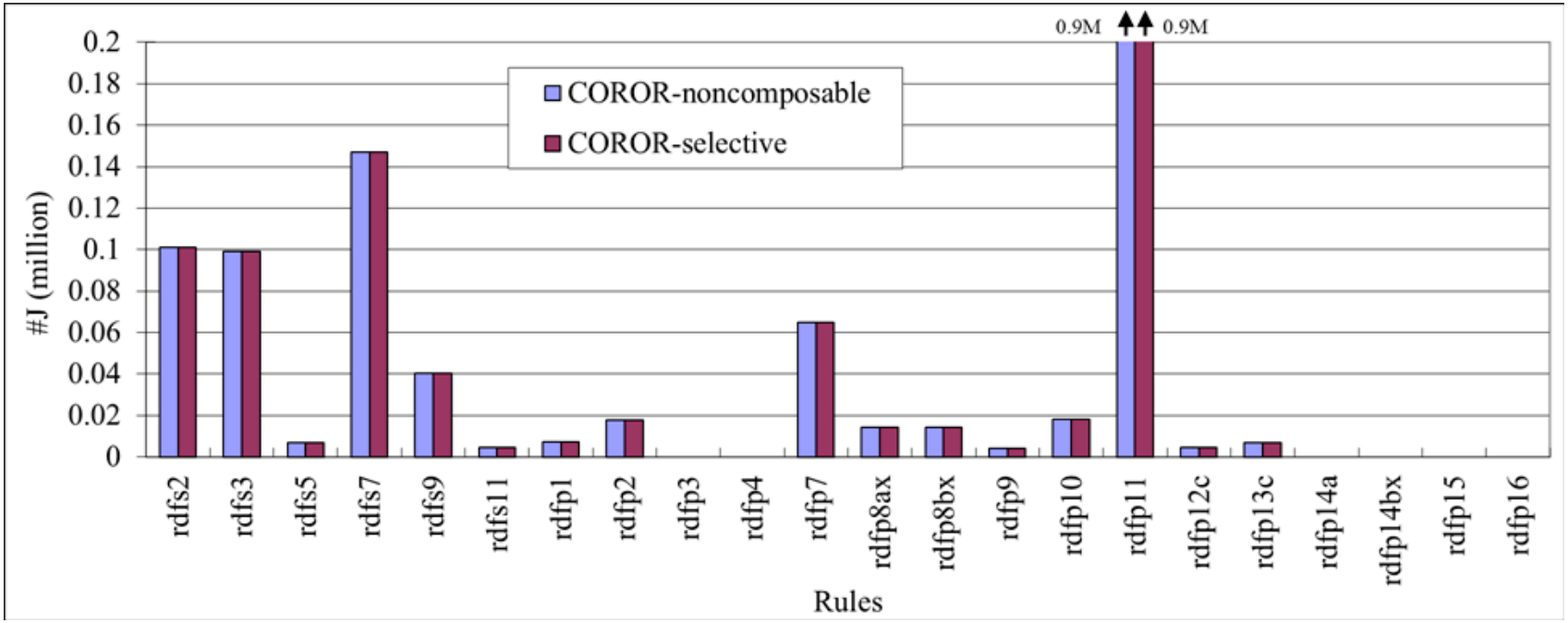


(b) Comparison between #J of COROR-noncomposable and COROR-selective

Figure 6-4: Comparison between #M/#J of COROR-noncomposable and COROR-selective

To further explain the observation found in Figure 6-4, without loss of generality, the #M/#J generated by all rules for reasoning the foaf ontology are compared between COROR-noncomposable and COROR-selective. As shown in Figure 6-5 COROR-selective reduces the #M of all unselected rules to zero, including rdfs12, rdfp3, rdfp4, rdfp14a, rdfp14bx, rdfp15 and rdfp16, showing their unselection. The #M of selected rules, e.g. rdfp1, remains the same. As shown in Figure 6-6 there is no difference between #J performed by COROR-noncomposable and that by COROR-selective. As one might have already noticed the #J for all unselected rules is zero even for COROR-noncomposable where all rules are loaded into memory, which indicates that no join operations are performed. By looking into the rules it is found that these unselected rules are quite optimized in terms of join sequences: singleton rules do not need joins; for rules with two conditions, e.g. rdfp3 and rdfp4, the lack of matched facts for one condition will cause no join operations to be performed for the rule; for rules with more than three conditions, e.g. rdfp14a, rdfp14bx, rdfp15 and rdfp16, expressivity constructs (OWL constructs that determines the selection of a rule, e.g. owl:hasValue, owl:someValuesFrom, owl:allValuesFrom; refer to section 3.4.1) are placed in the first condition of the join sequence, and therefore no join operations are needed since no facts are matched to the first condition of these rules. For this particular rule set it is the reduction on the number of matches that causes the reduction of reasoning time. However it is envisaged that in a less optimized rule set, e.g. expressivity constructs are not placed in the front of unselected rules, the application of the *selective rule loading* algorithm will also reduce the #J.


4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
#M
COROR-selective
COROR-noncomposable
lg-rdfs1
lg-rdfs2D
rdf1
rdfs2
rdfs3
rdfs4a
rdfs4b
rdfs5
rdfs6
rdfs7
rdfs8
rdfs9
rdfs10
rdfs11
rdfs12
rdfs13
rdfp1
rdfp2
rdfp3
rdfp4
rdfp5a
rdfp5b
rdfp6
rdfp7
rdfp8ax
rdfp8bx
rdfp9
rdfp10
rdfp11
rdfp12a
rdfp12b
rdfp12c
rdfp13a
rdfp13b
rdfp13c
rdfp14a
rdfp14bx
rdfp15
rdfp16
Rules


0.9M
0.9M
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
#J (million)
COROR-noncomposable
COROR-selective
rdfs2
rdfs3
rdfs5
rdfs7
rdfs9
rdfs11
rdfp1
rdfp2
rdfp3
rdfp4
rdfp7
rdfp8ax
rdfp8bx
rdfp9
rdfp10
rdfp11
rdfp12c
rdfp13c
rdfp14a
rdfp14bx
rdfp15
rdfp16
Rules

6.2.3.2 Two-Phase RETE Algorithm

Unlike the *selective rule loading* algorithm which only removes unnecessary rules and then uses a standard RETE network, the *two-phase RETE* algorithm can take an unmodified rule set but heuristically constructs a customized RETE network for the particular ontology taking into account the semantics of the ontology, particularly the OWL constructs contained by the ontology. As already discussed in section 3.4.2, the *two-phase RETE* algorithm interrupts the RETE construction process using initial fact matching, and information that is hard to collect without matching is then collected. According to the information, a customized beta network is then built for the particular ontology.

This subsection discusses how the application of *two-phase RETE* algorithm impacts on the memory usage and reasoning time required by the alpha network and the beta-network. Note that this discussion is based on results collected on desktop in order that more data can be analysed. The exact same COROR implementation as the one used to produce the results in Figure 6-1 is used in this analysis. However rather than running on Sun SPOT, it runs on the desktop machine as mentioned above in section 6.2.2. Since #M, #J and #IR are only relevant to algorithms and are independent of the platform on which the algorithms are running, the same #M, #J and #IR are generated.

Figure 6-7 compares the number of intermediate results generated in both alpha network ($\#IR_M$) and beta network ($\#IR_J$) between COROR-noncomposable and COROR-two-phase. The change related to $\#IR_M$ shows the benefit from alpha network node sharing and the change of $\#IR_J$ shows the benefit from join reordering. As shown in the figure, COROR-two-phase generates much less $\#IR_M$ than that generated by COROR-noncomposable, however there are barely no changes between the $\#IR_J$ generated by COROR-noncomposable and that generated by COROR-two-phase. This indicates that the major memory reduction comes from the reduction of $\#IR_M$, i.e. the sharing of common alpha nodes, however, nearly no memory reduction is from the decrease of $\#IR_J$, i.e. the join sequence reordering.

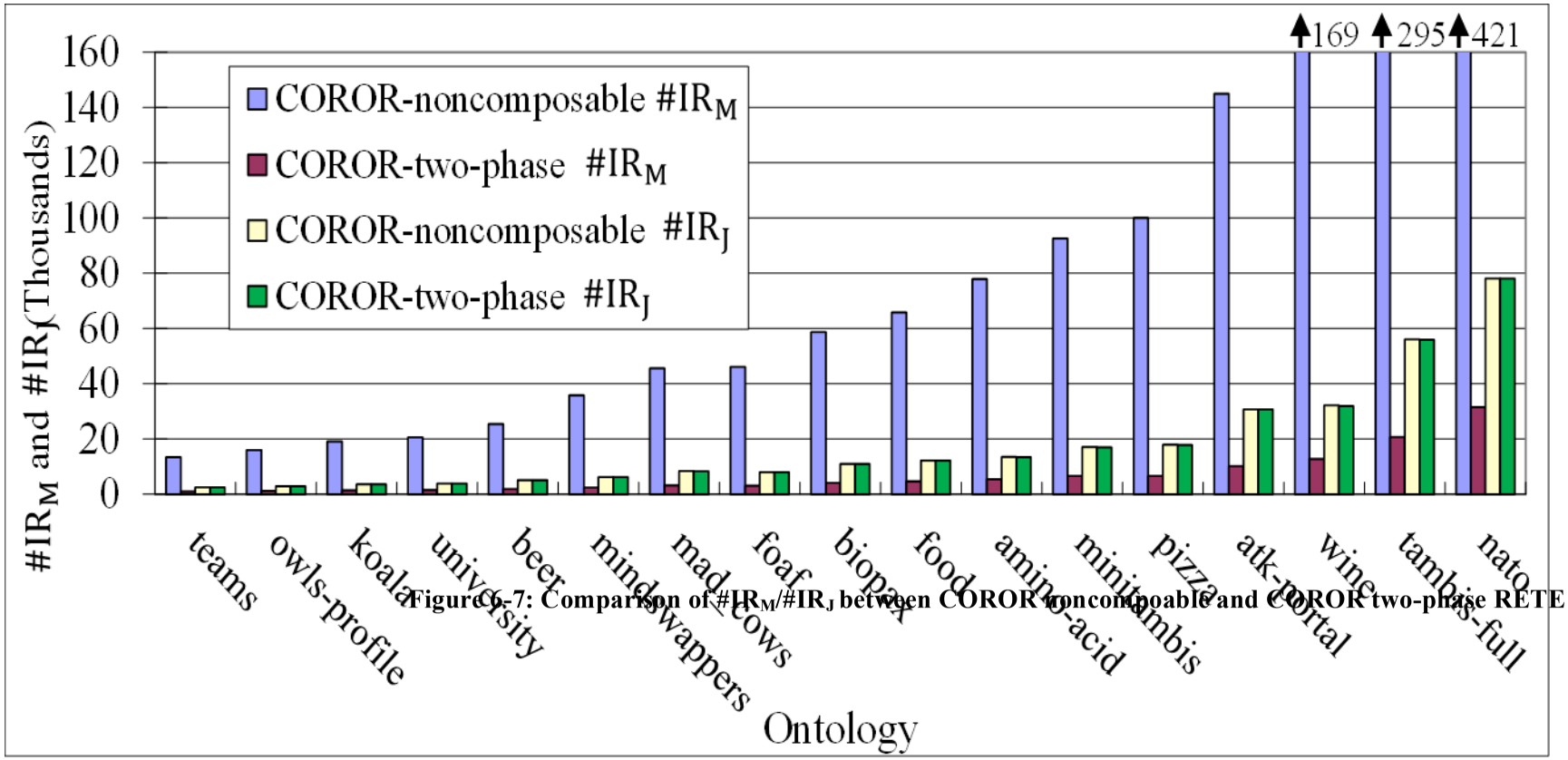


Figure 6-7: Comparison of $\#IR_M/\#IR_J$ between COROR noncomposable and COROR two-phase RETE

To further investigate the reason for the very small reductions on $\#IR_J$, an in-depth investigation into the heuristically optimized RETE network was carried out. Condition nodes in the alpha network are highly shared among rules. All rules, except for the rule rdfs12 and rdfs13, share at least one alpha node with other rules. Some alpha nodes are shared by more than two rules, e.g. the condition (*?v* owl:sameAs *?w*) is shared by rdfp6, rdfp7, rdfp9, rdfp10 and rdfp11. The wildcard condition, i.e. (*?v ?p ?l*), is shared by 21 rules. The highly shared alpha network enables only one set of intermediate results to be generated for all the conditions sharing the node, leading to the large reduction in $\#IR_M$.

However the beta network shows very small changes after the join sequences being reordered by heuristics in the *two-phase RETE* algorithm. Inspection into the join sequences before and after the application of each heuristic shows only two rules, i.e. rdfp11 and rdfp15, have their join sequences reordered by the most specific condition first heuristic and no join sequence is reordered by the connectivity heuristic. The join sequences before and after the application of the heuristic are given below.

The join sequence of rdfp11 after reordering

(*?u* owl:sameAs *?up*), **(*?u ?p ?v*)**, (*?v* owl:sameAs *?vp*)

The join sequence of rdfp11 before reordering

(*?u ?p ?v*), **(*?u* owl:sameAs *?up*)**, (*?v* owl:sameAs *?vp*)

The join sequence of rdfp15 after reordering

(*?v* owl:someValuesFrom *?w*), (*?v* owl:onProperty *?p*), **(*?x* rdf:type *?w*)**, **(*?u ?p ?x*)**

The join sequence rdfp15 before reordering

(*?v* owl:someValuesFrom *?w*), (*?v* owl:onProperty *?p*), **(*?u ?p ?x*)**, **(*?x* rdf:type *?w*)**

The join sequences of the other rules remain the same due to the join sequences of pD* entailment rules as selected, are already well optimized manually by the original rule authors, e.g. conditions are arranged in a sequence that a more specific condition is placed in front of a less specific condition, and in addition the join sequences are already well

connected. It is evident that swapping the positions of (*?u* owl:sameAs *?up*) and (*?u ?p ?v*) in rdfp11 do not change the $\#IR_J$ since they are the first two conditions in the join sequence (whatever the order of the first two conditions they will form the first join in the join sequence). Reordering the join sequence of rdfp15 leads to some reduction of $\#IR_J$. However this reduction is too small to be observable on diagram. In fact the $\#IR_J$ generated by the rdfp15 accounts for a very small part of the total generated intermediate results in the whole RETE network (only 0.91% for the wine ontology and even smaller for the other tested ontology) and therefore its reduction is not very obvious.

To study the effect of the *two-phase RETE* algorithm on reasoning time, the comparison of #J/#M between COROR-noncomposable and COROR-two-phase is illustrated respectively in Figure 6-8 and Figure 6-9. In a similar way, due to the highly shared alpha network as discussed above with regard to $\#IR_M$, the #M reduces by a large amount for all ontologies, which then indicates the large effects of the alpha node sharing heuristic on the reasoning time spent on performing match operations. However the small reductions of #J for all tested ontologies suggest that for this rule set, the selected the join sequence optimization heuristics do not have a very obvious effect on reducing the #J and therefore the reasoning time spent on performing join operations. This can also be explained by the observation that the join sequences for the rules except for rdfp11 and rdfp15 are already manually optimized in terms of the join sequence optimization heuristics, therefore leading to no changes in the majority of the join sequences of the selected rule set. However, different scales are used in Figure 6-8 and Figure 6-9, such that the absolute reduction of #J is much larger than that of #M, e.g. for the amino-acid ontology the reduction of the #J is 0.2M while this is 0.08M for the #M.

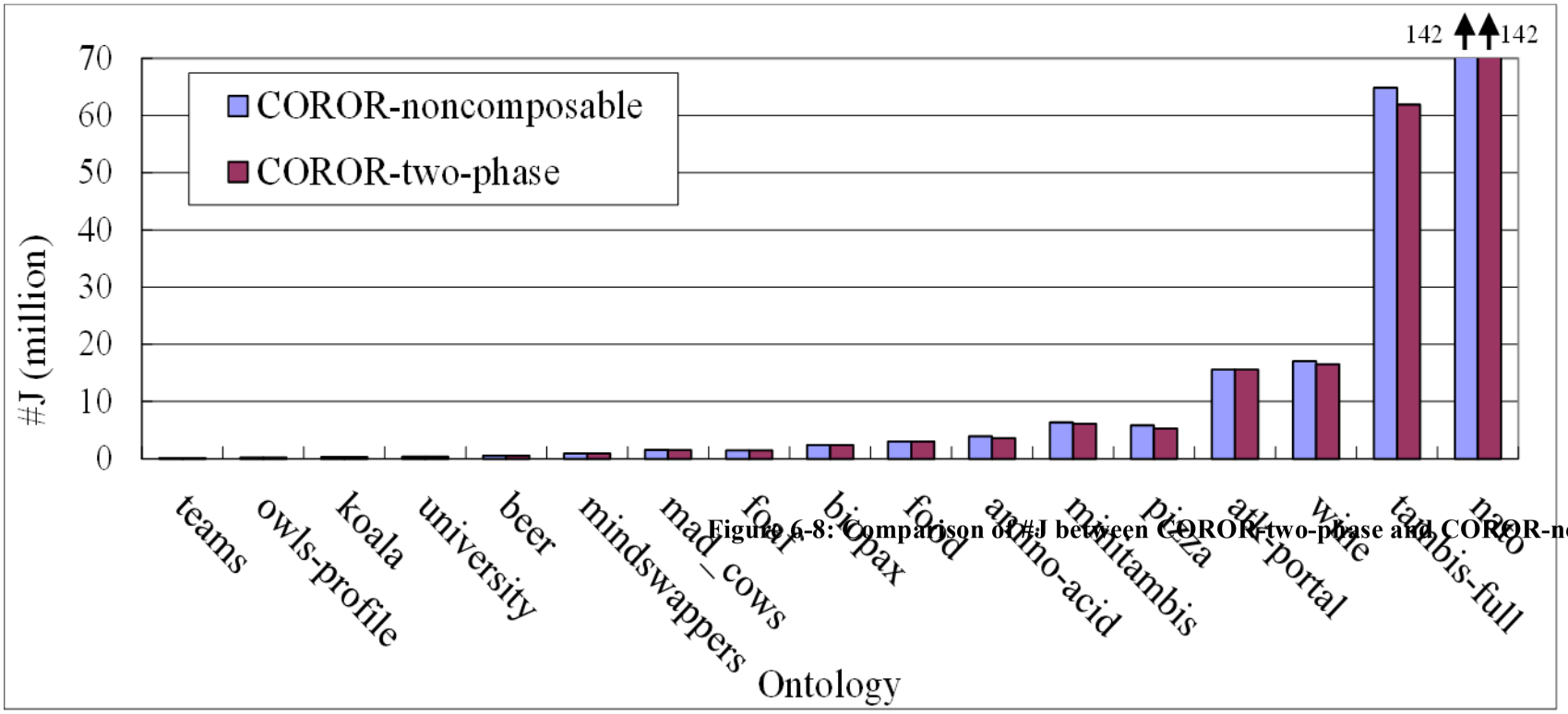
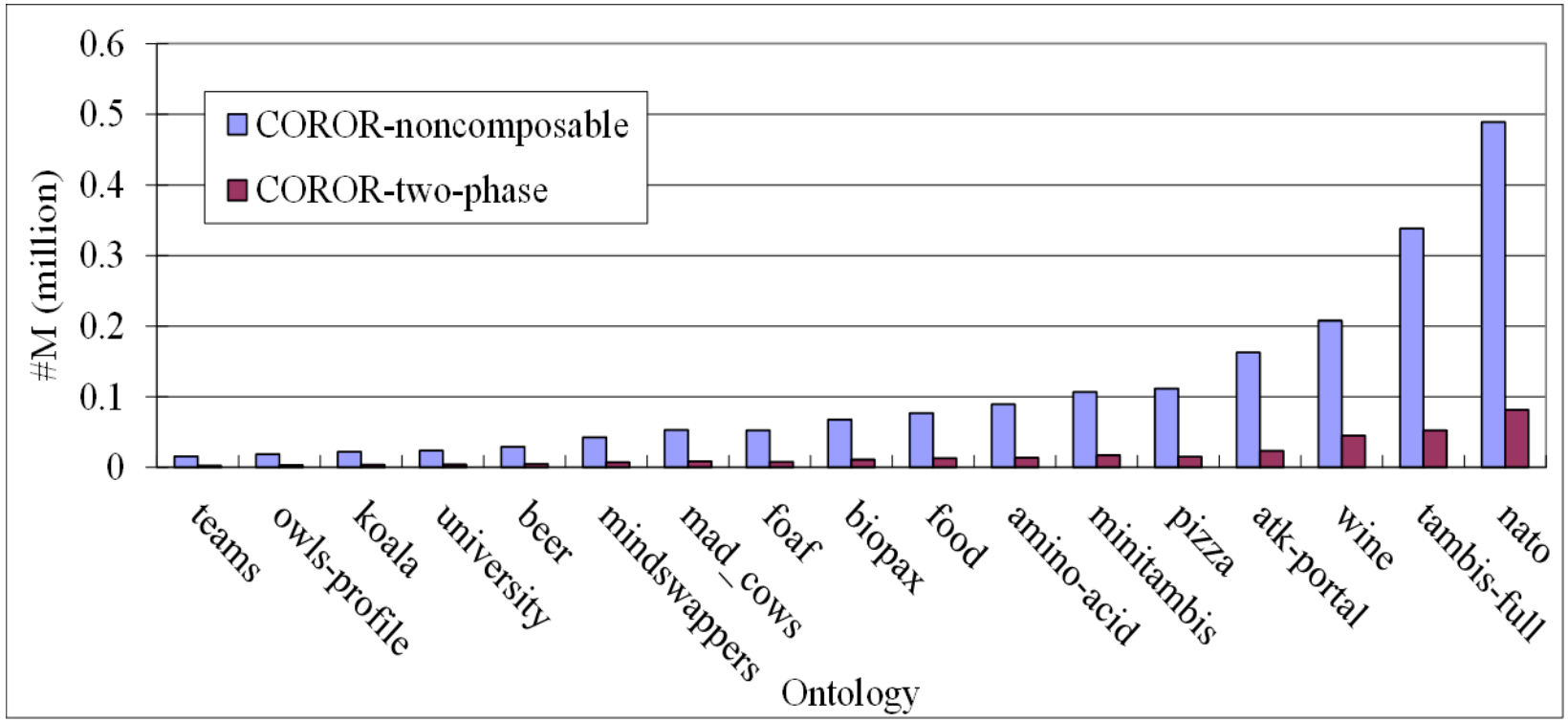


Figure 6-8: Comparison of #J between COROR-two-phase and COROR-noncomposable.


0.6
0.5
0.4
0.3
0.2
0.1
0
#M (million)
COROR-noncomposable
COROR-two-phase
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
biopax
food
amino-acid
minitambis
pizza
atk-portal
wine
tambis-full
nato
Ontology

All other ontologies (except for *owls-profile*, *beer*, *mindswappers*, *foaf*, *food*, *atk-portal* and *nato*) have reductions on both match operations and join operations. Therefore the time reductions (refer to Table 6-2) for them come from both the reduction in the number of match operations in the highly shared alpha network and the reduction of the number of join operations caused by heuristically reordering the rule rdfp15. As the time required by per join operation is different from that required by per match operation, it is not possible to determine which part contributes more to the total time reduction. The above listed ontologies (owls-profile, beer, mindswappers, foaf, food, atk-portal and nato) do not include owl:someValuesFrom and therefore no joins are performed for rdfp15 which is the only source for the reduction of #J. For those ontologies the time reductions come only from their highly shared alpha networks.

It appears from the above analysis that the *two-phase RETE* algorithm does not obviously optimize the beta network, leading to relatively small time reductions (represented by the small reductions of #J) and the small memory reductions (represented by the small reductions in $\#IR_J$). The main reason for this is the use of a manually optimized pD* rule set in the evaluation. Rules are quite optimized join sequences in terms of the heuristics used in this research: the join sequences are ordered with specific condition in front of less specific conditions in most times and joining conditions are connected. As will be shown later on, this is not a general case.

In order to show that the *two-phase RETE* algorithm can optimize the join sequence in general cases, leading to more time and memory reduction, three rules from the pD* rule set were manually reordered to use different join sequences (renamed as rdfp1m, rdfp2m and rdfp4m, as listed in Figure 6-10). The modified rules bring forward more general conditions (wildcard conditions) to the front of their join sequences, making them less optimized. In fact it is often that domain experts would author rules with a more general condition at the start of the join sequence, which would then lead to the long-chain effect. The memory usage and reasoning time required by COROR-two-phase and COROR-noncomposable to reason over the same set of ontology as selected in Figure 6-1 are tested again, however both the original and the modified rule set are used. The same experiment settings are used and the same COROR implementation is used. Results are illustrated in Figure 6-11 and Figure 6-12. Note that for brevity in Figure 6-11 and Figure 6-12 , COROR-noncomposable using the modified rule set is represented as “COROR-noncomposable modified” in the legends and the same rule applies to other COROR configurations.

[rdfp1m: (*?u ?p ?v*), (*?u ?p ?w*), (*?p* rdf:type owl:FunctionalProperty), notLiteral(*?v*) -> (*?v* owl:sameAs *?w*)]

[rdfp2m: (*?u ?p ?w*), (*?v ?p ?w*), (*?p* rdf:type owl:InverseFunctionalProperty) -> (*?u* owl:sameAs *?v*)]

[rdfp4m: (*?u ?p ?v*), (*?v ?p ?w*), (*?p* rdf:type owl:TransitiveProperty) -> (*?u ?p ?w*)]

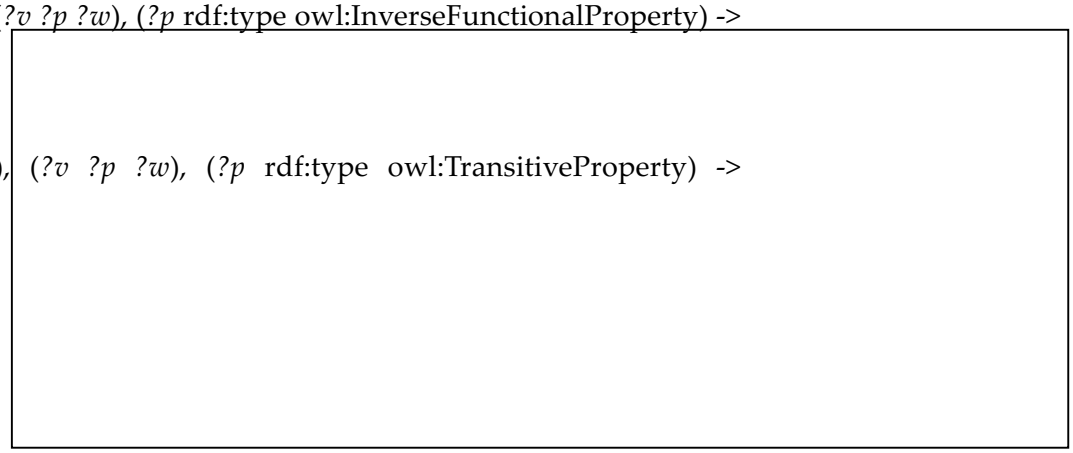
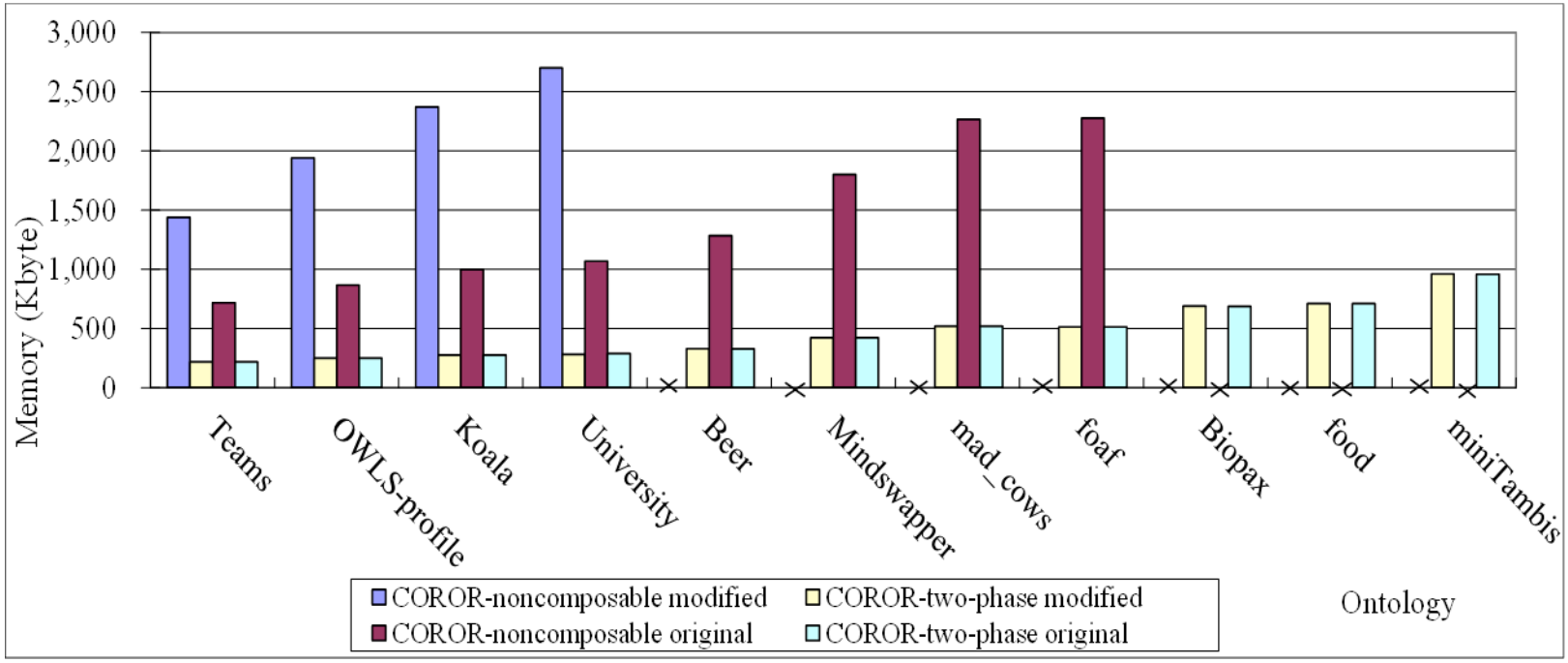


Figure 6-10: Modified rule rdfp1, rdfp2 and rdfp4.


3,000
2,500
2,000
1,500
1,000
500
0
Memory (Kbyte)
Teams
OWLS-profile
Koala
University
Beer
Mindswapper
mad_cows
foaf
Biopax
food
miniTambis
COROR-noncomposable modified
COROR-noncomposable original
COROR-two-phase modified
COROR-two-phase original
Ontology

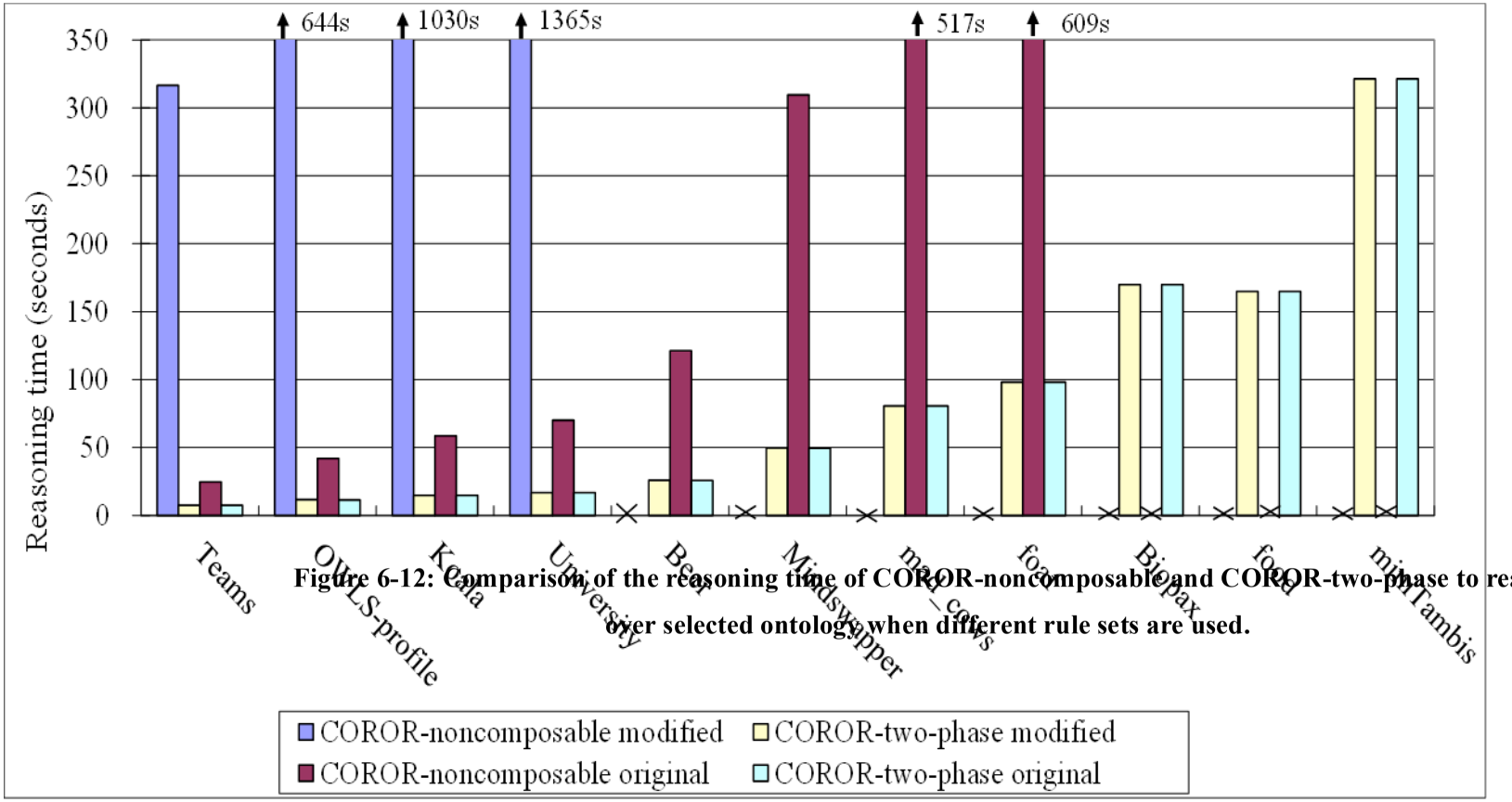
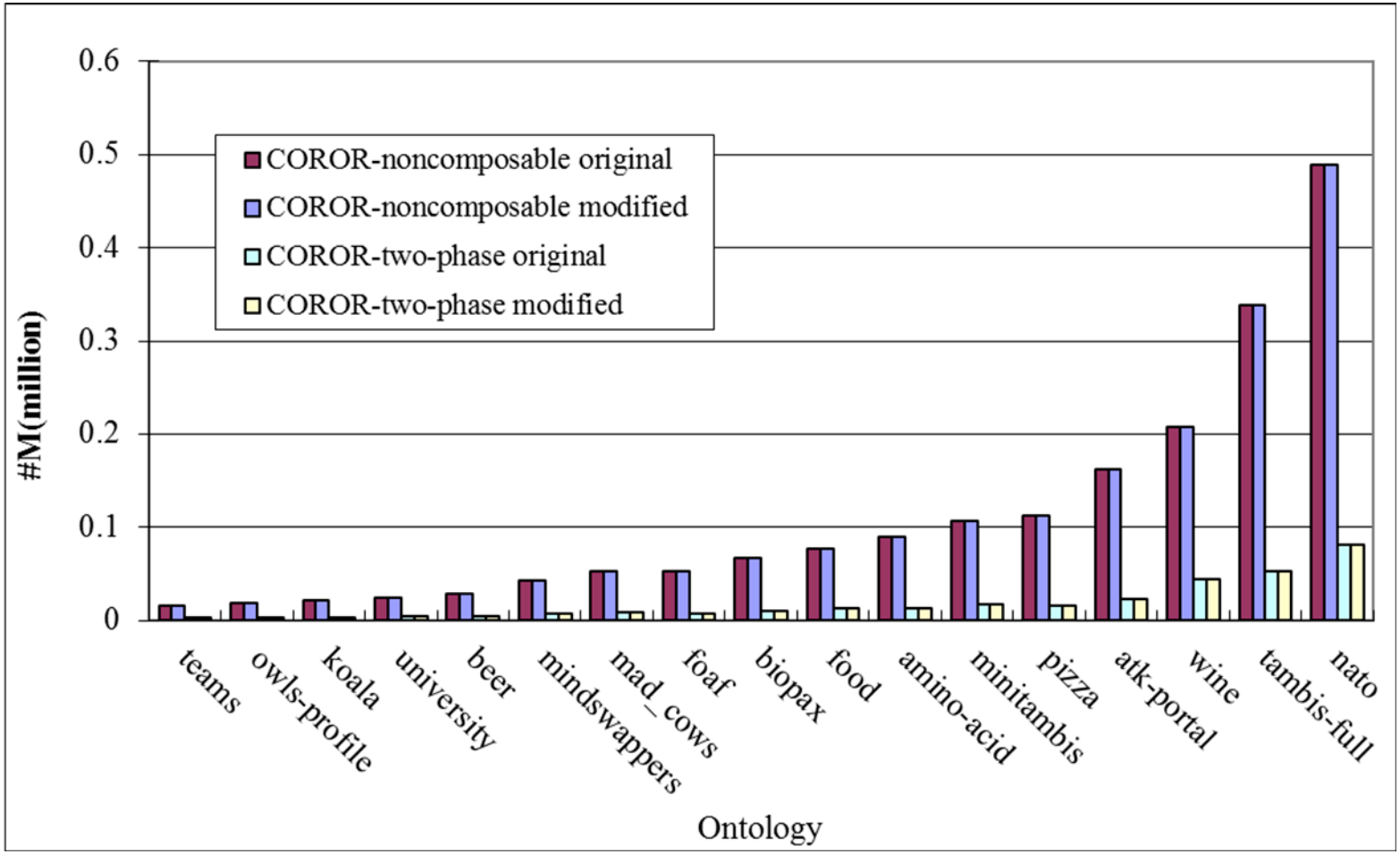
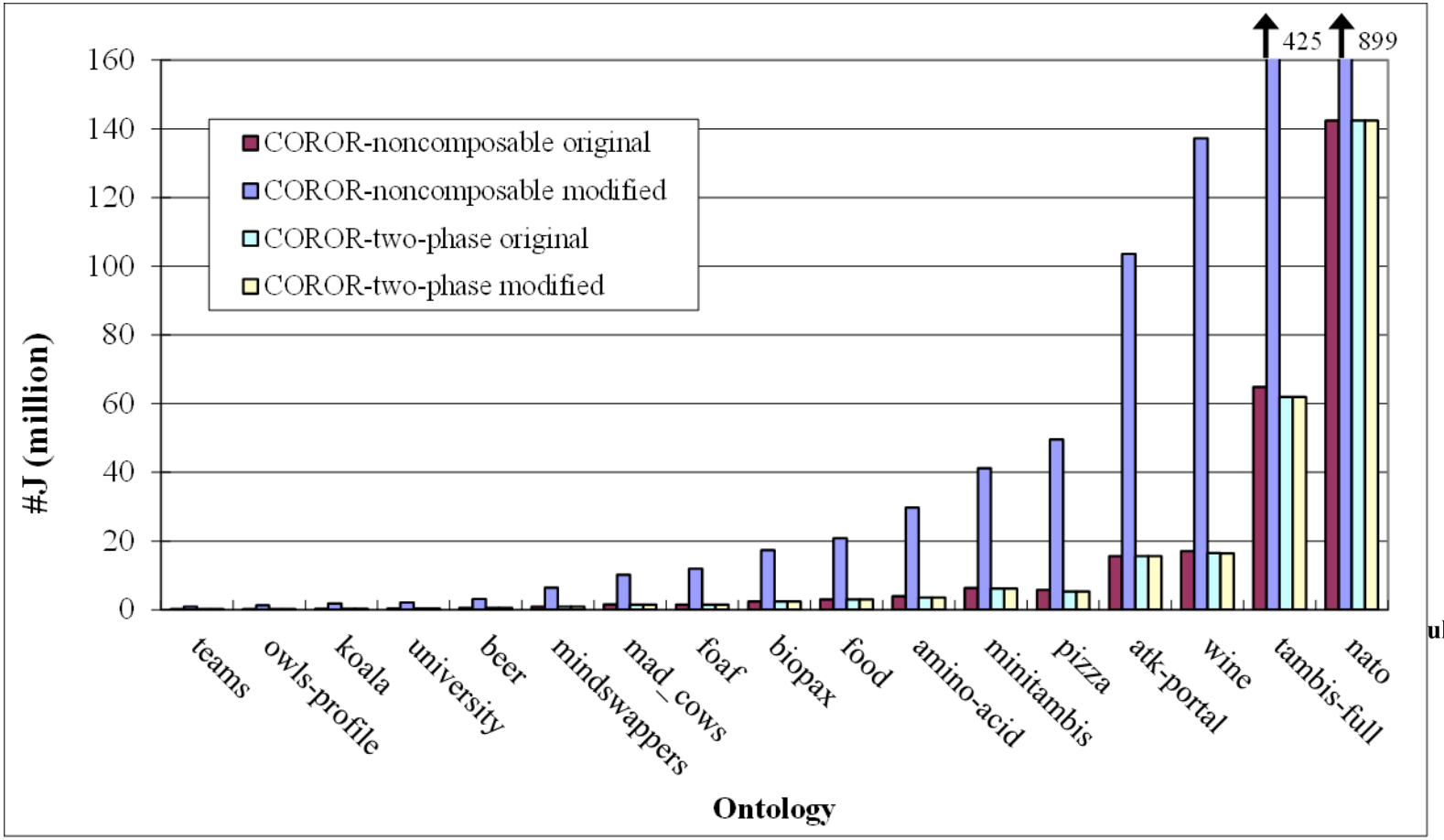


Figure 6-12: Comparison of the reasoning time of COROR-noncomposable and COROR-two-phase to reason over selected ontology when different rule sets are used.

As can be seen from Figure 6-11 and Figure 6-12 , the COROR-noncomposable uses a lot more memory and reasoning time when the modified rule set is used. Some ontologies such as beer, mindswappers, mad_cows and foaf that have valid measurements in Figure 6-1 require manual terminations for the modified rule set. However the memory usage and reasoning time required by the COROR two-phase stays unchanged for both the original and the modified rule set.

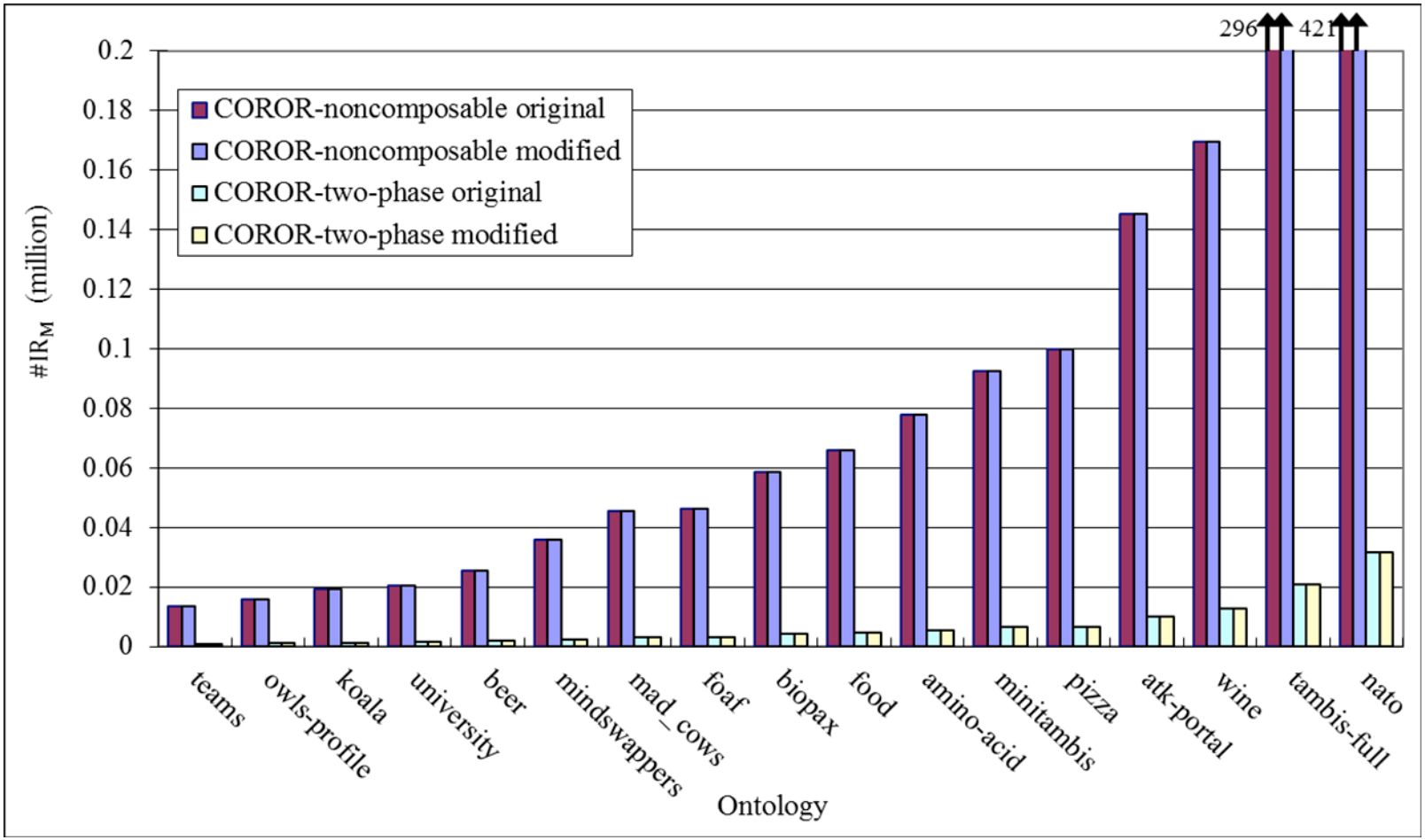
A similar investigation of the #IR and #J/#M is also performed on COROR-noncomposable and COROR-two-phase in order to show how the *two-phase RETE* algorithm affects on the reduction of memory usage and reasoning time when the modified rule set is used. Insight into #M (Figure 6-13) indicates that the use of a modified rule set does not lead to changes in the #M for both COROR-noncomposable and COROR-two-phase. This is because only the join sequences are changed but the conditions remain the same for the modified rule set and therefore the alpha network remains unchanged for the modified rule set, leading to the same amount of time used in performing the match operations. The #J of COROR-noncomposable has greatly increased for the modified rule set as sub-optimal join sequences are used. However, the #J remains the same for COROR two-phase (as indicated in Figure 6-14), which explains the large reduction of the reasoning time for COROR-two-phase when the modified rule set is used (comparing to COROR-noncomposable when the modified rule set is used, as illustrated in Figure 6-12). Further investigation into the rule join sequences after the application of join sequence reordering heuristics indicates that join sequences of the modified rules are reordered into the optimal sequences as given in the original rules. This suggests that the heuristics used in the *two-phase RETE* algorithm can optimize join sequences automatically using the information collected in the first phase without changing the semantics of rules, which otherwise requires manual optimization from a rule expert. In fact people who write custom/domain-specific rules are usually not rule experts and hence they could not do this optimization manually, which emphasizes the benefit of the *two-phase RETE* algorithm to automatically optimize the join sequences of rules. To summarise, the results presented here using the modified rule set show that the heuristics used in the *two-phase RETE* algorithm can optimize join sequences of rules in general cases leading to a great reduction of the #J and hence the reasoning time spent on beta network.

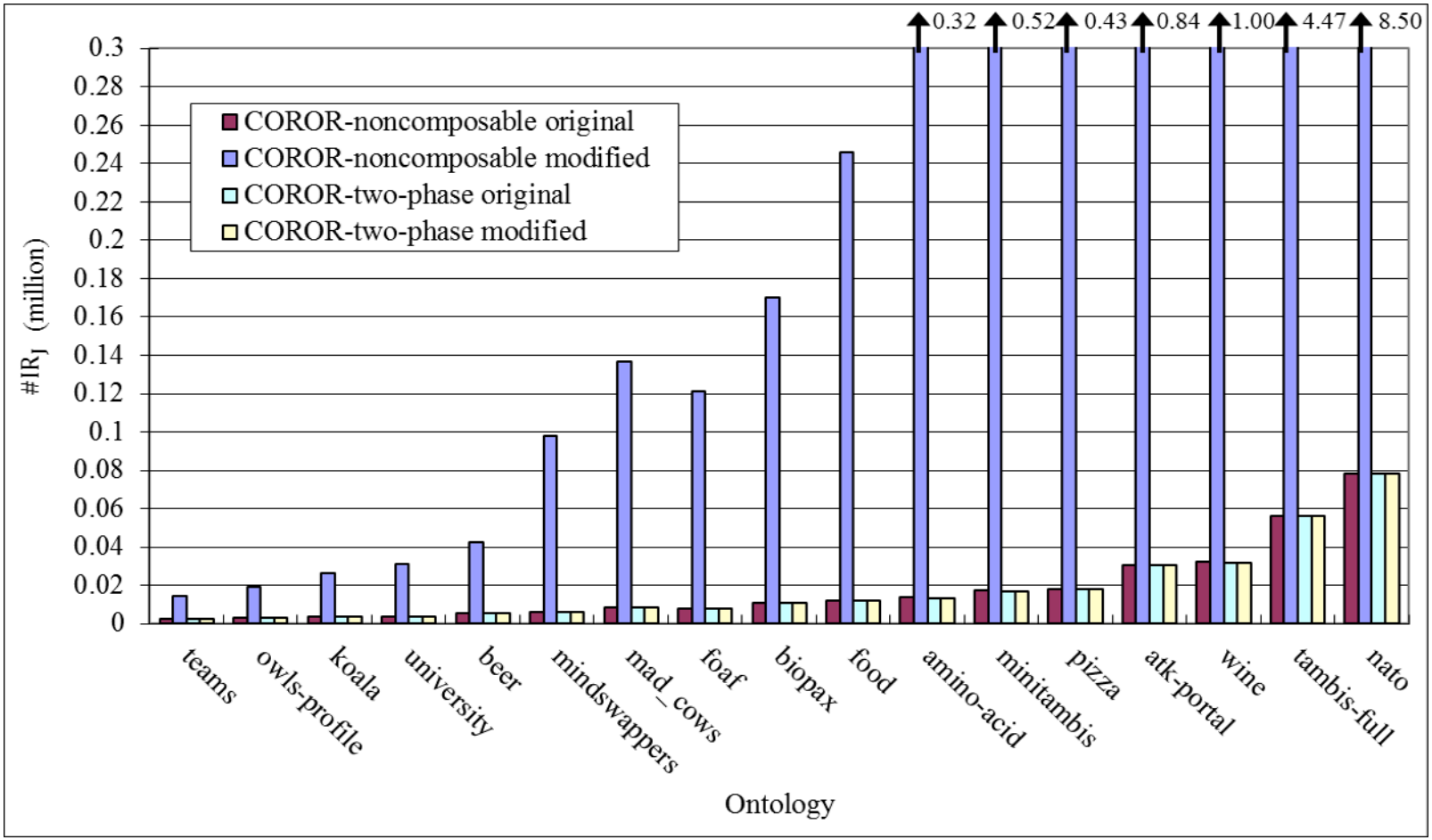

0.6
0.5
0.4
0.3
0.2
0.1
0
#M(million)
COROR-noncomposable original
COROR-noncomposable modified
COROR-two-phase original
COROR-two-phase modified
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
biopax
food
amino-acid
minitambis
pizza
atk-portal
wine
tambis-full
nato
Ontology


160
140
120
100
80
60
40
20
0
#J (million)
425
899
COROR-noncomposable original
COROR-noncomposable modified
COROR-two-phase original
COROR-two-phase modified
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
biopax
food
amino-acid
minitambis
pizza
atk-portal
wine
tambis-full
nato
Ontology

ule

The analysis of the $\#IR_M/\#IR_J$ show similar results as the corresponding investigation of #M/#J conducted above. The $\#IR_M$ is the same for both rule sets (Figure 6-16) as the conditions are not changed. Hence the alpha network remains the same for the modified rule set. The $\#IR_J$ for COROR-noncomposable increases greatly when the modified rule set is used (Figure 6-16) since sub-optimal join sequences are used. However the $\#IR_J$ for COROR-two-phase remains the same for both the modified rule set and the original rule set since the sub-optimal join sequences in the modified rule set have been automatically reordered into the optimal join sequences, as those in the original rule set manually optimized by rule experts. Similar conclusions can be drawn that the *two-phase RETE* algorithm can automatically optimize the join sequences of general rule sets, leading to great reduction of the $\#IR_J$. This also explains the great memory reduction of COROR-two-phase when using the modified rule set (Figure 6-11).


296
421
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
#IR_M (million)
COROR-noncomposable original
COROR-noncomposable modified
COROR-two-phase original
COROR-two-phase modified
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
biopax
food
amino-acid
minitambis
pizza
atk-portal
wine
tambis-full
nato
Ontology


0.32
0.52
0.43
0.84
1.00
4.47
8.50
0.3
0.28
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
#IRJ (million)
COROR-noncomposable original
COROR-noncomposable modified
COROR-two-phase original
COROR-two-phase modified
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
biopax
food
amino-acid
minitambis
pizza
atk-portal
wine
tambis-full
nato
Ontology

6.2.3.3 Hybrid Algorithm

Since the *selective rule loading* algorithm and the *two-phase RETE* algorithm do not affect each other in any way, it is possible to apply both optimizations at the same time. COROR hybrid uses a hybrid algorithm that combines the *selective rule loading* algorithm and the *two-phase RETE* algorithm by first generating a selected rule set and then applying the *two-phase RETE* algorithm using the selected rule set. However as illustrated in Figure 6-1, in this experiment this combination does not gain a lot more time or memory reduction comparing to COROR two-phase. In fact Table 6-2 indicates that for all selected ontologies, COROR-hybrid only uses from 10KB to 20KB less memory than that required by COROR-two-phase. The reasoning time required by COROR-hybrid is almost similar as required by COROR-two-phase, and for some ontologies such as Biopax and mad_cows COROR-hybrid requires even slightly more.

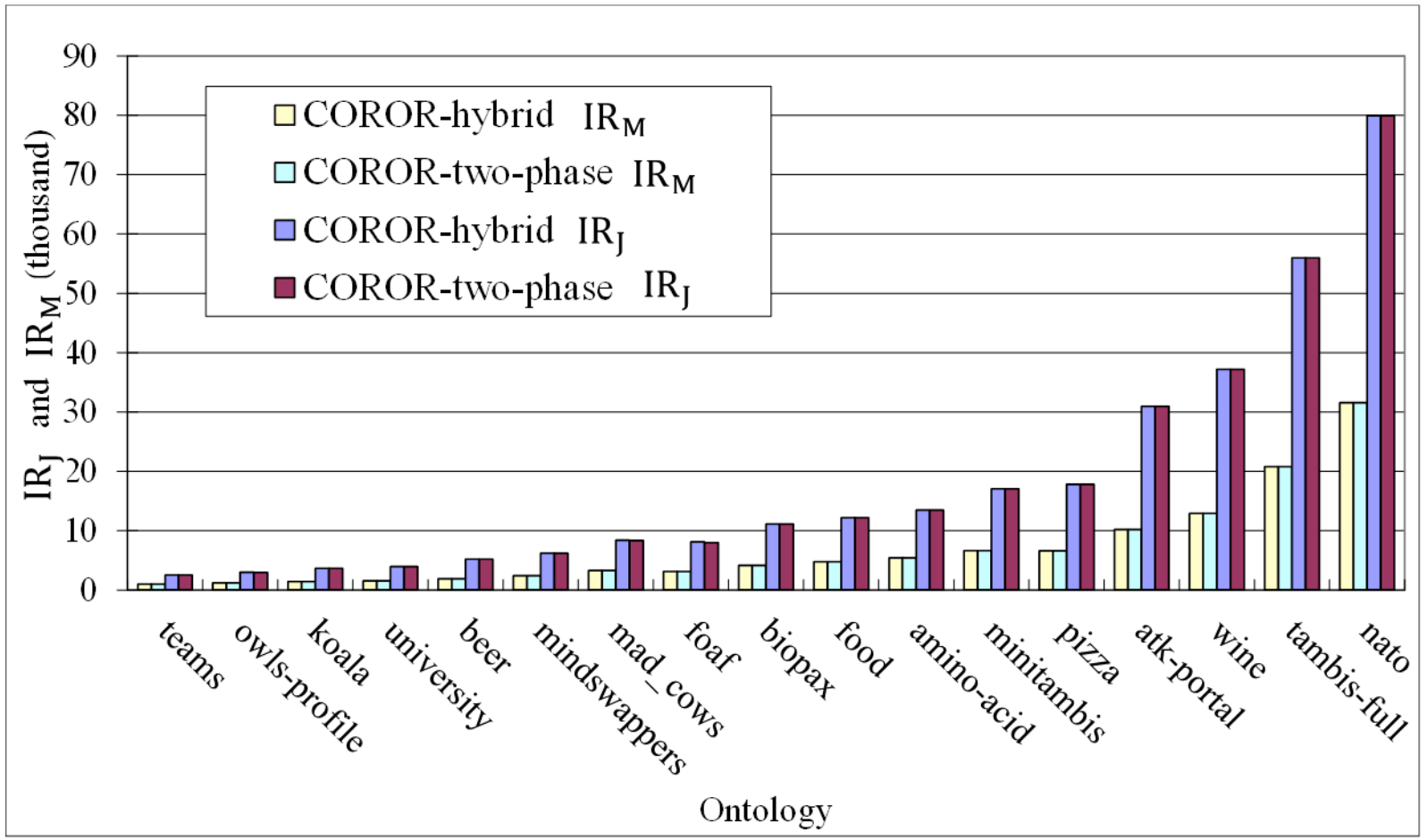
Investigation of $\#IR_M/\#IR_J$ shows that COROR-hybrid uses almost the same $\#IR_M/\#IR_J$ as COROR-two-phase (Figure 6-17). This can be explained as follows. If a rule is not selected by COROR-hybrid, then there must be a condition in it that has at least one OWL construct not included in the ontology (according to the principles of the *selective rule loading* algorithm) and so will have no matches at the initial matching stage when the number of matched facts is collected for each condition. Hence this condition is the most specific condition in the join sequence as it has no matched facts and is then re-ordered to the start of the join sequence by the *most specific-condition first* heuristic used in the *two-phase RETE* algorithm. This then leads to no join operations being performed for this join sequence and no intermediate results being generated in the corresponding beta network of the rule.

Still some intermediate results are generated in the alpha network because of successful matches in alpha nodes. However as the alpha network is highly shared in the *two-phase RETE* algorithm and for this rule set almost all of the unnecessary alpha network matches for the unnecessary rules are actually shared with other (necessary) rules, hence the unloading of a rule will not reduce much of the $\#IR_M$ generated in COROR hybrid. For example, in COROR hybrid the rule rdfp1 is unselected for the university ontology as the condition (?p rdf:type owl:FunctionalProperty) contains owl:FunctionalProperty which does not appear in the university ontology. The other conditions, e.g. (?u ?p ?v) in rdfp1, are shared by other (selected) rules, e.g. rdfs7x, and therefore no matter if the rule rdfp1 is unselected, the (shared) node (?u ?p ?v) is still constructed anyway (for the rule rdfs7x), and therefore unselecting rdfp1 does not cause the reduction of memory allocated

to the shared conditions. The small amount of memory reduction of COROR hybrid comes from both the removal of some unshared conditions in unselected rules and the removal of the data structures for unselected rules from the beta network.

The #M reduces for COROR-hybrid by a certain amount as shown in Figure 6-18 as some unshared conditions of unselected rules are missing from the alpha network, e.g. (?p rdf:type owl:FunctionalProperty) in rdfp1, leading to fewer match operations being required (although no successful matches for this condition since the missing of the OWL construct owl:FunctionalProperty in the ontology, still match operations are performed). The #J is the same for both COROR-hybrid and COROR-two-phase (Figure 6-19). Similarly, this is because the most specific conditions for unnecessary rules are those with no matched facts, and hence the *most-specific condition first* heuristic will reorder them to the start of the join sequences causing no join operation to be needed, as if they are "unselected".

The decrease in the #M reduces the reasoning time of COROR-hybrid by a small amount. However the introduction of the *selective rule loading* algorithm introduces extra time in computing the selective rule set, which in total may use more time than COROR two-phase. From these results, for this rule set it can be seen that the combination of the alpha node sharing heuristic and join sequence reordering heuristics in the *two-phase RETE* algorithm actually achieved much of the same benefit as the *selective rule loading* algorithm: in fact rather than optimizing for unnecessary rules, these heuristics can also optimize the alpha network and join sequences for selected rules, leading to even less time and memory requirements over that of COROR-selective. Hence COROR-hybrid has little benefit beyond that of COROR-two-phase. When combined with the restriction that the adoption of the *selective rule loading* algorithm limited the flexibility to alter rules (new construct-rule mapping entries are required) or to handle updates with new constructs (new constructs will not be handled since the rule is not in the selected rule set), it could be seen that the *selective rule loading* algorithm has limited utility. However, with a different rule set (especially domain specific rules) with less overlaps of conditions in the rules, the benefit of the *selective rule loading* algorithm would become much more pronounced with respect to alpha network memory and time savings.


90
80
70
60
50
40
30
20
10
0
IRJ and IRM (thousand)
COROR-hybrid IRM
COROR-two-phase IRM
COROR-hybrid IRJ
COROR-two-phase IRJ
teams
owls-profile
koala
university
beer
mindswappers
mad_cows
foaf
biopax
food
amino-acid
minitambis
pizza
atk-portal
wine
tambis-full
nato
Ontology

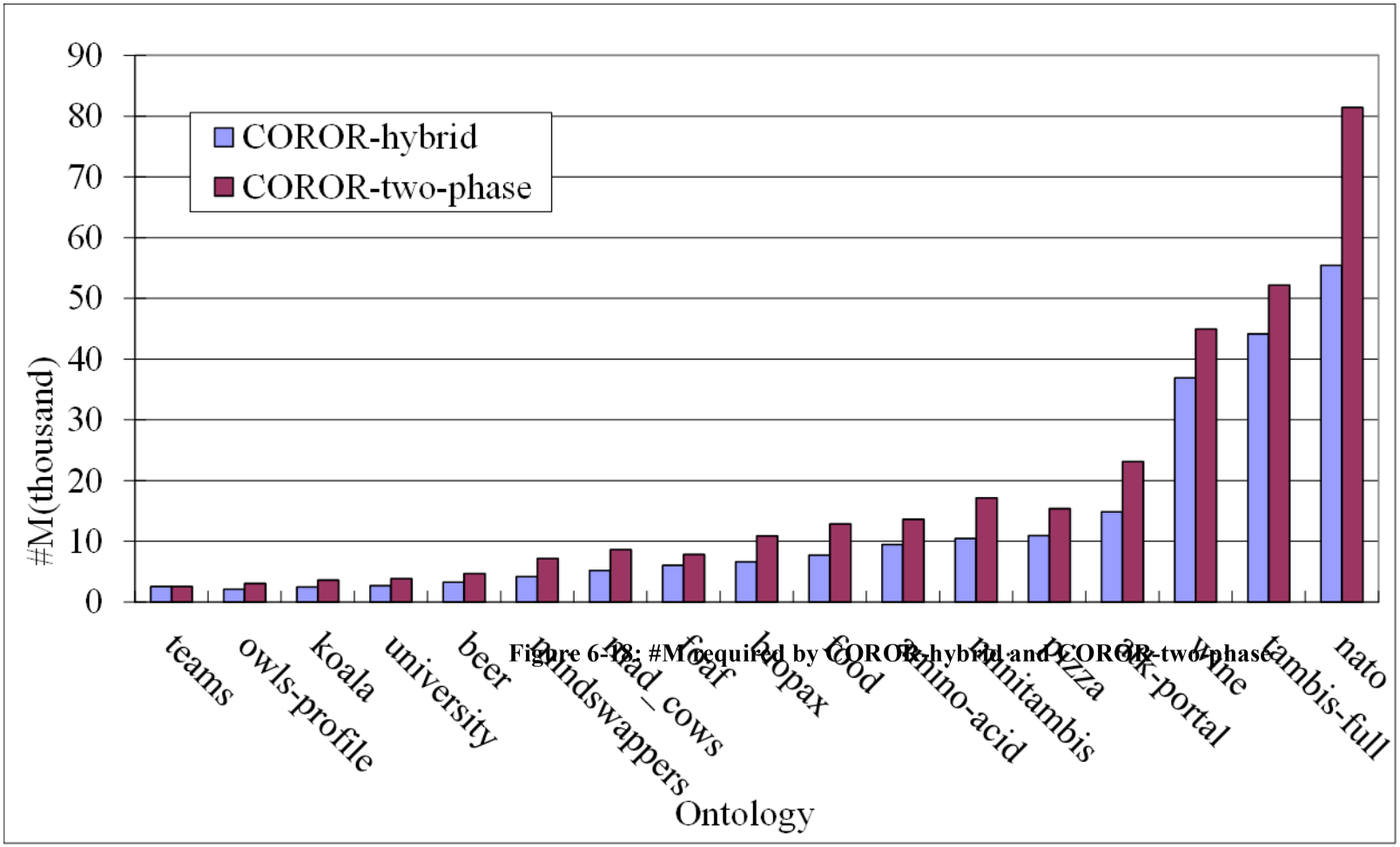


Figure 6-18: #M required by COROR-hybrid and COROR-two-phase

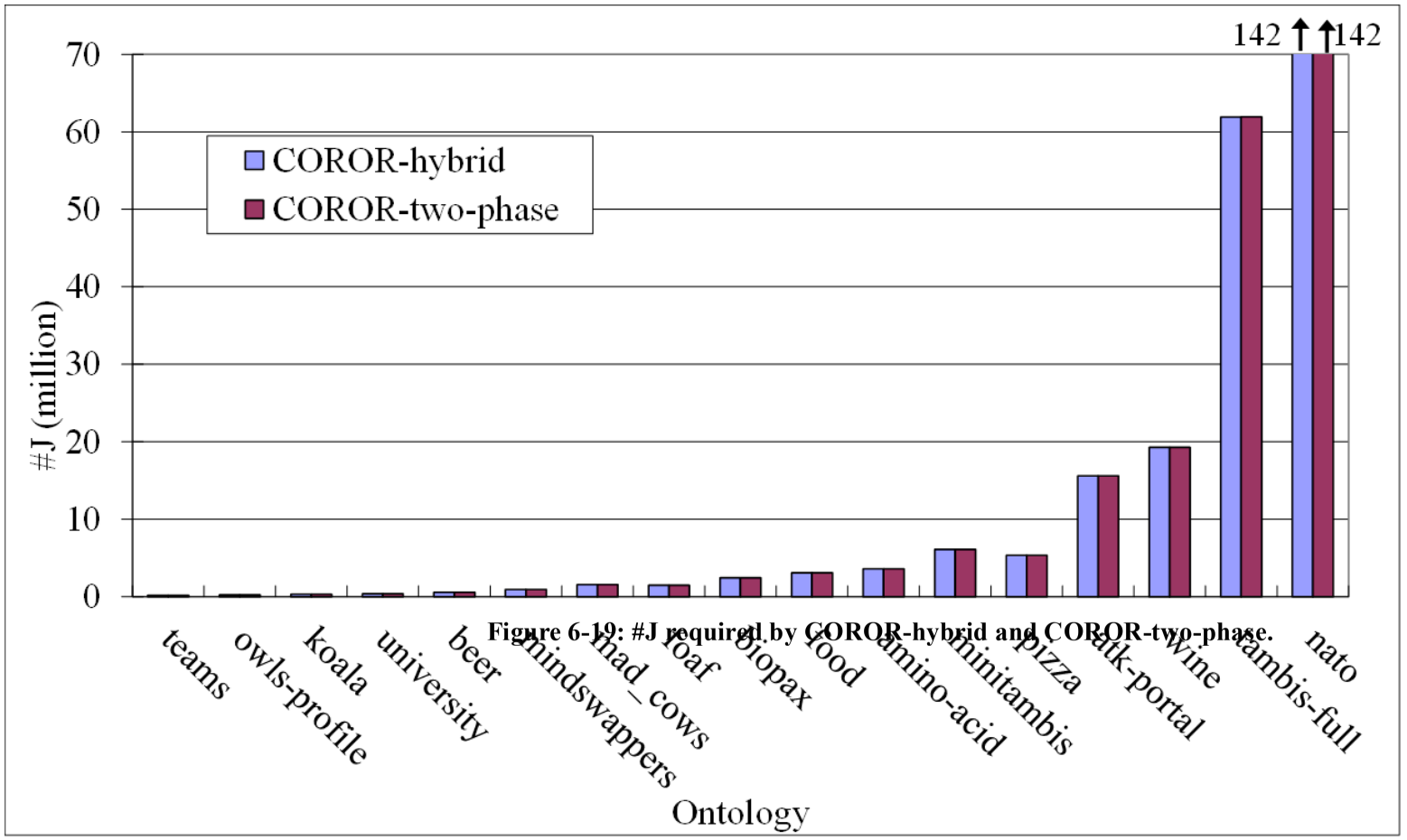


Figure 6-19: #J required by COROR-hybrid and COROR-two-phase.

6.2.4 Inter-Reasoner Comparison: Results and Discussions

The comparisons of reasoning time and memory usage between COROR and other state of the art reasoners are given in Figure 6-20a and Figure 6-20b. This experiment was performed on desktop using a J2SE platform. Experiment settings can be found in section 6.2.2. A similar measurement approach is used as described in section 6.2.2. The exact same COROR implementation as the one used in the intra-reasoner comparison is used. However rather than running on Sun SPOT, it runs on J2SE platform. Four other rule-entailment reasoners with similar reasoning algorithms and expressivity are also included in this experiment for comparison, including Jena, BaseVISor, swiftOWLIM, and Bossam. Since Jena also has a resolution engine, it was configured to use only forward chaining RETE engine and the same rule set as the one used in COROR was loaded. For brevity the Jena under this configuration is termed as Jena-forward. As discussed in the reasoner categorization as described in the related work in Chapter 2, SwiftOWLIM and BaseVISor are two desktop rule-entailment reasoners supporting similar semantics as the pD* entailment as well. However, as well as pD* entailment rules, the axiomatic triples and consistency rules are also supported by them. The expressivity of Bossam is not mentioned in the website[16], paper [Jang and Sohn 2004], and implementation, therefore made it difficult to judge its inference capability. COROR-hybrid was used in this comparison because it combines both composition algorithms and therefore can represent the performance gain achieved by both composition algorithms. Note that Pellet is also included in this experiment. It is a tableaux-based reasoner and performs complete OWL-DL reasoning. However it is included in this experiment not for comparing its performance side-by-side with the other rule-entailment reasoners (which are not complete OWL-DL reasoners). It is only included to give readers an intuition as to how COROR performs compared to a full-fledged complete OWL-DL reasoner.

The reasoning time required by the above reasoners to reason over a set of 17 ontologies (refer to Table 6-1 for more on the ontology) is shown in Figure 6-20a. As shown in the diagram, the time performance of COROR-hybrid is comparable to Jena-forward and BaseVISor. In contrast to the results listed in Figure 6-1b where COROR-hybrid uses much less reasoning time than COROR-noncomposable to reason a same ontology, here in the inter-reasoner comparison COROR-hybrid only slightly outperforms Jena-forward. Since (1) the exact same COROR implementation is used in both the intra-reasoner comparison and the inter-reasoner comparison, and (2) as discussed earlier in section 5.2.2, almost a same

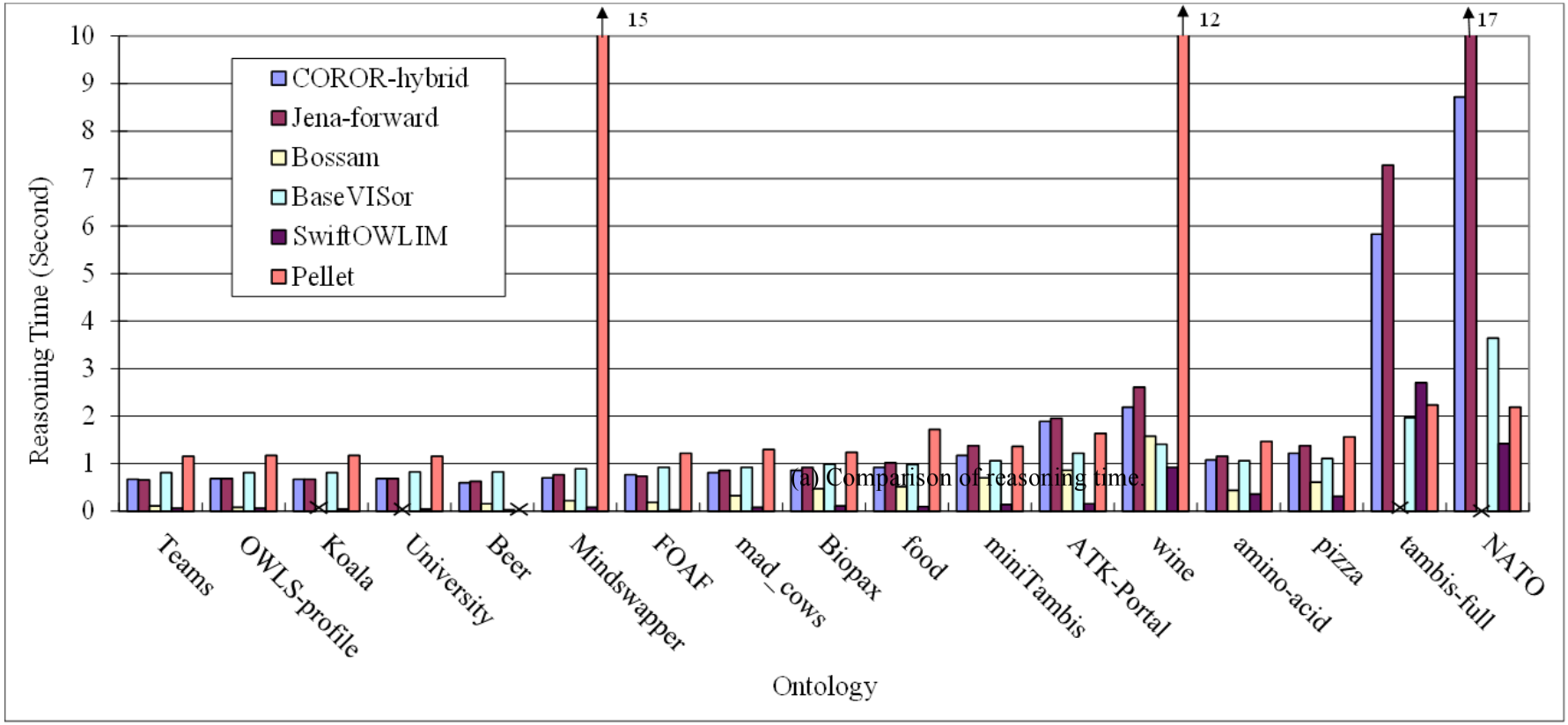
[16] http://bossam.wordpress.com/

RETE engine as the one used in Jena-forward is used in COROR-noncomposable except only for the use of J2ME CLDC conformant container classes in the one used in COROR-noncomposable in order to run on mobile devices, then a reasonable explanation for the different time reductions in Figure 6-1b and Figure 6-20a could be the performance differences of the different container classes used by COROR-noncomposable and Jena-forward. The List and Map used in COROR-noncomposable are self-defined by μJena, which could be relatively less optimized and slower compared to the standard Java container classes such as java.util.ArrayList and java.util.HashMap used in the Jena-forward, hence offsetting the time reduction gained by composing the reasoner.

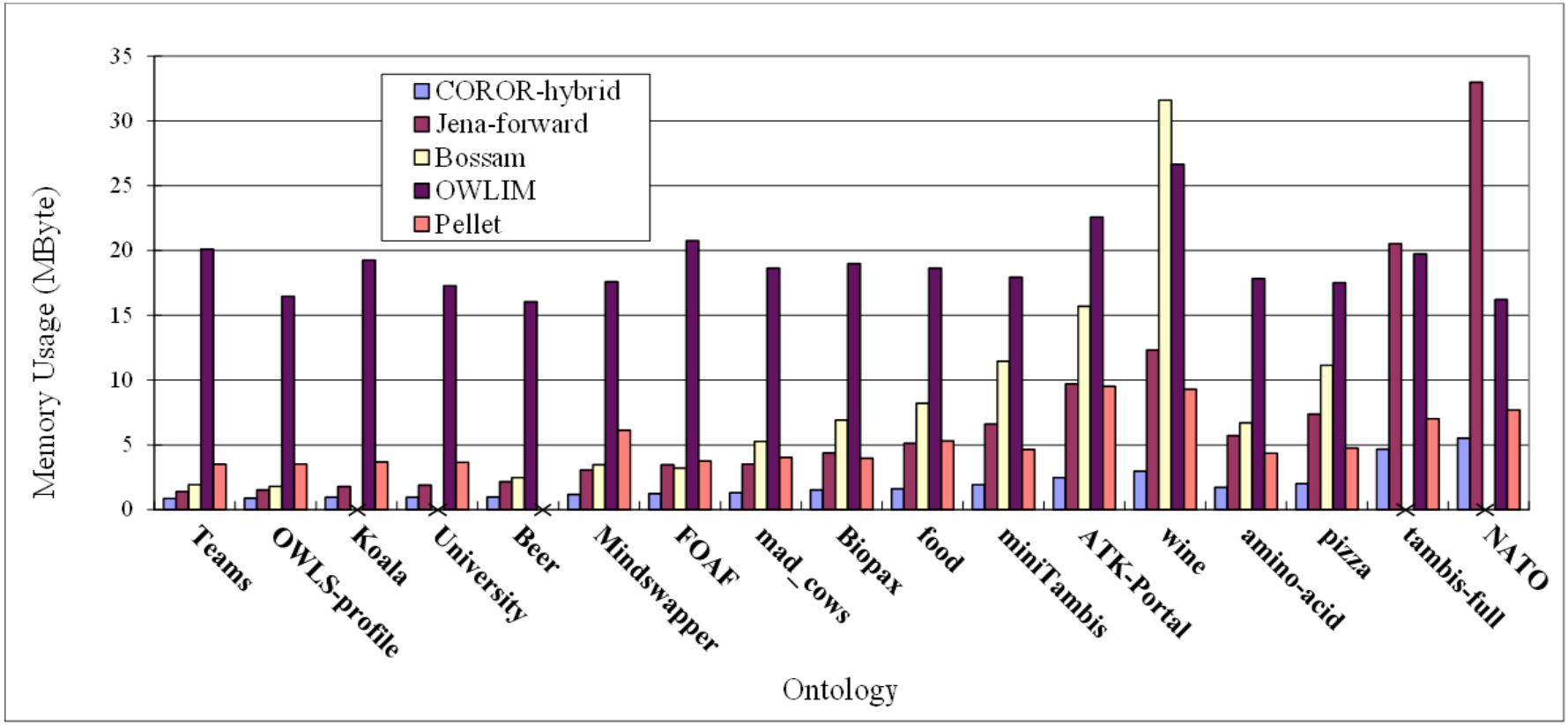
SwiftOWLIM is the fastest reasoner for most ontologies (except for Tambis-full where BaseVISor is the fastest). However as will be shown later in this section, it may trade memory for time. Although the time required by COROR-hybrid is less than that of Pellet for smaller ontologies (except for tambis-full and NATO which are the two largest in the selected ontology), Bossam is also fast for many ontologies, for some smaller ontology such as Teams, OWLS-profile, Beer, it can compete with swiftOWLIM. However it gives errors for four ontologies including Koala, University, tambis-full and NATO all of which are successfully reasoned by the other reasoners. Pellet generally has quite constant performance for most selected ontologies regardless of their sizes. However for smaller ontologies it uses more time than all the other rule-entailment reasoners. Pellet uses much more time to reason over wine and mindswapper. This is because the special structures used in the terminology and ABox definition which slows down the tableaux-based reasoning. Pellet does not have results for the Beer ontology as inconsistencies are detected.

As illustrated in Figure 6-20b, the memory performance of COROR is much better than the other reasoners. It uses the least memory for all selected ontologies. The memory usage grows much faster with the increase of both the size and the complexity of the ontology for Bossam and Jena-forward than COROR-hybrid does. For example, for the ATK-portal ontology Bossam uses 6 times more memory than COROR and for the Wine ontology it uses 13 times more memory than COROR. The memory footprint for swiftOWLIM is a lot larger than the COROR even for the very small ontology, e.g. even for teams it uses 20MB memory which is 15 times larger than COROR, which shows swiftOWLIM trades memory for time. The memory results indicate that much smaller memory footprint is required by COROR, and hence enable it to be better fit into resource-constrained devices. Note that BaseVISor hides its reasoning process from external inspection so it was not possible to

accurately measure its memory usage, and therefore it is omitted from the memory comparison.



(a) Comparison of reasoning time.


Memory Usage (MByte)
35
30
25
20
15
10
5
0
COROR-hybrid
Jena-forward
Bossam
OWLIM
Pellet
Teams
OWLS-profile
Koala
University
Beer
Mindswapper
FOAF
mad_cows
Biopax
food
miniTambis
ATK-Portal
wine
amino-acid
pizza
tambis-full
NATO
Ontology

6.2.5 Accuracy of the *Selective Rule Loading* Algorithm and the *Two-Phase RETE* Algorithm

In previous sections how the application of composition algorithms into a rule-entailment reasoner can affect its reasoning performance is studied. This section then discusses how the application of composition algorithms can affect the accuracy of reasoning. In theory, the exact same reasoning results are generated before and after the application of the *selective rule loading* algorithm and the *two-phase RETE* algorithm.

A direct and simple approach to demonstrate the reasoning accuracy of the composition algorithms is to compare reasoning results generated by different COROR composition modes to reason over the same ontologies side by side. Since as discussed in section 5.2.4 COROR-noncomposable uses a mobile version of the Jena RETE engine which has been widely used and tested in the OWL community, hence it is reasonable to compare its reasoning results with results generated by the other COROR modes (COROR-selective, COROR-two-phase and COROR-hybrid): if the same reasoning results are generated, it then can be concluded that the application of the composition algorithms does not change the reasoning accuracy.

An alternative approach may be comparing the reasoning results of COROR with other state of the art rule-entailment reasoners with the same semantics. However, this approach is not quite suitable to demonstrate the accuracy of the composition algorithms since the reasoning results rely largely on the rule set. Although some reasoners have their rule set also based on pD* semantics, the differences in the implementation of their rule sets, e.g. COROR does not use axiomatic triples but BaseVISor does; OWLIM condenses the triples generated for owl:sameAs, can cause different reasoning results to be obtained. Therefore their results cannot be compared side by side with COROR.

The comparison of reasoning results generated by the four COROR modes is performed in Ultra Edit[17] by first using the "Sort File" option to sort the result triples in an alphabetic order and then using the "Compare Files" option to perform comparison. Differences can be highlighted in the comparison window. Careful manual examination is then performed by the author, to find the differences between two results. Differences are all limited to the different names assigned to anonymous nodes. In fact these names are randomly assigned by the rule builtin assignAnon() as introduced in Table 5-2, but they still point to the same

[17] http://www.ultraedit.com/

resource. A summary of sizes of result ontology can be found in Table 6-4. No other differences are found in the compared results and therefore it is confident that the accuracy of the two designed reasoner composition algorithms is achieved.

Table 6-4: The size of result ontologies generated by each reasoner mode.

Ontology	Original Size	COROR-noncomposable	COROR-selective	COROR-two-phase	COROR-hybrid
teams	87	497	497	497	497
owls-profile	116	594	594	594	594
koala	147	710	710	710	710
university	169	760	760	760	760
beer	173	933	933	933	933
mindswapper	437	1350	1350	1350	1350
foaf	503	1772	1772	1772	1772
mad_cows	521	1695	1695	1695	1695
biopax	633	2228	2228	2228	2228
food	924	2437	2437	2437	2437
mini-tambis	1080	3407	3407	3407	3407
amino-acid	1465	2932	2932	2932	2932
atk-portal	1499	5418	5418	5418	5418
wine	1833	7043	7043	7043	7043
pizza	1867	3819	3819	3819	3819
tambis-full	3884	10959	10959	10959	10959
nato	5924	15746	15746	15746	15746

6.3 *Usability Test of TARS*

As already mentioned at the very beginning of this chapter, the major motivation for having an automatic reasoner selection process is to reduce the amount of efforts that could be involved in the future reasoner selection where interplay between semantic applications and reasoners could be extremely complicated due to the ever advancing of reasoner characteristics and application characteristics. RESP is then proposed, designed and implemented (as TARS) to enable application developers to select an appropriate reasoner independently using application characteristics. Although RESP automatically recommends the most appropriate reasoners for application developers, application developers are still involved in this process to identify application characteristics and to input the identified application characteristics. Furthermore once no appropriate reasoners are recommended they need to revise the identified application characteristics based on the results given by RESP, loosing or tightening the application characteristics and then entering another selection iteration. Reasoner experts are also involved in RESP to register candidate reasoners. Considering the major motivation of having RESP is to reduce the human-labour involved in the selection process, a usability test tends to be a best suitable evaluation for RESP. As a prototype complete implementation of RESP, TARS is then used in this evaluation.

Accuracy of TARS, i.e. if the most appropriate reasoner recommended by RESP is the most appropriate reasoner for the application in reality, is another important aspect to demonstrate. However it is not evaluated here since what is to be demonstrated in this thesis is that RESP as an abstract methodology does help application developers in selecting a reasoner, rather than the accuracy of the example application characteristics and connections derived in 4.3 and used in TARS. As a matter of fact, the accuracy of these example application characteristics and connections can be further refined when TARS comes into a practical stage.

Design, execution and results analysis of the usability test to TARS are presented in the reminder of this section.

6.3.1 Design of evaluation

Two tasks are designed to enable different participants to experience the distinct aspects of TARS. A first task is the *reasoner registration task* (task 1) that is designed to require reasoning-aware participants to register a reasoner with TARS using a reasoner description. It evaluates the usability of TARS reasoner registration interface as well as the difficulties involved in the identification of reasoner characteristics. A second task is the *reasoner registration task* (task 2) that is designed to require application-aware participants to use TARS to select a most appropriate reasoner for a semantic application following the RESP process. The aims of this task consist of three parts. Firstly it investigates the difficulties for participants in the identification of a complete and correct set of ACs. Secondly it requires participants to use TARS to perform reasoner selection following the RESP process, and feedback is collected on the usability of TARS.

In all 22 participants are selected and they are divided into two sub-groups according to the designed tasks: an *application-aware group* consisting of 17 application-aware participants with more experience in developing semantic applications and a *reasoning-aware group* consisting of 5 reasoning-aware participants with more experience in ontology reasoning. Six self-assessment questions designed by the author are used to enable participants to assess their own knowledge on semantic applications (Q2 to Q4) and on OWL reasoning (Q5 to Q7). These questions can be found in the post-task questionnaire in the attached DVD.

The question Q2 and Q3 assess the participant's level of knowledge on ontology and their frequency of using semantic applications. Each of the questions has 4 options describing different levels of knowledge (frequency): *expert* (*often*), *a lot* (*sometimes*), *some* (*seldom*),

and *none* (never). The question Q4 asks if the participant has ever designed a semantic application, and two options are provided for this question: *yes* or *no*. Similarly the question Q5 and Q6 correspondingly ask participants to assess their level of knowledge on ontology reasoning and the frequency of using an ontology reasoner. Q7 asks if the participant has ever looked into the internal reasoning mechanisms of an ontology reasoner.

In order to evaluate the overall knowledge on semantic applications or ontology reasoning from these assessment questions, a score system is designed and used in this thesis. For Q2, Q3, Q5 and Q6, 4 is assigned to *expert* (*often*), 3 to *a lot* (*sometimes*), 2 to *some* (*seldom*), and 1 to *none* (*never*). The question Q4 and question Q7 have two options yes and no. Since whether a participant has designed a semantic application or looked into the reasoning mechanisms of a reasoner can make big differences on his/her corresponding background knowledge, for Q5 and Q7, 4 is assigned to *yes* and 1 is assigned to *no*. Then the average score is calculated separately for Q2-Q4 and for Q5-Q7 in order to know the overall background knowledge of a participant on semantic application and on ontology reasoning. Adjectives are assigned to different average scores: if the average score is within [1, 2), then the participant is regarded as a *novice* user; If it is within [2, 3), then he/she is regarded as an *average* user; if it is within [3, 4] then he/she is regarded as an expert user.

The majority (82%) of application-aware participants have little or no knowledge on the intricacies of ontology reasoning (Figure 6-21). This fits into the vision motivating the development of RESP that semantic application developers would have little or no knowledge on semantic reasoning. The small size of the reasoning-aware group is reasonable as the reasoner registration task aims to collect expertise rather than large volumes of feedback. All the five reasoning-aware participants are expert reasoner users. The reasoner selection task is assigned to the application-aware group and the reasoner registration task to the reasoning-aware group.

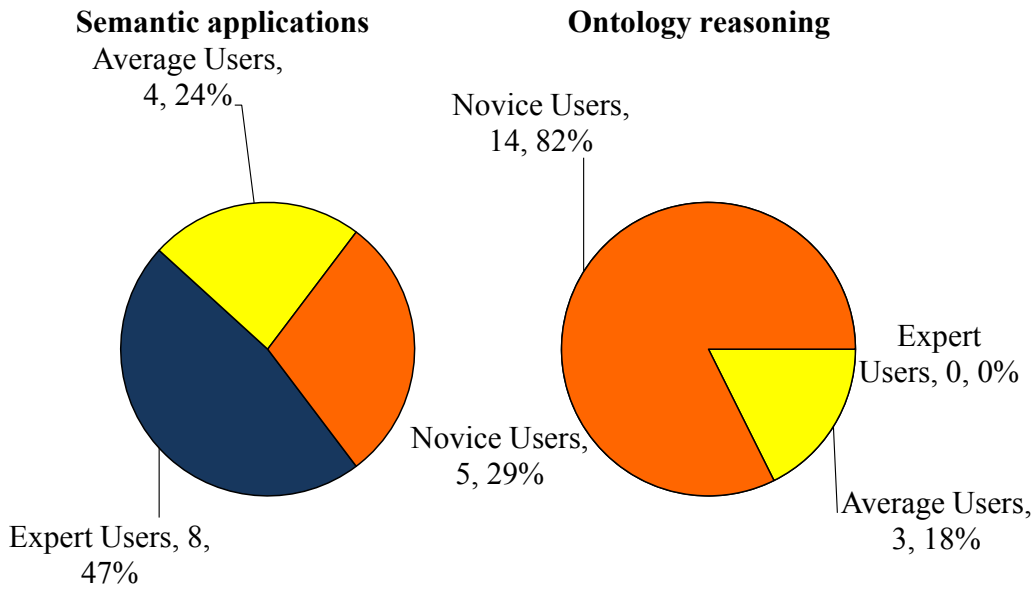


Figure 6-21: The level of background knowledge of application-aware participants on both semantic application and ontology reasoning (level of knowledge, number of participants, percentage).

Participants needed to carry out the corresponding task individually. Five steps needed to be followed for the reasoner selection task, including:

1. Participants are given a short introduction on the RESP process (and TARS interface). A step-by-step demonstration is given showing how to use TARS to perform reasoner selection.

2. A description on a semantic publish/subscribe system (can be found in Appendix B) for which a most appropriate reasoner is to be selected is given to participants and they are asked to individually analyse the description, identify the ACs, and input the identified ACs into TARS to start the reasoner selection using the RESP process.

3. Participant is asked to identify the most appropriate reasoner from the TARS results interface. Correct identification of all the ACs (as listed in Appendix B) for the given application description will lead to the selection of Pellet (which is with a 100% satisfaction rate). However over-identification or under-identification of ACs will lead to more than one appropriate reasoner to be identified (this may be the case for many other semantic applications but not for the given application) or none

to be identified.

4. (optional) If Pellet is not identified, participants are required to go into another RESP iteration revising the set of ACs previously identified (by losing some or tightening others), and invoke again the reasoner selection using TARS until Pellet is correctly identified.

5. Participants are required to complete two questionnaires: a post-task questionnaire for collecting feedback and comments on the functionality, usability, user interface and limitation of TARS, and a System Usability Scale (SUS) questionnaire as a general approach to evaluate the usability of TARS. The post-task questionnaire can be found in the attached DVD.

Two things need to be clarified. Firstly the application description was written with the collaboration of a reasoner expert (the author) and the author of the semantic publish/subscribe application, and to ensure the impartiality of the evaluation neither is involved in the evaluation. Secondly identified ACs for each participant are tracked by the system for each iteration for later analysis of the reason for incorrect identification.

Three steps need to be followed for the reasoner registration task, including:

1. Participants are given a short introduction on the RESP process (and TARS interface). A step-by-step demonstration is given showing how to use TARS to perform reasoner registration.

2. A description on BaseVISor is given to the reasoning-aware participants (can be found in Appendix B). They are asked to analyse the description and to identify the RCs for BaseVISor.

3. Reasoning-aware participants are required to register the reasoner into TARS.

4. A post-task questionnaire and a SUS questionnaire is given to each participant in order to collect their feedbacks on the usability of TARS (in terms of registering candidate reasoners).

Note that the description is a careful abstraction of the BaseVISor paper [Matheus et al 2006] with the collaboration by the same two reasoner experts as before. They were not involved in this experiment.

6.3.2 Results and discussions

All reasoning-aware participants successfully identified Pellet as the most appropriate reasoner in two iterations despite most having little or no knowledge on the specifics of ontology reasoning. Twelve out of the seventeen application-aware participants limited the reasoner selection results to only 2 candidate reasoners in the first iteration. With a little help by briefly explaining some application characteristics, all of them successfully selected Pellet as the most appropriate reasoners. The other five reasoning-aware participants selected the correct reasoner using only one iteration.

The history log of selection of ACs show that all the 12 participants who failed to select Pellet on the first iteration, failed to identify the AC *required expressivity SHION*. Hence KAON2 was also recommended as the other most appropriate reasoner in their first iteration for the given application since it fits well the other ACs of the application other than supporting less expressivity. Two other ACs of the given application were also neglected by some applications developers, including *complete reasoning* and *conjunctive query*. Five ACs were incorrectly selected, among which the AC *KB in database* was incorrectly selected by most of the application-aware participants (10 participants) while the AC *reasoning tasks* was incorrectly selected by four participants. The other three were only incorrectly selected by one participant.

Not selected AC	Count of participants	Causes
Ontology expressivity	12	Lack of knowledge.
Complete reasoning	2	Lack of knowledge.
Conjunctive queries	1	Lack of knowledge.

Table 6-5: Not selected ACs

Incorrectly selected AC	Count of participants	Causes
kb in database	10	Vague application description.
reasoning tasks	4	Vague application description, by mistake, self-inference, lack of knowledge.
user-defined datatypes	1	Self-inference.
closed-world queries	1	Lack of knowledge.
interface (remote)	1	Self-inference

Table 6-6: Incorrectly selected ACs.

A full list of the not identified and the incorrectly identified ACs as well as the count of participants for each is listed separately in Table 6-5 and Table 6-6. In addition, the reason for the incorrect identification of an AC is listed. The identified reasons are as a result of

discussions with each participant after the task. All the 12 participants not selecting the AC *ontology expressivity* regarded the lack of knowledge about the DL-style ontology expressivity (e.g. $\mathcal{SHOIN}(\mathcal{D})$) as the reason. This was also the reason for the not selection of the AC *complete reasoning* and the AC *conjunctive queries*, and the reason for the incorrect selection of the AC *reasoning tasks* and the AC *closed-world queries*. These problems can be avoided to a large extent in the future, by making the ACs more understandable, for example by using some explanation hints for each AC. This is partly shown by the fact that with a little explanation of the ACs to all the 12 application-aware participants that failed to identify Pellet in the first iteration, that they all successfully identified Pellet in the second iteration. All of the ten participants who incorrectly selected the AC *kb in database* regarded application description being vague as the reason: participants misunderstood the term "knowledge base" that is used in the application description as a type of database while in fact it represents the in-reasoner interpretation of the ontology. This reason can be partly handled using explanation hints for the AC as well. For example, an explanation can be supplied that KB does not have to be a database but can also be in-memory interpretation of ontology. Other reasons also exist. One of the reasons that cause the incorrect selection of the AC *reasoning tasks*, user-defined datatypes, and interface (remote) is self-inference which basically states that the AC is incorrectly selected based on participant's own understanding of the application although it is not explicitly stated in the application description. For example in the experiment since semantic publish/subscribe system was the application for which an appropriate reasoner was selected, a participant inferred that there must be some remote interface available for different brokers to communicate with each other. However, in fact in this application the inter-broker communication is not a part of the reasoner. However this problem is likely to be less of a problem in the real use of the tool, as in the experiment the participants were not the application developers of the system they were trying to characterise.

All reasoning-aware participants from the reasoning-aware group successfully registered BaseVISor to RESP, although with some not selected or incorrectly selected RCs. In a similar way, causes for not selection and incorrect selection were discussed with the participants after the task was complete. Not selected RCs and incorrectly selected RCs are respectively presented in Table 6-7 and Table 6-8 with the causes identified. Primarily two reasons were given for not selected or incorrectly selected RCs, the first being lack of knowledge and the second being vagueness in the reasoner description. For example, two reasoning-aware participants did not include the RC *reasoner type* in BaseVISor as they are

not familiar with the reasoner categorization used in this research (refer to section 2.3.1.1). This can be partly solved by giving detailed explanation hints for each RC. The RC *datatype support* was not selected by one participant, because it was overlooked. Reasoner expressivity (in OWL) was not selected by a participant because of the vagueness of the given reasoner description: RDFS constructs were not selected because they were not explicitly written in the reasoner description. As a matter of fact vague reasoner description was also the major reason for the incorrect selection of three other RCs, including *running platform* (J2ME platform was selected because BaseVISor was mentioned to be *embedded* into other applications, where the term embedded was interpreted by a reasoning-aware participant as can run on mobile devices), *reasoner expressivity* in DL (the original description does not explicitly state the given expressivity was given in OWL although OWL constructs are listed) and the *level of reasoner composition* (regarded procedural attachment in rule language as reasoner composition).

Not selected RC	Count of Participants	Causes
Reasoner type	2	Lack of knowledge
Native CWA support	2	Lack of knowledge and by mistake.
Query support	1	Lack of knowledge.
Datatype support	1	by mistake
Reasoner expressivity (in OWL)	1	Vague reasoner description.

Table 6-7: Not selected RCs.

Over-selected reasoner Characteristic	Count of Participants	Reasons
Running Platform	1	Vague reasoner description.
Reasoner expressivity (in DL)	1	Vague reasoner description.
Level of reasoner composition	1	Vague reasoner description, misunderstanding user-defined procedural attachment as composition of reasoning ability.
OS	1	Self-deduction

Table 6-8: Incorrectly selected RCs.

All identified reasons for not selection and incorrect selection point to the inappropriate naming of ACs and RCs (too technical terms in the names caused the lack of knowledge of some ACs and RCs) or the unclear nature of the given description. These problems however, can be solved without changing the TARS tool or the RESP process.

6.3.3 Questionnaires analysis

Two questionnaires, i.e. a post-task questionnaire and a SUS questionnaire, were used in this

experiment to collect feedback and comments on the functionality, usability, user interface and limitation of TARS. All 22 participants were required to fill the questionnaires after tasks. The following subsections correspondingly discuss feedback from the post-task questionnaire and the SUS questionnaire. The SUS questionnaire and the post-task questionnaire can be found in the attached DVD.

6.3.3.1 Post-task questionnaire

The post-task questionnaire consists of two parts with the first part containing 6 self-assessment questions on the background knowledge of the participant and the second part containing 15 evaluation questions on different facets of RESP. Only application-aware participants are required to answer self-assessment questions in the former part, and both application-aware and reasoning-aware participants are required to answer the evaluation questions in the second part. Some questions in the second part are task specific and therefore only participants taking the corresponding task need to answer them.

A scale ranging from *Strongly Disagree (1)* to *Strongly Agree (5)* is associated to evaluation question 8 to 20 and similarly for each level a numeric value is associated, as parenthesized. Question 21 and 22 collect comments and suggestions from participants respectively on the RESP reasoner selection process itself and its interfaces. Again mean values are calculated for questions and descriptive adjectives are associated to means: [1, 1.5) as *Strongly Disagree*, [1.5, 2.5) as *Disagree*, [2.5, 3.5) as *Neutral*, [3.5, 4.5) as *Agree* and [4.5, 5] as *Strongly Agree*. Mean value for each evaluation question can be found in Table 6-9.

Evaluation Question	Mean (overall)	Mean (app-aware)	Descriptive Adjective
8. I can understand the idea of reasoner characteristics.	4.0	4.0	Agree
9. I can understand the idea of application characteristics.	4.5	4.5	Agree
10. I can understand the RESP reasoner selection process.	4.4	4.5	Agree
11. (Task 2 only) I think the given set of application characteristics *precisely* capture the corresponding application characteristics of a realistic ontology-based application.	3.8	3.8	Agree
12. (Task 2 only) I think the given set of application characteristics *thoroughly* capture application characteristics of realistic ontology-based applications.	3.8	3.8	Agree
13. (Task 1 only) I think the given set of reasoner characteristics *precisely* capture the corresponding reasoner characteristics of a realistic ontology reasoner.	4.2	N/A	Agree
14. (Task 1 only) I think the given set of reasoner	3.8	N/A	Agree

characteristics *thoroughly* capture the corresponding reasoner characteristics of any realistic ontology reasoner.			
15. (Task 2 only) The manual approach to identify reasoner for an application, i.e. through discussion between reasoner experts and application developers, may have some biases due to reasons such as personal preference, miscommunication and so on.	3.9	3.9	Agree
16. (Task 2 only) I think the RESP reasoner selection process has the potential to avoid such bias mentioned above.	4.1	4.1	Agree
17. (Task 1 only) I find the reasoner registration interface is easy to use.	4.2	N/A	Agree
18. (Task 2 only) I find the reasoner selection interface is neat and easy to use.	4.3	4.3	Agree
19. (Task 2 only) I find the way reasoner selection results are presented is neat and easy for me to find out the appropriate reasoner(s) and the reason why the other reasoners are not appropriate.	4.6	4.6	Strongly Agree
20. (Task 2 only) I think the RESP reasoner selection process helps me in making a decision to select an appropriate reasoner for an application.	4.2	4.2	Agree

Table 6-9: Mean values (overall and application-aware) for evaluation questions

It can be concluded from Table 6-10 that most participants agreed that the concepts used in TARS, such as RC (question 8), AC (question 9) and the RESP reasoner selection process (question 10), were easily comprehensible. Some (7 out of 17) application-aware participants held neutral views on the precision (question 11) and completeness (question 12) of the set of example ACs, but still more than half of application-aware participants (10 out of 17) showed positive attitudes toward it. All five participants in the reasoning-aware group agreed that candidate RCs could precisely capture characteristics of realistic reasoners (question 14), however, two participants expressed neutral opinions on the completeness.

Question 15 and 16 focus on limitations of the conventional reasoners selection approach. Generally, participants agreed that the conventional approach, i.e. through discussion between reasoner experts and application developers, would bring about biases due to reasons such as personal preference, miscommunication and so on. It is also accepted by most (14 out of 17) participants that RESP is able to reduce or avoid such biases.

Feedback on TARS interfaces are collected in question 17, 18 and 19. It is generally accepted reasoner selection and reasoner registration interface are neat and easy to use (question 17 and 18). In addition the result interface is also considered as neat and clean for identifying the most appropriate reasoner. The reason that some ACs is not satisfied is also

considered easy to identify using the result interface according to feedback.

A mean of 4.4 in question 20, indicates that on average participants from the application-aware group regard the current RESP process as helpful for selecting the most appropriate reasoners for applications. In fact 16 out of 17 participants agree or strongly agree that RESP helped them in determining the most appropriate reasoner, while one held neutral opinions.

Comments and suggestions on RESP and TARS are collected in question 21. They concentrate on four aspects. Firstly application characteristics should be ordered and a guidance leading users through RESP would be more helpful for users to go through the selection process. Secondly predefined characteristic profiles could be constructed for some applications reducing effort required by application developers to identify application characteristics. Thirdly more complicated analysis could be introduced. For example users supply sample ontology, queries, and demanded results, and RESP analyses them and gives the most appropriate reasoner. Lastly it would be more helpful if result reasoners dynamically change in accordance with selected application characteristics, allowing users to know how the selection of a characteristic may impact on results. Comments and suggestions collected here can be taken as future work. Other comments about the TARS interface are gathered in question 22. They suggest using drop down lists rather than check boxes for some application characteristics, and using hover-over tips rather than question marks for annotations and using larger font.

6.3.3.2 SUS questionnaire

As a widely used tool for assessing system usability the SUS questionnaire was used in this experiment to survey the usability of RESP and TARS (the questionnaire used is given in the attached DVD). Both the reasoning-aware group and the application-aware group filled in this questionnaire. Scores are calculated for each questionnaire using the approach presented in [Brooke 1996] and they could range from 50 to 100. Since different tasks were carried out for different participant groups, their surveys are discussed separately. Table 6-11 and 6-12 separately presents the upper bound, lower bound, mean and standard deviation of each question for the application-aware group and the reasoning-aware group.

Table 6-11: Lower bound, upper bound, and mean score (in position) by questions for application-aware group.

Question	Lower Bound	Upper Bound	Mean	Stdev
1	2	5	3.8	0.8
2	1	3	1.6	0.8

3	3	5	4.5	0.6
4	1	4	2.1	1.1
5	3	5	4.1	0.7
6	1	3	1.5	0.6
7	3	5	4.3	0.7
8	1	3	1.5	0.7
9	2	5	4	0.9
10	1	4	2.4	1.1

Table 6-12: Lower bound, upper bound and mean score (in position) by questions for reasoning-aware group

Question	Lower Bound	Upper Bound	Mean	Stdev
1	2	5	3.6	1.1
2	1	2	1.6	0.5
3	4	5	4.6	0.5
4	1	3	1.8	0.8
5	3	5	4.4	0.9
6	1	2	1.6	0.5
7	1	5	3.4	1.5
8	1	4	1.6	1.3
9	4	5	4.4	0.5
10	1	3	1.6	0.9

A similar set of descriptive adjectives as that used above in Table 6-10 is used here to interpret the mean for each question. Mean values given in Table 6-11 and 6-12 show that feedback for most questions are positive. 12/17 participants agree that he/she would like to use this system frequently.

Table 6-13 gives the quartile breakdown of scores of surveys over the application aware group. Due to the small size of the reasoning-aware group (5 participants) quartile breakdown for it is not applicable.

Table 6-13: Lower bound, upper bound, and mean for each quartile of surveys over the application-aware group.

Quartile	Lower Bound	Upper Bound	Mean
1	50	67.5	58.8
2	70	80	75.6
3	82.5	87.5	84.4
4	90	100	93.5
Overall	50	100	79.0

The mean score is 79 for application-aware group and 80.5 for reasoning aware group. According to [Brooke 1996] they indicate that the usability of the reasoner selection and

reasoner registration of TARS is between *good* (71.4) and *excellent* (85.5).

6.4 *Summary and Key Findings*

This section presents the summary of the evaluation and key findings observed throughout the investigation of the research problem. They are presented from two perspectives: the reasoner composition algorithms (section 6.4.1) and RESP (section 6.4.2).

6.4.1 Reasoner Composition Algorithms

As discussed early in the introduction chapter, to reduce the resource consumption of OWL reasoning so that OWL reasoning can be deployed in resource-constrained environments, is the major motivation of having the reasoner composition approaches. Hence the evaluation of COROR concentrates on the performance changes before and after the application of the two developed reasoner composition algorithms. Results in Figure 6-1 show that the application of composition algorithms greatly reduces the memory consumption and reasoning time for COROR to fully compute pD* entailments for ontologies. An investigation into the inside of these two composition algorithms shows that both algorithms can always automatically compose a customized selective rule set or a customized RETE network for the particular ontology to be reasoned, hence leading to a large amount of memory/time reductions for all tested ontologies, which gives a general answer to the research question.

The evaluation also shows that composition at different levels has different capabilities in reducing resource consumptions. The *selective rule loading* algorithm composes at the rule level by removing unnecessary rules and keeping only the required rules for the ontology to be reasoned. Therefore the amount of memory that is originally allocated to unnecessary rules in a noncomposable reasoner is saved. However the loaded rules are still the same as they are authored which sometimes maybe quite inefficient (imagine that these rules are authored by domain experts who have little knowledge on rule optimization), and the RETE network is also unoptimized.

The *two-phase RETE* algorithm however performs composition inside the RETE algorithm. It composes customized RETE network for the ontology to be reasoned by applying two state of the art join sequence optimization heuristics and an alpha network optimization heuristic. However their applications are automatically taking the characteristics of the ontology to be reasoned into consideration. Unlike the work in [Ishida 1994] that uses an extra pre-execution for gathering the characteristics of the fact base (which may not be practical in resource-constrained devices as the limited resources), the *two-phase RETE*

algorithm uses an interrupted RETE construction mechanism that integrates the gathering of ontology characteristics in the first RETE cycles. As discussed in section 6.2.3.2, although all rules are loaded into the RETE engine, the *two-phase RETE* algorithm can optimize unnecessary rules as if they are "unloaded". Besides, join sequences and alpha networks of loaded rules are also optimized. Therefore in most cases the *two-phase RETE* algorithm can gain more memory/time reduction than the *selective rule loading* algorithm where loaded rules are not optimized. However given that the *two-phase RETE* algorithm gains memory/time reductions by optimizing rule join sequences, it is possible that for some special cases where rules are very large in number but small in individual size (i.e. rules contain less than two conditions) and there are small amount of shared conditions among rules, that the *selective rule loading* algorithm may have more performance reduction than the *two-phase RETE* algorithm.

An interesting finding from investigating the *selective rule loading* algorithm is its capability to predict the amount of memory reduction without performing reasoning (as discussed in the section 6.2.3.1). Since in the experiment the same rule appears to use the similar percentage of memory for all tested ontologies, the memory reduction (in percentage) for a different ontology then can be simply calculated by having the *selective rule loading* algorithm running through the ontology analysing the unnecessary rules (without reasoning) and then adding up the individual percentage for all the unnecessary rules. This finding can be useful to predict memory reduction for fast ontology prototyping when specific memory limitations are imposed. However since this finding is not a claim of this thesis it is not formally verified.

Two Designed Reasoner Composition Mechanisms vs. State of the Art

State of the art reasoner composition mechanisms are discussed in detail in section 2.3.3 and by analysing these mechanisms some aspects are derived for the reasoner composition research in this thesis to follow. Here how well the *selective rule loading* algorithm and the *two-phase RETE* algorithm satisfy these aspects is discussed.

The first aspect is the design of an automatic composition process. It is achieved. Both composition algorithms designed in this thesis are automatic composition algorithms.

The second aspect is that the approach would be free from a priori analysis. State of the art automatic reasoner composition mechanisms require some types of a priori analysis, e.g. to construct rule patterns, to group rules, or to assign weights to dynamic rules, which requires

manual analysis and also limits the application of the composition mechanisms to only one semantics or rule set. At the moment the *selective rule loading* algorithm relies on a priori manual analysis for the rule-construct dependencies. Hence it cannot be dynamically applied to a different semantic once the rule-construct dependencies are fixed. However the *two-phase RETE* algorithm composes inside the RETE algorithm and requires no such a priori analysis and is independent of specific rule sets. Hence unlike all state of the art automatic composition mechanisms, the *two-phase RETE* algorithm has the flexibility to be applied onto a different rule set at runtime.

The third aspect is the ability to compose for both ABox rules and TBox rules. The state of the art approaches, including the *dynamic rule generation* approach and the *incremental loading of rules/triples* (ILR/ILT) approach, works only on ABox rules. TBox rules are still processed in an uncomposed way, which may lead to resource waste for TBox reasoning. However, both composition algorithms have been designed in this thesis to work for both ABox rules and TBox rules.

The fourth aspect is to compose inside the reasoning algorithm, which is achieved by the *two-phase RETE* algorithm.

However compared to state of the art work, there are two limitations for the composition algorithms designed in this thesis. One limitation would be the size of an individual rule. Dynamic rules generated by the dynamic rule generation approaches are often very small and simple. This can generate smaller individual rules with much shorter join sequences, leading to less joins to be performed and less intermediate results to be generated and cached. Although the composition algorithms designed in this thesis can reduce the number of rules or can optimize the join sequence leading to a better join network, however the size of an individual rule is still unchanged and therefore the join sequences (although are optimized) are still long. Some interesting potential future work would be to combine the dynamic rule generation approach and the *two-phase RETE* algorithm since they compose at different levels. A second limitation is that the composition algorithms designed in this thesis lack the ability to compose at the ontology level as in ITL. Although the partition sizes used in ITL are too large for resource-constrained environments as targeted by this thesis, an incremental ontology loading approach will still be helpful for reducing the reasoning time and memory when applied in a desktop reasoner.

6.4.2 RESP

To reduce the amount of effort that could be involved in the future reasoner selection is the

major motivation for having RESP and therefore it is natural to have a usability evaluation for the implementation of RESP, that is TARS. Results reveal that the naming of application characteristics is a big issue that causes application-aware participants fail to identify the correct set of application characteristics for their application. An inappropriate name for an application characteristic (e.g. using reasoning-related terms in the name) may leave application-aware participants unable to understand the application characteristic and then fail to identify it. However with brief explanations of these inappropriate names, all participants can correctly identify the set of application characteristics for their applications. Hence expressing application characteristics using appropriate names, e.g. using domain specific languages to naming application characteristics or giving detailed explanation hints to application characteristics, would be very important for application developers to effectively use TARS to select a most appropriate reasoner for their applications.

Questionnaires show that most application-aware participants agree that the RESP reasoner selection process is easy to understand and is easy to follow. Most than half of application-aware participants regard the example application characteristics used in TARS in general to precisely capture the characteristics of semantic applications. 14 out of 17 application-aware participants agree that the conventional consultation-based approach may introduce bias in selecting reasoners but RESP somewhat overcomes this drawback. 16 out of 17 application-aware participants agree or strongly agree that RESP helped them in determining the most appropriate reasoner and one held a neutral opinion. SUS questionnaires were also used in the experiment. An average score of 79 for application-aware group and 80.5 for reasoning aware group indicates that the usability of the reasoner selection and reasoner registration of TARS is between *good* (71.4) and *excellent* (85.5).

Some drawbacks and limitations are identified for TARS and RESP. For example, the connections between ACs and RCs still need to be identified by collaboration of reasoner experts and domain experts, but even though this is a one-off task and then the connections can be reused in the future reasoner selection for an application domain, it still requires large amount of discussion. Secondly a set of general applicable ACs for all application domains may not exist. The example ACs used in the experiments are to some extent domain-neutral and therefore generally applicable. However the drawbacks are obvious in that application developers think it is difficult to understand some of them. Therefore domain-specific ACs (written in domain-specific language) may be important for application developers to better use RESP to perform selection, which means different sets of application characteristics need to be identified for different application domains (or they should be mapped to some

generally applicable ACs). Thirdly this process is still at its early stage and therefore no specific technical specifications are yet provided. However adding those in to RESP will not change the process and on the other hand this provides even more flexibility to implementers to use their own preferred format or approach, e.g. some would prefer to use rule engine to perform matchmaking but some others may prefer to use hardcode. Finally the RESP process is only semi-automatic. For example application developers are still largely involved in the process to identify application characteristics and reasoner experts are involved in identifying reasoner characteristics and registering them into RESP. It would be useful to use some kind of benchmarking tools to automatically analyse and register reasoners or allow identified application characteristics to be automatically adjusted according to the characteristics of existing candidate reasoners. These drawbacks and limitations can be considered in future work.

In this chapter the evaluation is discussed, the next chapter concludes this thesis.

Chapter 7
Conclusions and Future Work

7.1 *Progress vs. Objectives*

This section lists research question and derived research objectives and discusses how well they have been achieved. This thesis investigates the research question as to:

"How an appropriate resource-constrained OWL reasoner can be automatically composed and be selected based on application characteristics."

In order to investigate this research question, five objectives were derived:

- ***Objective 1:*** survey the state of the art OWL reasoners, identifying Reasoner Characteristics (RCs) and categorizing them. Identify an appropriate type of reasoner upon which the reasoner composition research should be based. Survey semantic applications, identifying reasoning-related Application Characteristics (ACs).

- ***Objective 2***: design automatic reasoner composition mechanisms and implement them in a resource-constrained reasoner.

- ***Objective 3***: study the performance impact on the resource-constrained reasoner brought by the application of composition algorithm(s).

- ***Objective 4***: design and implement a reasoner selection process that enables an application developer to automatically select a most appropriate reasoner for their semantic application based on application characteristics.

- ***Objective 5***: evaluate the usability of the reasoner selection process designed in objective 4.

Two automatic composition mechanisms, i.e. a *selective rule loading* algorithm and a *two-phase RETE* algorithm, are designed in response to the first half of the research question as to "how a resource-constrained OWL reasoner can be composed based on application characteristics". An automatic reasoner selection process is designed in response to the second half of the research question as to "how a resource-constrained OWL reasoner can be selected based on application characteristics". The reminder of this section discusses in detail how each research objective is achieved in this thesis.

State of the Art Objective: Objective 1

In response to objective 1, two surveys, a survey of state of the art OWL reasoners and a survey of semantic applications, were performed separately in section 2.3.1 and 2.3.2. In these surveys, a categorization of state of the art OWL reasoners was constructed, a set of reasoner characteristics was distilled and interplay between semantic applications and reasoners was studied.

Five reasoner categories are derived based upon their reasoning algorithms, which are DL-tableaux reasoners, rule-entailment reasoners, resolution-based reasoners, hybrid reasoners and miscellaneous reasoners. This categorization is unique and could be taken as a basis for future research that is based upon a type of reasoner.

As a part of objective 1 the reasoning composability (cf. section 2.3.3) are discussed for each of the above reasoner category. Both rule-entailment reasoners and resolution-based reasoners are regarded to have the best potential for composition. The suitability for rule-entailment reasoners and resolution-based reasoners to apply in resource-constrained environments are then discussed (cf. section 2.3.4). Rule-entailment reasoners are found to have more suitability than the others to run in resource-constrained environments. Hence rule-entailment reasoners are identified as the most suitable type of reasoners based on which the reasoner composition research is carried out. The examination and discussion of reasoner composability enables further composition research to be carried out for other reasoner types.

Furthermore a set of 18 reasoner characteristics were distilled for the survey. They provide a basis for the automatic reasoner selection research in this thesis. These reasoner characteristics can cover a variety of aspects of OWL reasoners ranging from reasoning algorithm to explanation. Their distillation and the survey based on these reasoner characteristics enable future research to be better carried out on these aspects.

The survey of semantic applications was performed over five types of semantic applications, which are the semantic publish/subscribe systems type, the semantic context-aware systems type, the clinical, medical and bioinformatics systems type, the semantic sensor network systems type, and the software engineering applications type. This survey investigated the requirements of particular applications/application types and the interplay between these requirements and the selected reasoner, facilitating the research of an automatic reasoner selection process. As a matter of fact the discussion of interplay in 4.3 from 11 reasoning-related aspects is based on this survey.

Design objectives: objective 2 and objective 4

Objective 2 and 4 are the design objectives. In response to objective 2 two automatic reasoner composition algorithms, i.e. the *selective rule loading* algorithm and the *two-phase RETE* algorithm, are designed and implemented in a prototype resource-constrained rule-entailment reasoner COROR. To achieve the objective 4, RESP is designed and implemented as a prototype desktop application TARS.

The design of the two composition algorithms (cf. section 3.4) followed the aspects pointed out in discussion of reasoner composability given in section 2.3.3. Hence they perform automatic reasoner composition which enables the reasoner to be automatically composed in a dynamic environment when different ontologies are used. The *two-phase RETE* algorithm composes inside the RETE algorithm and hence it is independent of the semantics or rule sets. Therefore unlike existing automatic composition algorithms such as the dynamic rule generation approach and the ILT/ILR approach (cf. section 2.3.3), no a priori manual analysis is required for the *two-phase RETE* algorithm in order to execute on a different semantics or rule set, which provide it with flexibility in a dynamic environment with changing semantics. Both designed composition algorithms execute for both ABox and TBox reasoning, which can further save resources in resource-constrained environments. Finally the *two-phase RETE* algorithm uses two state of the art join sequence optimization heuristics. However, it introduces a new way to apply them: rather than using an extra pre-execution of RETE to collect the required information about the ontology (as in [Ishida 1994]), an *interrupted RETE construction* approach is used that integrate the information collection into the execution of RETE, hence potentially reducing the resource cost.

In section 5.2, both composition algorithms were implemented in the *enhanced μJena* which is constructed by porting the Jena engine to a cut mobile Jena framework, *μJena*. The implemented composable reasoner is called COROR, the COmposable Rule-entailment Owl

Reasoner. It is the first automatically composable rule-entailment reasoner for resource-constrained environments. It enables OWL reasoning to be carried out in smaller resource-constrained devices such as sensors. Its design and implementation can provide some hints for similar research in the future.

The design of RESP is discussed in Chapter 4. RESP enables automatic reasoner selection to be independently performed by application developers to choose a most appropriate reasoner for their applications using only the application characteristics. Applications and reasoners are respectively abstracted in RESP as application characteristics and reasoner characteristics, and the selection is performed through matchmaking using pre-defined connections between application characteristics and reasoner characteristics. This approach is novel. It enables the materialization of reuse of the knowledge required in selecting a reasoner and hence reduces the effort required to put in for each reasoner selection. RESP itself is an abstract process without specifying any technical detail. However a set of example candidate application characteristics and their corresponding connections are derived from the discussions of interplay between applications and reasoners from 12 reasoning-related aspects. Although the derived application characteristics and connections are not mature enough for practical use, the way these reasoning-related aspects are examined shows a good example for the follow-on research of how application characteristics and connections can be derived.

RESP is implemented as a java desktop application TARS that allows application developers to perform reasoner selection independently following RESP and allows reasoner experts to register reasoners with TARS. TARS is the first tool performing automatic reasoner selection.

Evaluation objectives: objective 3 and objective 5

The objective 3 and 5 are evaluation objectives.

In response to objective 3, two performance experiments, an intra-reasoner comparison (section 6.2) and an inter-reasoner comparison (section 6.3), were conducted to respectively (1) study the performance impacts brought by the two designed reasoner composition algorithms to a rule-entailment reasoner, and (2) compare the performance of COROR to the state of the art rule-entailment reasoners. The intra-reasoner comparison compared the reasoning time and memory usage required by the four COROR composition modes, which are COROR-noncomposable, COROR-selective, COROR-two-phase and COROR-hybrid,

to fully compute entailments for the same ontology. The inter-reasoner comparison compared the time/memory usage required by COROR-hybrid with four state of the art rule-entailment reasoners, which are Jena forward, BaseVISor, swiftOWLIM, and Bossam. Results of the intra-reasoner comparison reveal that the application of the two designed composition algorithms can greatly reduce the time/memory required for performing rule-entailment reasoning. Results of the inter-reasoner comparison reveal that COROR-hybrid can use much less memory than the other reasoners without sacrificing reasoning time. The experiments and investigations performed thoroughly studied the reasoning performance impact by applying the designed composition algorithms.

To achieve the objective 5, a usability experiment was performed over TARS. In this evaluation, participants were grouped into two groups according to their background, i.e. an application-aware group and a reasoner-aware group. Each group was asked to perform a different task in order to experience a different facet of TARS. Application-aware participants were asked to select a most appropriate reasoner for the given application scenario following RESP. Reasoner-aware participants were asked to register candidate OWL reasoners with TARS. Results are positive. Most participants regarded TARS and RESP to be helpful for them to identify a most appropriate reasoner for semantic applications. An average SUS score of 79 for application-aware group and 80.5 for reasoning-aware group indicate TARS has usability between good and excellent. Results also revealed some limitations. A major one is the use of a too reasoning-related name for an application characteristic would affect the application-aware participants to identify the correct set of application characteristics. With some brief explanation to some application characteristics, application-aware participants then could successfully identify the correct set of application characteristics. This issue can be addressed using domain-specific languages to express application characteristics or using explanation hints. This usability experiment well evaluated the usability designed automatic reasoner selection process. In the meantime, this experiment helped the identification of limitations of the existing designed, enabling further improvement of the process and the implementation.

7.2 *Contributions*

Two contributions are identified. The major contribution is the design of two novel automatic reasoner composition algorithms for rule-entailment reasoners, termed the *selective rule loading algorithm* and the *two-phase RETE algorithm*, and the implementation of them in COROR (*COmposable Rule-entailment Owl Reasoner*). This contribution answers the first half of the research question as to "how a resource-constrained OWL

reasoner can be composed based on application characteristics". Evaluation results indicate that applying those two composition algorithms to a noncomposable rule-entailment reasoner can reduce a large amount of memory and time spent on OWL reasoning (the average reduced time/memory consumption for all the tested ontologies is 33%/35% for the *selective rule loading* algorithm and 78%/74% for the *two-phase RETE* algorithm, which can reduce the resource required of OWL reasoning so that OWL reasoning can run in resource-constrained environments.

This contribution enables OWL reasoning to be executed on smaller resource-constrained devices such as sensors or enables larger ontology to be reasoned on the same resource-constrained environment, which can further reduce the resource required for OWL-based intelligent data processing or management, hence enabling such intelligence to be better introduced into resource-constrained devices/applications, such as medical devices, in-vehicle network, wireless sensors/motes and so on.

This contribution and its related work were published in three papers. An initial design and a prototype implementation of the *selective rule loading* algorithm on a desktop reasoner are described in:

> W. Tai, J. Keeney and D. O'Sullivan, "An Automatically Composable OWL Reasoner for Resource Constrained Devices", in *Proceeding of the International Conference on Semantic Computing* (ICSC'09), Pages 495 -502, 2009.

In this paper, an initial OWL reasoner classification is also presented. A sketch of how a composable resource-constrained rule-entailment reasoner can be applied in a wireless sensor network management system to perform localized fault correlation (on sensor node) is described. This has been published in:

> R. Brennan, W. Tai, D. O'Sullivan, M. S. Aslam, S. Rea and D. Pesch, "Open Framework Middleware for intelligent WSN topology adaption in smart buildings", *Proceedings of the International Conference on Ultra Modern Telecommunications* (ICUMT'09), Pages 1-7, 2009.

A full description of both the *selective rule loading* algorithm and the *two-phase RETE* algorithm is described in:

> W. Tai, J. Keeney and D. O'Sullivan, "A COmposable Rule-Entailment Owl Reasoner for Resource-Constrained Devices", *Proceedings of the International*

Symposium on Rule-Based Reasoning, Programming, and Applications (RuleML'11), Pages 212 – 226, 2011.

The implementation of these two composition algorithms in COROR is included in this paper. Furthermore, the intra-reasoner comparison between different COROR composition modes and the inter-reasoner comparison between COROR and state of the art OWL reasoners are also described in this paper. In addition a final version of the reasoner classification is also included in this paper.

The minor contribution is the design and implementation of an automatic *REasoner Selection Process* (RESP) and a prototypical implementation, termed the *Tool for Automatic Reasoner Selection* (TARS). This contribution directly answers the second half of the research question as to "how a resource-constrained OWL reasoner can be selected based on application characteristics". Users can use TARS to perform automatic reasoner selection following RESP process, and also can use register new candidate reasoners to TARS. Usability evaluation shows TARS/RESP helps application developers in selecting a most appropriate reasoner for application with a little or even no help from reasoner experts (in contrast to the existing consultation-based reasoner selection process in which application developers totally rely on reasoner experts to select an appropriate reasoner).

This contribution provides a solution to the foreseen problem that future reasoner selection could get too complicated to be suitable for consultation-based reasoner selection process. It changes the existing human-to-human reasoner selection process to a semi-automatic human-to-computer process, which reduced the human effort required in the reasoner selection. Furthermore, the identified reasoner characteristics and their survey provide researchers a detailed overview of some state of the art reasoners from a variety of facets. In addition it provides a good starting point for people to have their own survey of reasoners. The examination of interplay between semantic applications and reasoners gives good hints for future research in order to study the selection of reasoners for semantic applications.

A paper was published on this contribution:

W. Tai, J. Keeney and D. O'Sullivan, "RESP: A Computer Aided OWL REasoner Selection Process", *Proceedings of the International Conference on Semantic Computing* (ICSC'11), 2011.

In this paper, the survey of semantic applications and the discussions on the 11 reasoning-

related aspects for candidate application characteristics and connections are presented. The design of RESP and the implementation of TARS are included as well. Furthermore, this paper also describes and discusses the usability evaluation carried out on TARS.

7.3 *Limitation and Future Work*

In previous chapters limitations for the automatic reasoner composition research and the automatic reasoner selection research have been clearly identified and discussed. In this section future work corresponding to these limitations are identified.

As discussed in section 3.4.2.2, using the number of matched facts collected in the initial matching for each condition as the specificity to reorder the join sequences is a direct approach but sometimes lacks accuracy. To (partly) solve this problem more information can be collected during or before the initial fact matching so that all collected information can together decide how the join sequences can be optimized. For example, the join selectivity factor between two joining conditions can be evaluated and used to determine the join sequence if these two conditions have the similar amount of matched facts. Other optimizations can also be introduced to improve the performance of RETE algorithm from other aspects. For example some indexing mechanisms can be combined to improve the searching performance.

Another piece of future work that needs to be undertaken in the reasoner composition research area is the extension of COROR to support OWL 2 RL. In fact the extension of OWL 2 has been discussed from the design perspective in section 3.5 and the rule-construct dependency graphs for OWL 2 RL entailment rules have been drawn. However, as discussed in 5.2.5 two obstacles need to be coped with in order to put this extension into practice: the lack of OWL 2 ontology manipulation API in μJena and the absence of a Jena compatible OWL 2 RL rule set. Early attempts to draft an OWL 2 RL rule set was impeded by the intensive and complex use of RDF list operations in OWL 2 RL semantics. A naïve solution to construct a built-in for each list operation, however, will require the construction of a large amount of complex built-ins, greatly complicating the rule set, and limiting the potential for node sharing capabilities and join sequence reordering.

Some other future research or development also exists. The first is to use other pattern matching algorithms such as TREAT and LEAPS (as introduced in section 2.4.2) to replace RETE and to study their composability. The second is to extend COROR to support conjunctive queries enabling complex queries to be placed and answered. Another possible extension to COROR is the DIG interface that enables distributed reasoning so that

distributed reasoning can be performed: simple reasoning tasks are processed locally in the sensor while complex reasoning tasks are sent to the server side where it is processed by a full-fledged reasoner, e.g. Pellet. This can achieve task balance among networked (resource-constrained) devices based on their processing capabilities.

In regard to RESP, one piece of research that can be considered is the further reduction of human effort required in identifying application characteristics for applications. This can be achieved through pre-defining application characteristic profiles for different application areas (as profiles for candidate reasoners) and hence application developers can modify existing profiles based on their applications, or the provision of step-by-step instructions to guide application developers to perform the identification of application characteristics. A second piece of future work that can be considered is the specification of technical detail for RESP so as to ease the development a RESP implementation, e.g. the format of application characteristics, reasoner characteristics, and connections, and the algorithm for matchmaking. Thirdly, the performance of reasoners could be taken as an application characteristic in the selection process. Some other future work from the implementation perspective includes (1) to implement TARS as a web-accessible tool and extend it to support OWL 2 reasoners, (2) to improve the interfaces of TARS (e.g. to enable the selection results changing dynamically with the changes in the selected ACs so that users can clearly know how the changes in ACs affect the selection results) and (3) to give more explanation hints to application characteristics.

7.4 *Final Remarks*

The continual development of OWL and OWL reasoning technologies has encouraged a lot of applications to adopt them in order to increase interoperability or to enable intelligent data processing. However during such semantic application development, a lot of problems were, are, and will be raised, such as applying OWL reasoning onto large database, applying OWL reasoning to a frequently changing knowledge base and so on. In this thesis two such problems are addressed and the usage of the proposed solutions will push forward the application of OWL reasoning to resource-constrained environments and also provides an alternative way to enable easier OWL reasoner selection. Solving such problems will hopefully encourage the adoption of OWL and OWL reasoning technologies by more applications, which, in turn, will enable the further development of OWL and OWL reasoning technologies.

References

[Acciarri et al 2005] A. Acciarri, D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, M. Palmieri, and R. Rosati, "QuOnto: Querying Ontologies", *Proceedings of the 20th National Conference on Artificial Intelligence (*AAAI'05*)*, 2005.

[Agostini et al 2005] A. Agostini, C. Bettini, and D. Riboni, "Loosely coupling ontological reasoning with an efficient middleware for context-awareness", *Proceedings of the 2nd Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services* (Mobiquitous'05), 2005.

[Ali 2010] S. Ali, "Semantic Interoperability of Ambient Intelligent Medical Devices and e-Health Systems", PhD Dissertation, Computer Science, University of Saarland, Saarbruecken, 2010.

[Ali and Kiefer 2009] S. Ali and S. Kiefer, "µOR - A Micro OWL DL Reasoner for Ambient Intelligent Devices", *Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing* (GPC'09), 2009.

[Allegrograph 2011] AllegroGraph RDFStore v4.2, Available at: http://www.franz.com/agraph/allegrograph/, Last visited: Oct. 2011.

[Antoniou et al 2005] G. Antoniou, C. V. Damásio, B. Grosof, I. Horrocks, M. Kifer, J. Małuszyński, and P. F. Patel-Schneider, "Combining Rules and Ontologies. A survey", *REWERSE Technical Report,* REWERSE-DEL-2005-I3-D3*,* 2005.

[Baader and Sattler 2001] F. Baader and U. Sattler, "An Overview of Tableau Algorithms

for Description Logics", *Studia Logica,* Volume 69, Issue 1, Pages 5-40, 2001.

[Baader et al 2005] F. Baader, S. Brandt, and C. Lutz, "Pushing the EL Envelope", *Proceedings of the 19th Internationnal Joint Conference on Artifical Intelligence (*IJCAI'05*)*, Pages 364-369, 2005.

[Baader et al 2006] F. Baader, C. Lutz, and B. Suntisrivaraporn, "CEL - A Polynomial-time Reasoner for Life Science Ontologies", *Proceedings of the 3rd International Joint Conference on Automated Reasoning (*IJCAR'06*)*, Pages 287-291, 2006.

[Baader et al 2007] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*, 2nd ed., Cambridge University Press New York, New York, 2007.

[Baader et al 2008] F. Baader, S. Brandt, and C. Lutz, "Pushing the EL Envelope Further", *Proceedings of the Washington DC workshop on OWL: Experiences and Directions (*OWLED'08DC*)*, 2008.

[Barbieri et al 2009a] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Continuous Queries and Real-time Analysis of Social Semantic Data with C-SPARQL", *Proceedings of Social Data on the Web (*SDoW'09*)*, 2009.

[Barbieri et al 2009b] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "C-SPARQL: SPARQL for continuous querying", *Proceedings of the 18th International Conference on World Wide Web (*WWW'09*)*, Pages 1061-1062, 2009.

[Barbieri et al 2010a] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Incremental Reasoning on Streams and Rich Background Knowledge", *Proceedings of the 7th Extended Semantic Web Conference (*ESWC'10*)*, Pages 1-15, 2010.

[Barbieri et al 2010b] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus, "An execution environment for C-SPARQL queries", *Proceedings of the 13th International Conference on Extending Database Technology (*EDBT'10*)*, 2010.

[Batory 1994] D. Batory, "The LEAPS Algorithms", *Technical Report,* 899216, Department of Computer Science, The University of Texas, Austin, 1994.

[Bechhofer et al 2005] S. Bechhofer, I. Horrocks, and D. Turi, “The OWL Instance Store: System Description”, *Proceedings of the 20th International Conference on Automated Deduction* (CADE’05), 2005.

[Berchtold et al 1998] S. Berchtold, C. Böhm, and H.-P. Kriegal, "The pyramid-technique: towards breaking the curse of dimensionality", *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (COMAD’98), 1998.

[Berners-Lee et al 2001] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web”, in *Scientific American Magazine*, May 2001.

[Berstel 2002] B. Berstel, "Extending the RETE algorithm for even management", *Proceedings of the 9th International Symposium on Temporal Representation and Reasoning (*TIME'02*)*, 2002.

[Bishop et al 2011] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov, “OWLIM: A Family of Scalable Semantic Repositories”, *Semantic Web Journal*, volum 2 issue 1, 2011.

[Bodriguez et al 2009] A. Bodriguez, R. McGrath, Y. Liu, and J. Myers, "Semantic Management of Streaming Data”, *the 2nd International Workshop on Semantic Sensor Networks* (SSN’09), 2009.

[Boehm et al 2008] S. Boehm, J. Koolwaajj, M. Luther, B. Souville, M. Wagner, and M. Wibbels, “Introducting YOUIT”, *Proceedings of the 7th International Semantic Web Conference (*ISWC’08*)*, Pages 804-817, 2008.

[Bolles et al 2008] A. Bolles, M. Grawunder, and J. Jacobi, "Streaming SPARQL - Extending SPARQL to Process Data Streams”, *Proceedings of European Semantic Web Conference (*ESWC'08*)*, Pages 448-462, 2008.

[Brennan et al 2009] R. Brennan, W. Tai, D. O'Sullivan, M. S. Aslam, S. Rea, and D. Pesch, "Open Framework Middleware for Intelligent WSN

Topology Adaption in Smart Buildings", *Proceedings of International Conference on Ultra Modern Telecommunications & Workshops*, Pages 1-7, 2009.

[Brooke 1996] J. Brooke, "SUS-A quick and dirty usability scale", *Usability evaluation in industry,* CRC Press, Pages 189-194, 1996.

[Brownston et al 1985] L. Brownston, R. Farrell, and E. Kant, "Programming Expert Systems in Ops5: An Introduction to Rule-Based Programming", Addison-Wesley, Boston, MA, 1985.

[Calder et al 2010] M. Calder, R. A. Morris, and F. Peri, "Machine reasoning about anomalous sensor data", *Journal of Ecological Informatics,* volume 5, issue 1, Pages 9-18, 2010.

[Calvanese et al 2007] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family", *Journal of Automated Reasoning,* volume 39, issue 3, Pages 385-429, 2007.

[Calvanese et al 2010] D. Calvanese, E. Kharlamov, W. Nutt, and D. Zheleznyakov, "Evolution of DL-Lite Knowledge Bases", *Proceedings of the International Semantic Web Conference* (ISWC'10), Pages 112-128, 2010.

[Calvanese et al 2011] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. Fabio Savo, "The MASTRO system for ontology-based data access", *Journal of Semantic Web: Interoperability, Usability and Applicability*, volume 2, issue 1, Pages 43-53, 2011.

[Carroll and De Roo 2004] J. J. Carroll and J. De Roo, "OWL web ontology language test cases", *W3C Recommendation*, Available at: http://www.w3.org/TR/owl-test/, 2004.

[Carroll et al 2004] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: implementing the semantic web recommendations", *Proceedings of the 13th international World Wide Web conference (*WWW'04*)*, Pages 74-83, 2004.

[Chen et al 2005] H. Chen, T. Finin, and A. Joshi, “The SOUPA ontology for pervasive computing”, *Ontologies for Agents: Theory and Experiences*, Springer-Verlag, Berlin, Pages 233-258, 2005.

[Choi et al 2008] C. Choi, I. Park, S. J. Hyun, D. Lee, and D. H. Sim, "MiRE: A Minimal Rule Engine for context-aware mobile devices", *Proceedings of the 3rd International Conference on Digital Information Management (ICDIM'08)*, Pages 172-177, 2008.

[CLIPS] CLIPS: A Tool for Building Expert Systems, Available at: http://clipsrules.sourceforge.net/, Last visited: Oct. 2011.

[Compton et al 2009a] M. Compton, C. Henson, L. Lefort, H. Neuhaus, and A. Sheth, "A Survey of the Semantic Specification of Sensors", *Proceedings of International Workshop on Semantic Sensor Networks* (SSN’09), 2009.

[Compton et al 2009b] M. Compton, H. Neuhaus, K. Taylor, K. -N. Tran, “Reasoning about Sensors and Compositions”, *Proceedings of International Workshop on Semantic Sensor Networks* (SSN’09), 2009.

[de Bruijn et al 2005] J. de Bruijn, A. Polleres, R. Lara, and D. Fensel, "OWL DL vs. OWL Flight: Conceptual Modeling and Reasoning for the Semantic Web", *Proceedings of the 14th International World Wide Web Conference (*WWW'05*)*, 2005.

[DIG] The new DIG interface standard (DIG 2.0), Available at: http://dl.kr.org/dig/interface.html, Last visited: Oct. 2011.

[Ding et al 2009] Y. Ding, Q. Wang, and J. Huang, "The Performance Optimization of CLIPS", *Proceedings of the 9th International Conference on Hybrid Intelligent Systems (*HIS'09*)*, 2009.

[Dobson et al 2006] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications", *ACM Transactions on Autonomous and Adaptive Systems (*TAAS*),* Volume 1, Issue 2, Pages 223-259, 2006.

[Donini et al 1992] F. M. Donini, M. Lenzerini, D. Nardi, A. Schaerf, W. Nutt, “Adding Epistemic Operators to Concept Languages”,

Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92), Pages 342-353, 1992.

[Donini et al 1998] F. M. Donini, M. Lenzerini, D. Nardi and A. Schaerf, "AL-log: Integrating Datalog and Description Logic", *Intelligent and Cooperative Information Systems*, Volume 10, Issue 3, Pages 227-252, 1998.

[Doyle 1977] J. Doyle, "Truth maintenance systems for problem solving", *Proceedings of the 5th International Joint Conference on Artificial Intelligence (*IJCAI'77*)*, 1977.

[Drabent et al 2007] W. Drabent and J. Henriksson and J. Maluszynski, "HD-Rules: a hybrid system interfacing Prolog with DL-reasoners", *Proceedings of the 2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services, co-located with the International Conference on Logic Programming* (ICLP'07), 2007.

[Drools 2010] Drools Business Logic Integration Platform, Available at: http://www.jboss.org/drools, Last visited: Oct. 2010.

[Drools Documentation V4.x] Drools Documentation for V4.x, Available at: http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html_single/index.html, Last visited: Mar. 2011.

[Eid et al 2007] M. Eid, R. Liscano, and A. El Saddik, "A Universal Ontology for Sensor Networks Data", *Proceedings of International Conference on Computational Intelligence for Measurement Systems and Applications* (CIMSA'07), Pages 59-62, 2007.

[Eiter et al 2005] T. Eiter, G. Ianni, R. Schindlauer and H. Tompits, "NLP-DL: A Knowledge-Representation System for Coupling Nonmonotonic Logic Programs with Description Logics", *Proceedings of the International Semantic Web Conference* (ISWC'05), 2005.

[Ejigu et al 2007] D. Ejigu, M. Scuturici, and L. Brunie, "An Ontology-Based Approach to Context Modeling and Reasoning in Pervasive Computing", *Proceedings of the 5th Annual IEEE International*

Conference on Pervasive Computing and Communications Workshops (PerCom Workshops '07), Pages 14-19, 2007.

[Elmasri and Navathe 2003] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 4th edition, Addison-Wesley, 2003.

[Fabret et al 1993] F. Fabret, M. Régnier, and E. Simon, "An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases", *Proceedings of the 19th International Conference on Very Large Data Bases (*VLDB'93*)*, 1993.

[Forgy 1982] C. Forgy, "Rete: A Fast Algorithm for the many pattern/many object pattern match problem", *Artificial Intelligence,* Volume 19, Issue 1, Pages 17-37, 1982.

[Golbeck et al 2003] J. Golbeck, F. Fragoso, F. Hartel, J. Hendler, J. Oberthaler and B. Parsia, "National Cancer Institute's Thesaurus and Ontology", *Journal of Web Semantics*, Volume 1, Issue 1, 2003.

[Gomez et al 2008] M. Gomez, A. Preece, M. P. Johnson, G. d. Mel, W. Vasconcelos, C. Gibson, A. Bar-Noy, K. Borowiecki, T. L. Porta, D. Pizzocaro, H. Rowaihy, G. Pearson, and T. Pham, "An Ontology-Centric Approach to Sensor-Mission Assignment", *Proceedings of 16th International Conference on Knowledge Engineering and Knowledge Management (*EKAW'08*)*, Pages 347-363, 2008.

[Grau et al 2007] B. C. Grau, C. Halaschek-Wiener, and Y. Kazakov, "History matters: Incremental ontology reasoning using modules", *Proceedings of the 6th International and the 2nd Asian Semantic Web Conference (*ISWC2007+ASWC2007*)*, Pages 183-196, 2007.

[Grau and Halaschek-Wiener 2010] B. C. Grau and C. Halaschek-Wiener, "Incremental Classification of Description Logics Ontologies", *Journal of Automated Reasoning (*JAR*),* Volume 44, Issue 4, Pages 337-369, 2010.

[Grosof et al 2003] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, "Description logic programs: combining logic programs with description logic", *Proceedings of the 12th international conference on*

World Wide Web (WWW'03), Pages 48 – 57, 2003.

[Gu et al 2005] T. Gu, H. K. Pung and D. Q. Zhang, "A service-oriented middleware for building context-aware services", *Journal of Network and Computer Applications*, Volume 28, Issue 1, Pages 1-18, 2005.

[Gu et al 2007] T. Gu, Z. Kwok, K. K. Koh, and H. K. Pung, "A Mobile Framework Supporting Ontology Processing and Reasoning", *Proceedings of Workshop on Requirements and Solutions for Pervasive Software Infrastructures*, 2007.

[Guha and Hayes 2003] R. V. Guha and P. Hayes, "Lbase: semantics for languages of the semantic web", *W3C Working Group Note*, Available at: http://www.w3.org/TR/lbase/, Oct. 2003.

[Guo et al 2005] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems", *Web Semantics: Science, Services and Agents on the World Wide Web,* Volume 3, Issue 2-3, Pages 158-182, 2005.

[Gupta et al 1993] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally", *Proceedings of ACM SIGMOD international conference on Management of data*, Pages 157-166, 1993.

[Haarslev and Möller 1999] V. Haarslev and R. Möller, "An Empirical Evaluation of Optimization Strategies for ABox Reasoning in Expressive Description Logics", *Proceedings of the 1999 International Workshop on Description Logics (*DL'99*)*, 1999.

[Haarslev and Möller 2002] V. Haarslev, and R. Möller, "Practical Reasoning in RACER with a Concrete Domain for Linear Inequations", *Proceedings of International Workshop on Description Logics* (DL'02), 2002.

[Haarslev and Möller 2003a] V. Haarslev and R. Möller, "Incremental Query Answering for Implementing Document Retrieval Services", *Proceedings of International Workshop on Description Logics* (DL'03), Pages 85-94, 2003.

[Haarslev and Möller 2003b] V. Haarslev, and R. Möller, "Description Logic Systems

with Concrete Domains: Applications for the Semantic Web", *Proceedings of International Workshop on Knowledge Representation meets Databases* (KRDB'03), 2003.

[Haarslev et al 2001] V. Haarslev, R. Möller, and A.-Y. Turhan, "Exploiting Pseudo Models for TBox and ABox Reasoning in Expressive Description Logics", *Proceedings of the 1st International Joint Conference on Automated Reasoning (*IJCAR'01*)*, Pages 61-75, 2001.

[Halaschek-Wiener et al 2006] C. Halaschek-Wiener, B. Parsia, and E. Sirin, "Description Logic Reasoning with Syntactic Updates", *Proceedings of the 5th Ontologies, Databases, and Applications of Semantics (ODBASE'06)*, Pages 722-737, 2006.

[Halaschek-Wiener 2007] C. Halaschek-Wiener, "Expressive syndication on the web using a description logic based approach", *Ph.D. Dissertation*, University of Maryland, College Park, MD, 2007.

[Halaschek-Wiener and Hendler 2007] C. Halaschek-Wiener and J. Hendler, "Toward expressive syndication on the web", *Proceedings of the 16th international conference on World Wide Web (*WWW'07*)*, Pages 727-736, 2007.

[Halaschek-Wiener and Kolovski 2008] C. Halaschek-Wiener and V. Kolovski, "Syndication on the Web using a description logic approach", *Journal of Web Semantics: Science, Services and Agents on the World Wide Web,* Volume 6, Issue 3, 2008.

[Hall et al 2004] L. Hall, A. Gordon, R. James, and L. Newall, "A Lightweight Rule-Based AI Engine for Mobile Games", *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology* (ACE'04), 2004.

[Hanson and Hasan 1993] E. N. Hanson and M. S. Hasan, "Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing", *Technical Report*, CIS Department, University of Florida, 1993.

[Hanson et al 2002] E. N. Hanson, S. Bodagala, and U. Chadaga, "Trigger Condition Testing and View Maintenance Using Optimized

Discrimination Networks", *IEEE Transactions on Knowledge and Data Engineering,* Volume 14, Issue 2, Pages 261-280, 2002.

[Harris et al 2004] M. A. Harris, J. Clark, A. Ireland, J. Lomax, M. Ashburner, R. Foulger, K. Eilbeck, S. Lewis, B. Marshall, and C. Mungall, "The Gene Ontology (GO) database and informatics resource", *Nucleic acids research*, Volume 32, Database issue, Pages D258 - D261, 2004.

[Hatzi et al 2009] O. Hatzi, G. Meditskos, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas, "PORSCE II: Using Planning for Semantic Web Service Composition", *Proceedings of the International Competition on Knowledge Engineering for Planning and Scheduling* (ICKEPS'09), in conjunction with *the International Conference on Automated Planning and Scheduling* (ICAPS'09), Pages 38-45, 2009.

[Henss et al 2009] J. Henss, J. Kleb, S. Grimm, and J. Bock, "A Database Backend for OWL", *Proceedings of the International Workshop on OWL: Experiences and Directions* (OWLED'09), 2009.

[Herzog et al 2008] A. Herzog, D. Jacobi, and A. Buchmann, "A3ME - An Agent-Based Middleware Approach for Mixed Mode Environments", *Proceedings of the Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM'08)*, Pages 191-196, 2008.

[Hogan et al 2009] A. Hogan, S. Decker and A. Polleres. "Scalable Authoritative OWL Reasoning for the Web", *International Journal on Semantic Web and Information Systems* (IJSWIS'09), Volume 5, Issue 2, 2009.

[Horridge and Bechhofer 2011] M. Horridge and S. Bechhofer, "The OWL API: A Java API for OWL ontologies", *Semantic Web Journal: Interoperability, Usability and Applicability*, Volume 2, Issue 1, Pages 11-21, 2011.

[Horrocks et al 2000] I. Horrocks, U. Sattler, and S. Tobies, "Practical Reasoning for Very Expressive Description Logics", *Logic Journal of the*

IGPL, Volume 8, Issue 3, Pages 239-263, 2000.

[Horrocks and Patel-Schneider 2004a] I. Horrocks and P. F. Patel-Schneider, "Reducing OWL entailment to description logic satisfiability", *Journal of Web Semantics,* Volume 1, Issue 4, Pages 345-357, 2004.

[Horrocks and Patel-Schneider 2004b] I. Horrocks and P. F. Patel-Schneider, "A Proposal for an OWL Rules Language", *Proceedings of the 20th International Conference on World Wide Web* (WWW'04), Pages 482-496, 2004.

[Horrocks et al 2004] I. Horrocks, L. Li, D. Turi, and S. Bechhofer, "The instance store: Description logic reasoning with large numbers of individuals", *Proceedings of the 2004 International Workshop on Description Logics (*DL'04*)*, Pages 31-40, 2004.

[Horrocks et al 2005] I. Horrocks, P. F. Patel-Schneider, and S. Bechhofer, "OWL rules: a proposal and prototype implementation", *Journal of Web Semantics*, Volume 3, Issue 1, Pages 23-40, 2005.

[Hustadt et al 2004a] U. Hustadt, B. Motik, and U. Sattler, "Reducing SHIQ-Description Logic to Disjunctive Datalog Programs", *Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (*KR'04*)*, Pages 152-162, 2004.

[Hustadt et al 2004b] U. Hustadt, B. Motik and U. Sattler, "Reasoning in Description Logics with a Concrete Domain in the Framework of Resolution", *Proceedings European Conference on Artificial Intelligence (*ECAI'10*)*, 2004.

[Hustadt et al 2005] U. Hustadt, B. Motik, and U. Sattler, "Data Complexity of Reasoning in Very Expressive Description Logics", *Proceedings of the 19th International Joint Conference on Artificial Intelligence (*IJCAI'05*)*, 2005.

[Ishida 1988] T. Ishida, "Optimizing Rules in Production System Programs", *Proceedings of the 7th National Conference on Artificial Intelligence (*AAAI'87*)*, Pages 699-704, 1988.

[Ishida 1994] T. Ishida, "An optimization algorithm for production systems", *IEEE Transactions on Knowledge and Data Engineering,*

Volume 6, Issue 4, Pages 549-558, 1994.

[Jang and Sohn 2004] M. Jang and J.-C. Sohn, "Bossam: An Extended Rule Engine for OWL Inferencing", *Proceedings of the 3rd International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML'04)*, Pages 128-138, 2004.

[Jarke and Koch 1984] M. Jarke and J. Koch, "Query Optimization in Database Systems", *ACM Journal of Computing Surveys,* Volume 16, Issue 2, Pages 111-152, 1984.

[Jena 2010] Jena – A Semantic Web Framework for Java, Available at: http://jena.sourceforge.net/, Last visited: Oct. 2011.

[Jena TDB] TDB - A SPARQL Database for Jena, Available at: http://openjena.org/TDB/, Last visited: Oct. 2011.

[JESS] JESS, the rule engine for Java platform, Available at: http://www.jessrules.com/jess/index.shtml, Last visited: Sep. 2011.

[Kalyanpur et al 2006] A. Kalyanpur, B. Parsia, and E. Sirin, "Debugging Unsatisfiable Classes in OWL Ontologies", *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, Volume3, issue 4, 2006.

[Kang and Cheng 2004] J. A. Kang and A. M. K. Cheng, "Shortening matching time in OPS5 production systems", *IEEE Transactions on Software Engineering,* Volume 30, Issue 7, Pages 448-457, 2004.

[Katz and Parsia 2005] Y. Katz, and B. Parsia, "Towards a Nonmonotonic Extension to OWL", *Proceedings of International Workshop on OWL: Experiences and Directions* (OWLED'05), 2005.

[Kaviani et al 2008] N. Kaviani, B. Mohabbati, D. Gasevic, M. Finke, "Semantic Annotations of Feature Models for Dynamic Product Configuration in Ubiquitous Environments", *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering* (SWESE'08), 2008.

[Keeney et al 2008] J. Keeney, D. Roblek, D. Jones, D. Lewis, D. O'Sullivan, "Extending Siena to support more expressive and flexible

subscriptions", *Proceedings of International Conference on Distributed Event-Based Systems* (DEBS'08), 2008.

[Keeney et al 2010] J. Keeney, C. Stevens, D. O'Sullivan, "Extending a Knowledge-based Network to support Temporal Event Reasoning", *Proceedings of the 12th IEEE/IFIP Network Operations & Management Symposium* (NOMS'10), Pages, 631-638, 2010.

[Keet et al 2007] C. M. Keet, M. Roos, and M. S. Marshall, "A Survey of Requirements for Automated Reasoning Services for Bio-Ontologies in OWL", *Proceedings* of *the 3rd OWL: Experiences and Directions Workshop (OWLED2007)*, 2007.

[Keivanloo et al 2010] I. Keivanloo, L. Roostapour, P. Schugerl, and J. Rilling, "SE-CodeSearch: A scalable Semantic Web-based Source Code Search Infrastructure", *Proceedings of 2010 IEEE International Conference on Software Maintenance (ICSM'10)*, Pages 1-5, 2010.

[Kim et al 2008] J.-H. Kim, H. Kwon, D. -H. Kim, H. –Y. Kwak, and S. –J. Lee, "Building a Service-Oriented Ontology for Wireless Sensor Networks", *Proceedings of International Conference on Computer and Information Science* (ICCIS'08), 2008.

[Kim et al 2010] T. Kim, I. Park, S. J. Hyun, and D. Lee, "MiRE4OWL: Mobile Rule Engine for OWL", *Proceedings of the 2nd IEEE International Workshop on Middleware Engineering (*ME'10*)*, Pages 317-322, 2010.

[Kiryakov et al 2005] A. Kiryakov, D. Ognyanov, and D. Manov, "Owlim-a pragmatic semantic repository for owl", *Proceedings of the 6th Web Information Systems Engineering Workshop (*WISE'05*)*, Pages 182 – 192, 2005.

[Kleemann 2006] T. Kleemann, "Towards mobile reasoning", *Proceedings of the 2006 international workshop on description logics* (DL'06), 2006.

[Kleemann and Sinner 2006] T. Kleemann and A. Sinner, "User Profiles and Matchmaking on Mobile Phones", *Proceedings of the 16th*

International Conference on Applications of Declarative Programming for Knowledge Management (INAP'09), Pages 135-147, 2006.

[Kolas et al 2009] Dave Kolas, Ian Emmons, and Mike Dean, "Efficient Linked-List RDF Indexing in Parliament", *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems* (SSWS'09), Pages 17-32, 2009.

[Krötzsch et al 2010] M. Krötzsch, A. Mehdi, and S. Rudolph, "Orel: Database-Driven Reasoning for OWL 2 Profiles", *Proceedings of the 23rd International Workshop on Description Logics (*DL'10*)*, Pages 114-124, 2010.

[Lee and Schor 1992] H. S. Lee and M. I. Schor, "Match algorithms for generalized RETE networks", *Journal of Artficial Intelligence,* Volume 54, Issue 3, Pages 249-274, 1992.

[Levy and Rousset 1998] A. Y. Levy and M. Rousset, "Combining horn rules and description logic in CARIN", *Artificial Intelligence*, Volume 104, Issue 1-2, Pages 165-209, 1998.

[Liang et al 2009] S. Liang, P. Fodor, H. Wan, and M. Kifer, "OpenRuleBench: An Analysis of the Performance of Rule Engine", *Proceedings of the 18th International World Wide Web Conference (*WWW'09*)*, Pages 601-610, 2009.

[Liebig and Noppens 2004] T. Liebig and O. Noppens, "ONTOTRACK: Combining Browsing and Editing with Reasoning and Explaining for OWL Lite Ontologies", *Proceedings of the 3rd International Semantic Web Conference (*ISWC'04*)*, Pages 244-257, 2004.

[Luther et al 2008] M. Luther, Y. Fukazawa, M. Wagner, S. Kurakake, T. Naganuma, M. Wagner, and S. Kurakake, "Situational reasoning for task-oriented mobile service recommendataion", *Journal of the Knowledge Engineering Review*, Volume 23, Issue 1, 2008.

[Luther and Böhm 2009] M. Luther and S. Böhm, "Situation-Aware Mobility: An Application for Stream Reasoning", *Proceedings of the 1st International Workshop on Stream Reasoning* (SR'09), 2009.

[Lutz 1999] C. Lutz, “Reasoning with Concrete Domains”, *Proceedings of International Joint Conference on Artificial Intelligence* (IJCAI’99), 1999.

[Ma et al 2006] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu, "Towards a complete OWL ontology benchmark”, *Proceedings of the 3rd European Semantic Web Conference* (ESWC’06), Pages 125-139, 2006.

[Matheus et al 2006] C. J. Matheus, K. Baclawski, and M. M. Kokar, "BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rules”, *Proceedings of the 2nd International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML'06)*, Pages 67-74, 2006.

[McDermott et al 1978] J. McDermott, A. Newell, and J. Moore, "The efficiency of certain production system implementations”, *Pattern-directed inference systems*, Academic Press, Orlando, FL, 1978.

[McGuinness and Van Harmelen 2004] D. L. McGuinness and F. Van Harmelen, "OWL Web Ontology Language Overview”, *W3C Recommendation,* Available at: http://www.w3.org/TR/owl-features/, 2004.

[Meditskos and Bassiliades 2008a] G. Meditskos and N. Bassiliades, "Combining a DL Reasoner and a Rule Engine for Improving Entailment-Based OWL Reasoning”, *Proceedings of the 7th International Conference on The Semantic Web (*ISWC'08*)*, 2008.

[Meditskos and Bassiliades 2008b] G. Meditskos and N. Bassiliades, "A Rule-Based Object-Oriented OWL Reasoner”, *IEEE Transactions on Knowledge and Data Engineering,* Volume 20, Issue 3, Pages 397-410, 2008.

[Meditskos and Bassiliades 2010] G. Meditskos and N. Bassiliades, "DLEJena: A practical forward-chaining OWL 2 RL reasoner combining Jena and Pellet”, *Journal of Web Semantics: Science, Services and Agents on the World Wide Web,* Volume 8, Issue 1, Pages 89-94, 2010.

[Mei et al 2005] J. Mei, E. P. Bontas, and Z. Lin, "OWL2Jess: A Transformational Implementation of the OWL Semantics”,

*Proceedings of the 3rd International Symposium on Parallel and Distributed Processing and Applications Workshop (*ISPA'05*)*, Pages 599-608, 2005.

[Mendler and Scheele 2009] M. Mendler and S. Scheele, "Towards a Type System for Semantic Streams", *Proceedings of the 1st International Workshop on Stream Reasoning* (SR'09), 2009.

[Micro Jena 2010] Micro Jena (μJena), Available at: http://poseidon.ws.dei.polimi.it/ca/?page_id=59, Last visited: Oct. 2011.

[Miranker 1987] D. P. Miranker, "TREAT: A better match algorithm for AI production systems", *Proceedings of the 6th National Conference on Artifical Intelligence (*AAAI'87*)*, Pages 42-47, 1987.

[Motik 2007] B. Motik, R. Shearer, I. Horrocks, "Optimized Reasoning in Description Logics Using Hypertableaux," Proceedings of the 21st International Conference on Automated Deduction (CADE'07), Pages 67-83, 2007.

[Motik 2008] B. Motik, "KAON2 - Scalable Reasoning over Ontologies with Large Data Sets", *ERCIM news,* Volume 2008, Issue 72, 2008.

[Motik and Sattler 2006] B. Motik and U. Sattler, "A comparison of reasoning techniques for querying large description logic aboxes", *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (*LPAR'06*)*, Pages 227-241, 2006.

[Motik et al 2006] B. Motik, I. Horrocks, R. Rosati, and U. Sattler, "Can OWL and Logic Programming Live Together Happily Ever After", *Proceedings of the 5th International Semantic Web Conferece (*ISWC'06*)*, Pages 501-514, 2006.

[Motik et al 2007] B. Motik, I. Horrocks, and U. Sattler, "Adding Integrity Constraints to OWL", *Proceedings of the Workshop on OWL: Experiences and Directions* (OWLED'07), 2007.

[Motik et al 2009] B. Motik, R. Shearer, and I. Horrocks, "Hypertableau reasoning

for description logics", *Journal of Artificial Intelligence Research*, Volume 36, Issue 1, Pages 165-228, 2009.

[Nayak et al 1988] P. Nayak, A. Gupta, and P. Rosenbloom, "Comparison of the Rete and Treat production matchers for Soar (A summary)", *Proceedings of the 7th National Conference on Artificial Intelligence (*AAAI'88*)*, Pages 693-698, 1988.

[Nerode and Shore 1997] A. Nerode and R. A. Shore, *Logic for Applications*, 2nd edition, Springer-Verlag, New York, NY, 1997.

[Obermeyer et al 1995] L. Obermeyer, D. P. Miranker, and D. Brant, "Selective indexing speeds production systems", *Proceedings of the 7th International Conference on Tools with Artificial Intelligence (*ICTAI'95*)*, Pages 15-12, 1995.

[openGALEN] The openGALEN project, Available at: http://www.opengalen.org/index.html, Last visited: Oct. 2011.

[OWLJessKB 2011] OWLJessKB: a semantic web reasoning tool. Available at: http://edge.cs.drexel.edu/assemblies/software/owljesskb/, Last visited: Oct. 2011.

[OWL Test Case Results] OWL Test Results (Semi-Official Semi-Static View), Available at: http://www.w3.org/2003/08/owl-systems/test-results-out, Last visited: Oct. 2011.

[OWL 2 Direct Semantics] B. Motik, P. F. Patel-Schneider, and B. C. Grau, "OWL 2 Web Ontology Language: Direct Semantics", *W3C Recommendation,* Aailable at: http://www.w3.org/TR/owl2-direct-semantics/, 2009.

[OWL 2 Overview] OWL 2 Working Group, "OWL 2 Web Ontology Language Document Overview", *W3C Recommendation,* Available at: http://www.w3.org/TR/owl2-overview/, 2009.

[OWL 2 Profiles] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz, "OWL 2 Web Ontology Language Profiles", *W3C Recommendation,* Available at: http://www.w3.org/TR/owl-profiles/, 2009.

[OWL 2 RDF-Based Semantics] M. Schneider, "OWL 2 Web Ontology Language: RDF-

Based Semantics", *W3C Recommendation,* Available at: http://www.w3.org/TR/owl2-rdf-based-semantics/, 2009.

[Özacar et al 2007] T. Özacar, Ö. Öztürk, and M. O. Ünalir, "Optimizing a Rete-based Inference Engine using a Hybrid Heuristic and Pyramid based Indexes on Ontological Data", *Journal of Computers,* Volume 2, Issue 4, Pages 41-48, 2007.

[Pan 2004] J. Z. Pan. "Reasoning Support for OWL-E (Extended Abstract)", *Proceedings of Doctoral Programme in the 2004 International Joint Conference of Automated Reasoning* (IJCAR'04), 2004.

[Paolucci et al 2002] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara, "Semantic Matching of Web Services Capabilities", *Proceedings of the 1st International Semantic Web Conference* (ISWC'02), Pages 333-347, 2002.

[Parsia et al 2006] B. Parsia, C. Halaschek-Wiener, and E. Sirin, "E.S.: Towards Incremental Reasoning Through Updates", *Proceedings of the 15th International World Wide Web Conference* (WWW'06), 2006.

[Patel-Schneider et al 2004] P. F. Patel-Schneider, P. Hayes, and I. Horrocks, "Web Ontology Language (OWL) Abstract Syntax and Semantics", *W3C Recommendation,* Available at: http://www.w3.org/TR/owl-semantics/ , 2004.

[Pellet ICV] Pellet ICV Integrity Constraints Validator, Available at: http://clarkparsia.com/pellet/icv/, Last visited: Oct. 2011.

[PelletDB] Introducing PelletDB – Expressive, Scalable Semantic Reasoning for Enterprise, Available at: http://clarkparsia.com/pelletdb/, Last visited: Oct. 2011.

[RacerPro Release Notes v1.9.2] RacerPro 1.9.2 BETA Release Notes, Available at: http://www.racer-systems.com/products/racerpro/manual.phtml, Last visited: Oct. 2011.

[RacerPro Reference Manual v1.9.2] RacerPro Reference Manual v1.9.2, Available at: http://www.racer-systems.com/products/racerpro/manual.phtml,

last visited: Oct. 2011.

[RDFS++] RDFS++ Overview, Available at: http://www.franz.com/agraph/support/learning/Overview-of-RDFS++.lhtml, Last visited, Oct. 2011.

[Rector 2002] A. Rector, "Analysis of propagation along transitive roles: Formalisation of the galen experience with medical ontologies". *Proceedings of the international workshop on Description Logic* (DL'02), 2002.

[Ren et al 2010a] Y. Ren, J. Z. Pan, and Y. Zhao, "Towards Scalable Reasoning on Ontology Streams via Syntactic Approximation", *Proceedings of International Workshop on Ontology Dynamics (*IWOD'10*)*, 2010.

[Ren et al 2010b] Y. Ren, J. Z. Pan, and Y. Zhao, "Soundness Preserving Approximation for TBox Reasoning", *Proceedings of the 25th AAAI Conference (*AAAI'10*)*, 2010.

[Ren et al 2010c] Y. Ren, J. Z. Pan, and Y. Zhao, "Towards Soundness Preserving Approximation for ABox Reasoning of OWL2", *Proceedings of the International Description Logic Workshop (*DL'10*)*, Pages 325-335, 2010.

[Rosati 1999] R. Rosati, "Towards expressivity KR systems integrating datalog and description logics", *Proceedings of the international workshop on description logic* (DL'99), 1999.

[Russomanno et al 2005] D. J. Russomanno, C. R. Kothari, and O. A. Thomas, "Building a Sensor Ontology: A Practical Approach Leveraging ISO and OGC Models", *Proceedings of The 2005 International Conference on Artificial Intelligence* (ICAI'05), 2005.

[Scales 1986] D. J. Scales, "Efficient Matching Algorithm for the SOAR/OPS5 Production System", *Technical Report KSL-86-47,* Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Stanford, 1986.

[Schmolze and Snyder 1997] J. G. Schmolze and W. Snyder, "Detecting redundant production rules", *Proceedings of the 14th National Conference*

on Artificial Intelligence (AAAI'97) *and the 9th Conference on Innovative Applications of Artificial Intelligence* (IAAI'97), 1997.

[Schuegerl et al 2008] P. Schuegerl, J. Rilling, and P. Charland, "Enriching SE Ontologies with Bug Report Quality", *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering* (SWESE'08), 2008.

[Seitz et al 2010] C. Seitz, S. Lamparter, T. Schöler, and M. Pirker, "Embedded Rule-based Reasoning for Digital Product Memories", *Proceedings of 2010 AAAI Spring Symposium Series* (AAAI - SSS'10), 2010.

[Sesame User Guide] User Guide for Sesame 2.3, Available at: http://www.openrdf.org/doc/sesame2/users/, Last visited: October 2011.

[Shahri et al 2007] H. H. Shahri, J. A. Hendler, and A. A. Porter, "Software configuration management using ontologies", *Proceedings of International Workshop on Semantic Web Enabled Software Engineering*, 2007.

[Sheth et al 2008] A. Sheth, C. Henson, and S. S. Sahoo, "Semantic Sensor Web", *IEEE Internet Computing Magazine*. Volume 12, Issue 4, Pages 78-83, 2008.

[Sinner and Kleemann 2005] A. Sinner and T. Kleemann, "KRHyper - In Your Pocket", *Proceedings of the 20th International Conference on Automated Deduction* (CADE05), Pages 452-457, 2005.

[Sirin and Parsia 2007] E. Sirin and B. Parsia, "SPARQL-DL:SPARQL Query for OWL-DL", *Proceedings of the 3rd OWL: Experiences and Directions Workshop* (OWLED'07), 2007.

[Sirin et al 2004] E. Sirin, B. Parsia, and J. Hendler, "Composition-driven filtering and selection of semantic web services", *IEEE Intelligent Systems,* Volume 19, Issue 4, Pages 42-49, 2004.

[Sirin et al 2007] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner", *Web Semantics:*

Science, Services and Agents on the World Wide Web, Volume 5, Issue 2, Pages 51-53, 2007.

[Smitha and Geneseretha 1985] D. E. Smitha and M. R. Geneseretha, "Ordering conjunctive queries", *Journal of Artificial Intelligence,* Volume 26, Issue 2, Pages 171-215, 1985.

[SNOMED] G. Héja, G. Surján and P. Varga, "Ontological analysis of SNOMED CT", *BMC Medical Informatics and Decision Making*, Volume 8, Issue Suppl. 1, 2008.

[SPIN 2011] OWL 2 RL in SPARQL using SPIN, available at: http://composing-the-semantic-web.blogspot.com/2009/01/owl-2-rl-in-sparql-using-spin.html, Last visited: Oct. 2011.

[Squawk JVM] Squawk Java Virtual Machine Project, Available at: https://squawk.dev.java.net/, Last visited: Nov. 2010

[Staudt and Jarke 1996] M. Staudt and M. Jarke, "Incremental maintenance of externally materialized views", *Proceedings of the 22nd International Conference on Very Large Data Bases* (VLDB'96), Pages 75-86, 1996.

[Steller and Krishnaswamy 2008] L. Steller and S. Krishnaswamy, "Pervasive Service Discovery: mTableaux Mobile Reasoning", *Proceedings of the 7th International Conference on Semantic Systems* (I-Semantics'08), Pages 93-101, 2008.

[Stocker and Smith 2008] M. Stocker and M. Smith, "Owlgres: A scalable OWL Reasoner", *Proceedings of the 5th International Workshop on OWL Experiences and Directions* (OWLED'08), 2008.

[Stuckenschmidt et al 2010] H. Stuckenschmidt, S. Ceri, E. D. Valle, and F. v. Harmelen, "Towards Expressive Stream Reasoning", *Proceedings of the Dagstuhl Seminar on Semantic Aspects of Sensor Networks*, 2010.

[SUN SPOT 2010] SUN SPOT project, Available at: http://sunspotworld.com/, Last visited: Oct. 2011.

[Suntisrivaraporn 2008] B. Suntisrivaraporn, "Module extraction and incremental classification: a pragmatic approach for EL+ ontologies",

Proceedings of the 5th European semantic web conference on the semantic web: research and applications (ESWC'08), 2008.

[Surnia 2011] Surnia reasoner, Available at: http://www.w3.org/2003/08/surnia/, Last visited: Oct. 2011.

[SwiftOWLIM ver 2.9.1 SysDoc] "SwiftOWLIM System Documentation ver. 2.9.1", *OWLIM documentation*, 2007, Available at: http://www.ontotext.com/owlim/OWLIMSysDoc.pdf, Last visited: Oct. 2010.

[SWOOP] SWOOP - A Hypermedia-based Featherweight OWL Ontology Editor, Available at: http://www.mindswap.org/2004/SWOOP/, Last visited: Oct. 2011.

[Sycara et al 2003] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan, "Automated discovery, interaction and composition of semantic web services", *Journal of Web Semantics,* Volume 1, Issue 1, Pages 27-46, 2003.

[Tai et al 2009] W. Tai, R. Brennan, J. Keeney, and D. O'Sullivan, "An Automatically Composable OWL Reasoner for Resource Constrained Devices", *Proceeding of 3rd IEEE International Conference on Semantic Computing* (ICSC'09), Pages 495-502, 2009.

[Tai et al 2011] W. Tai, J. Keeney, and D. O'Sullivan, "COROR: A COmposable Rule-entailment Owl Reasoner for Resource-Constrained Devices", *Proceeding of 5th International Symposium on Rules: Research Based and Industry Focused co-located with the 22nd International Join Conference on Artificial Intelligence* (RuleML'11), 2011.

[Tambe and Rosenbloom 1989] M. Tambe and P. Rosenbloom, "Eliminating expensive chunks by restricting expressiveness", *Proceedings of the 11th International Joint Conference on Artificial Intelligence* (IJCAI'89), 1989.

[Tambe et al 1992] M. Tambe, D. Kalp, and P. S. Rosenbloom, "An efficient algorithm for production systems with linear-time match", *Proceedings of the 4th International Conference on Tools with*

Artificial Intelligence (ICTAI'92), Pages 36-43, 1992.

[Tao et al 2010] J. Tao, E. Sirin, J. Bao, D. L. McGuinness, "Integrity Constraints in OWL", *Proceedings of the 24th AAAI Conference on Artificial Intelligence* (AAAI'10), 2010.

[ter Horst 2005a] H. J. ter Horst, "Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary", *Web Semantics: Science, Services and Agents on the World Wide Web,* Volume 3, Issue 2-3, Pages 79-115, 2005.

[ter Horst 2005b] H. J. ter Horst, "Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity", *Proceedings of the 4th Internation Semantic Web Conference* (ISWC'05), Pages 668 – 684, 2005.

[Tobies 2001] S. Tobies, "Complexity Results and Practical Algorithms for Logics in Knowledge Representation", *PhD Dissertation*, LuFG Theoretical Computer Science, RWTH Aachen University, Aachen, 2001.

[Tsarkov and Horrocks 2006] D. Tsarkov and I. Horrocks, "FaCT++ Description Logic Reasoner: System Description", *Proceedings of the 3rd International Joint Conference on Automated Reasoning* (IJCAR'06), Pages 292-297, 2006.

[Tsarkov et al 2006] D. Tsarkov, A. Riazanov, S. Bechhofer, and I. Horrocks, "Using Vampire to Reason with OWL", *Proceedings of 3rd International Semantic Web Conference* (ISWC'04), Pages 471-485, 2004.

[Ullman 1982] J. D. Ullman, *Principles of Database Systems*, 2nd edtion, Computer Science Press, New York, 1982.

[Ushchold et al 2003] M. Ushchold, P. Clark, F. Dickey, C. Fung, S. Smith, S. Uczekaj, M. Wilke, S. Bechhofer, and I. Horrocks, "A semantic infosphere", *Proceedings of the 2nd International Semantic Web Conference* (ISWC'03), Pages 882-896, 2003.

[van der Gaag and de Koning 1994] L. C. van der Gaag and C. de Koning, "Reason

Maintenance for Production Systems", *Technical Report*, Department of Computer Science, Utrecht University, 1994.

[Vassiliadis et al 2009] V. Vassiliadis, J. Wielemaker, and C. Mungall, "Processing OWL2 ontologies using Thea: An application of logic programming", *Proceedings of the 5th International Workshop on OWL: Experiences and Directions* (OWLED'09), 2009.

[Volz et al 2003] R. Volz, S. Decker, and D. Oberle, "Bubo-Implementing OWL in rule-based systems", *Proceedings of the 12th international conference on World Wide Web* (WWW'03), 2003.

[Volz et al 2005] R. Volz, S. Staab, and B. Motik, "Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases", *Journal of Data Semantics II,* Volume 2260/2005, Pages 1-34, 2005.

[Wang and Hanson 1992] Y. Wang and E. N. Hanson, "A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions", *Proceedings of the 8th International Conference on Data Engineering* (ICDE'92), Pages 88-97, 1992.

[Wang et al 2004] X. Wang, Q. Zhang, T. Gu, and H. K. Pung, "Ontology-based context modeling and reasoning using OWL", *Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops* (PerCom Workshop'04), Pages 18-22, 2004.

[Weißenberg 2004] N. Weißenberg, "Using ontologies in personalized mobile applications", *Proceedings of the 12th annual ACM international workshop on geographic information systems* (GIS'04), 2004.

[Witmate 2011] Mate your wit: the most performed mobile/embedded java logic/rule engine, Available at: http://www.witmate.com/default.html, Last visited: Oct. 2011.

[Wright and Marshall 2003] I. Wright and J. Marshall, "The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case", *International Journal of Intelligent Games and Simulation,* Volume 2, Issue 1, Pages 36-48, 2003.

[Wu et al 2008] Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan, "Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle", *Proceedings of IEEE 24th International Conference on Data Engineering* (ICDE'08), Pages 1239-1248, 2008.

[Zhang et al 2004] L. Zhang, Y. Yu, J. Lu, C. Lin, K. Tu, M. Guo, Z. Zhang, G. Xie, Z. Su, and Y. Pan, "ORIENT: Integrate Ontology Engineering into Industry Tooling Environment", *Proceedings of the 3rd International Semantic Web Conference* (ISWC'04), Pages 823-837, 2004.

[Zhou et al 2006] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan, "Minerva: A scalable OWL ontology storage and inference system", *Proceedings of the 1st Aisan Semantic Web Conference* (ASWC'06), Pages 429 – 443, 2006.

[Zou et al 2004] Y. Zou, T. Finin, and H. Chen, "F-OWL: an Inference Engine for Semantic Web", *Proceedings of the 3rd International Workshop on Formal Approaches to Agent-Based Systems* (FAABS'04), Pages 16-18, 2004.

[µJena] µJena, Available at: http://poseidon.elet.polimi.it/ca/?page_id=59, Last visited: Oct. 2011.

Appendix A
A Survey on OWL Reasoners

A survey is conducted on OWL reasoners in this research to construct the reasoner categorization. In total 26 reasoners were surveyed. 18 reasoner characteristics were surveyed for each of the reasoners. Survey was conducted through literature review, web browsing, and examination of code. Results are presented in the following tables (

Table **A-1** to Table A-9). A color scheme is used to give readers a general idea the level a reasoner characteristic is satisfied by a reasoner: blue indicates this characteristic is not comparable among reasoners; red means this characteristic is not satisfied; green means this characteristic is (relatively) better satisfied than the others; yellow means this characteristic is (relatively) less satisfied compared to the other reasoners.

Table A-1: Results of the survey of OWL reasoners

	Bossam	**Hoolet**	**Pellet**
Reasoning algorithm	RETE	FOL prover (Vampire)	DL tableaux
Reasoner type	Rule-entailment	Resolution-based	DL-tableaux
Reasoner expressivity	OWL DL	OWL DL	OWL DL and OWL 2 EL
Completeness	No	Yes	Complete for OWL DL
Reasoning tasks	Entailment, Conjunctive query answering	Unknown	Entailment, Conjunctive query answering (through Ortiz API), KB consistency, Concept Satisfiability, Classification, Realization
Materialization	Yes (total)	Unknown	Yes
Incremental reasoning	Addition, No deletion support	Unknown	Consistency (not stable), Classification (accessed through OWLAPI)
Query support	Buchingae rule language (atomic, conjunctive)	Simple queries	SPARQL (Oritz, Jena ARQ, Protégé SAPRQL engine)
Rule support	Buchingae rule, RuleML, SWRL	SWRL	SWRL (DL-safe)
Closed-world features	rule, query	Unknown	Closed-world OWL semantics for integrity constraints (PelletDB), rule, query
Concrete domain	XSD datatypes, datatype computation/comparison	XSD datatypes, datatype computation/comparison	XSD datatypes, User-defined datatypes, datatype computation/comparison
Database support	No	No	reasoning (PelletDB)
Remote interface	Self-defined interface for distributed reasoning	No	DIG
User access	Command line	GUI	GUI (protégé), Command line (The Prolog OWL Shell)
Explanation	Native	OWLAPI blackbox	Native and OWLAPI blackbox
Ontology manipulation	API	OWLAPI	API (Ortiz), Jena and OWLAPI
Platform	J2SE (entire) or J2ME CDC (core)	Java	J2SE
OS	Windows, Linux, MacOS, Symbian, Android, WinMobile, TinyOS	Linux	Windows, Linux, MacOS

Table A-2: Results of the survey of OWL reasoners (cont'd)

	KAON2	RacerPro (v2.0)	Jena
Reasoning algorithm	Disjunctive Datalog	DL tableaux	Resolution and RETE
Reasoner type	Resolution-based	DL-tableaux	Hybrid
Reasoner expressivity	OWL DL (SHIQ(D) subset)	OWL DL and OWL 2 (SHIQ(D) subset)	OWL DL
Completeness	Yes	Yes	No
Reasoning tasks	Entailment, Conjunctive query answering, KB consistency, Concept satisfiability, Classification	Entailment Conjunctive query answering, KB consistency, Classification, Realization, Concept satisfiability	Entailment, Conjunctive query answering (ARQ), KB consistency, Concept satisfiability, Classification
Materialization	Yes	Yes	Yes (total)
Incremental reasoning	Materialization	Unknown	Addition, Deletion
Query support	SPARQL	nRQL (native) and SPARQL (protégé)	SPARQL (ARQ)
Rule support	SWRL (DL-safe), F-logic rules (function free)	nRQL, SWRL	Jena rules
Closed-world features	Limited NaF support in F-logic rules/ontology	query	rule, query
Concrete domain	XSD datatypes, datatype computation/comparison	XSD datatypes, datatype computation/comparison	XSD datatypes, User-defined datatypes, datatype computation/comparison
Database support	Native	Reasoning (AllegroGraph) and OWLAPI	Access (TDB and SDB)
Remote interface	RMI and DIG	OWLlink and DIG	DIG 1.0 (removed after v2.6.0)
User access	No	GUI (RacerPorter, Protégé)	Command line
Explanation	No	Native and OWLAPI blackbox (?)	Native
Ontology manipulation	API	GUI (RacerPorter, Protégé)	API
Platform	J2SE	C++, J2SE or LISP	J2SE
OS	Windows, Linux, MacOS	Windows, MacOs X and Linux (java adaptor available)	Windows, Linux, MacOS

Table A-3: Results of the survey of OWL reasoners (cont'd)

	FaCT++	Surnia	F-OWL
Reasoning algorithm	DL tableaux	FOL theorem prover (OTTER)	FOL theorem prover (XSB)
Reasoner type	DL-tableaux	Resolution-based	Resolution-based
Reasoner expressivity	(Full) OWL DL and (Partially) OWL 2	OWL Full	OWL Lite, (Partially) OWL DL, (Partially) OWL Full
Completeness	Yes for OWL DL	Unknown	No
Reasoning tasks	KB consistency, Classification, Realization, Concept satisfiability (all reasoning tasks are accessed through OWLAPI)	Unknown	Entailment, Conjunctive query answering
Materialization	Unknown	Unknown	Yes (Tabling in XSB)
Incremental reasoning	Yes (but not clear if it is classification or consistency)	Unknown	Unknown
Query support	Atomic	Unknown	RDQL
Rule support	Unknown	Unknown	F-logic rules
Closed-world features	Unknown	Unknown	rule, query
Concrete domain	XSD datatypes	Unknown	XSD datatypes, Datatype comparison
Database support	Access (through OWLAPI[18])	Unknown	Reasoning over DB using FLORA-2
Remote interface	DIG (before v1.4)	Unknown	Unknown
User access	GUI (through Protégé), Command line (The Prolog OWL Shell)	Command line in input file	Command line and GUI
Explanation	OWLAPI blackbox	Unknown	XSB justification library
Ontology manipulation	OWLAPI v3.1.0	Unknown	No
Platform	C++, Java or LISP	Python	Flora-2, Java
OS	Windows, MacOS, Linux	Windows, Linux	Windows, Unix

[18] OWLAPI does not natively provide database support but there is a third party tool OWLDB enabling database access through OWLAPI. Find in http://sourceforge.net/projects/owldb/ for OWLDB.

Table A-4: Results of the survey of OWL reasoners (cont'd)

	Euler (EYE)	Minerva (IBM IODT)	CEL
Reasoning algorithm	Resolution with Ruler path detection	DL-tableaux for TBox reasoning and SQL engine for ABox reasoning	CEL subsumption algorithm
Reasoner type	Resolution-based	Hybrid	Miscellaneous
Reasoner expressivity	OWL 2 RL	(Partial) OWL DL	OWL 2 EL
Completeness	Yes	No	Yes
Reasoning tasks	Entailment, Conjunctive query answering	Entailment Conjunctive query answering	Classification, KB consistency, Realization
Materialization	Unknown	Unknown	Unknown
Incremental reasoning	Unknown	Unknown	Partial incremental classification
Query support	SPARQL	SPARQL	SPARQL (through Protégé)
Rule support	Unknown	SQL	Unknown
Closed-world features	Unknown	rule, query	query (through Protégé)
Concrete domain	Unknown	XSD datatypes, Datatype comparison and computation	Unknown
Database support	Unknown	Reasoning (on DB2, Derby and HSQLDB)	Unknown
Remote interface	Unknown	DIG	DIG
User access	GUI	GUI (through IBM IODT)	GUI (through Protégé), Command line
Explanation	Native	Depends on each individual reasoner	OWLAPI blackbox
Ontology manipulation	Unknown	IODT	OWLAPI
Platform	Java ,or C#, or Python, or JavaScript, or Prolog	Java	Java, LISP
OS	Windows, Linux	Windows, Linux, MacOS	Windows

Table A-5: Results of the survey of OWL reasoners (cont'd)

	OWL2Jess	OWLLisaKB	QuOnto
Reasoning algorithm	Jess RETE Engine (translates OWL file into Jess facts using an XSLT style sheet)	LISA RETE Engine	DL-Lite query unfolding algorithm
Reasoner type	Rule-entailment	Rule-entailment	Miscellaneous
Reasoner expressivity	Unknown	Most OWL lite	DL-Lite
Completeness	Unknown	Unknown	Yes
Reasoning tasks	OWL entailment, Conjunctive query answering	Entailment, KB Consistency	OWL Entailment, Conjunctive query answering, Classification, KB consistency,
Materialization	Yes (total)	Yes (total)	Unknown
Incremental reasoning	Addition, Deletion	Addition, Deletion (unknown)	Unknown
Query support	Jess queries	Lisa query language	SPARQL (evaluated using SQL engine)
Rule support	Jess rules	Lisa production rules	Unknown
Closed-world features	rule, query	rule, query	Epistemic query answering Identification constraints, Epistemic constraints (all through MASTRO)
Concrete domain	Integer, String, User-defined functions are allowed in rules	Unknown	XSD datatypes
Database support	Unknown	Wilbur triple store	Reasoning
Remote interface	Unknown	Unknown	DIG
User access	Command line, GUI (through Jess)	Unknown	GUI (MASTRO plugin to Protégé, QToolKit, ROWLKit)
Explanation	Unknown	Unknown	Unknown
Ontology manipulation	No	Unknown	OWLAPI (through ROWLKit)
Platform	Java	LISP	Java
OS	Windows, Linux, MacOS	Windows, Linux	Windows, Linux, MacOS

Table A-6: Results of the survey of OWL reasoners (cont'd)

	Owlgres	**BaseVISor**	**Thea**
Reasoning algorithm	DL reasoning and RDBMS	RETE (memory-based), Linear evaluation of rules as SQL (persistent-based)	Prolog (SWI-Prolog)
Reasoner type	Hybrid	Rule-entailment	Resolution-based
Reasoner expressivity	OWL 2 QL	OWL 2 RL (used to be pD*)	OWL 2
Completeness	Unknown	Yes	Unknown
Reasoning tasks	OWL Entailment, Conjunctive query answering, KB consistency, classification	OWL Entailment, Conjunctive query answering, KB consistency, classification	OWL Entailment, Conjunctive query answering,
Materialization	Unknown	Yes (total)	Depends on SWIProlog
Incremental reasoning	Unknown	Addition, Deletion (unknown)	Unknown
Query support	SPARQL	BaseVISor query, RuleML query	Prolog goal clause
Rule support	Unknown	RuleML	SWRL
Closed-world features	query (SPARQL queries are evaluated using PostgreSQL)	rule, query	query, rule
Concrete domain	XSD datatypes, user-defined datatypes and datatype comparison and computation (supported through PostgreSQL)	XSD datatypes, Datatype comparison and computation, User-defined datatypes can be achieved through user-defined rule builtins	XSD datatypes
Database support	Reasoning (PostgreSQL)	Reasoning (through BaseVISor PersistentBatch)	Prolog database
Remote interface	Unknown	Can be deployed as a SOAP, RESTful web service	HTTP client/server libraries in SWI-Prolog
User access	Unknown	Command line, GUI (through BaseVISor plugin for TopBraid Composer)	Command line (The Prolog OWL Shell)
Explanation	Unknown	No	Unknown
Ontology manipulation	OWLAPI, Jena	API	OWLAPI, SWI-Prolog semweb library
Platform	Java	Java	SWI-Prolog
OS	Windows, Linux, MacOS	Windows, Linux, MacOS	Windows, Linux, MacOS

Table A-7: Results of the survey of OWL reasoners (cont'd)

	Oracle db	**SwiftOWLIM**	**BigOWLIM**
Reasoning algorithm	RDBMS, SQL Engine	Forward-chaining (TRREE engine)	Forward-chaining (TRREE engine with owl:sameAs optimization)
Reasoner type	Miscellaneous	rule-entailment	rule-entailment
Reasoner expressivity	OWL 2 RL	OWL 2 RL, OWL 2 QL, OWL-Horst	OWL 2 RL, OWL 2 QL, OWL-Horst
Completeness	Unknown	No (datatype reasoning is not supported)	No (datatype reasoning is not supported)
Reasoning tasks	OWL Entailment, Conjunctive query answering,	OWL Entailment, conjunctive query answering, classification	OWL Entailment, conjunctive query answering, classification, KB consistency
Materialization	Yes	Yes (total)	Yes (total)
Incremental reasoning	Unknown	Addition	Addition, Deletion
Query support	SQL, SPARQL	SPARQL and SeRQL (through Sesame)	SPARQL and SeRQL (through Sesame)
Rule support	SWRL	OWLIM rules (JDK v1.6 and above)	OWLIM rules (JDK v1.6 and above)
Closed-world features	rule, query	query, rule	query, rule
Concrete domain	XSD datatypes	No	No
Database support	Reasoning (Oracle DB 11g)	Access (through Sesame)	Reasoning (through Sesame)
Remote interface	Unknown	Web services	Web services
User access	GUI, Command line	Command line (Sesame console)	Command line (Sesame console)
Explanation	Native	Unknown	Unknown
Ontology manipulation	No	Sesame	Sesame
Platform	Oracle DB	Java	Java
OS	Windows	Linux, Windows	Linux, Windows

Table A-8: Results of the survey of OWL reasoners (cont'd)

	O-DEVICE	HermiT	DLEJena
Reasoning algorithm	RETE (CLIPS rule engine using dynamic rule generation)	Hypertableau	Jena for ABox reasoning, Pellet for TBox classification
Reasoner type	rule-entailment	DL-tableaux	Hybrid
Reasoner expressivity	Partial OWL DL	OWL 2 DL	OWL 2 RL
Completeness	No	Yes	Yes
Reasoning tasks	OWL Entailment, conjunctive query answering, KB consistency, classification	OWL Entailment, conjunctive query answering, KB consistency, classification, realization	OWL Entailment, conjunctive query answering, KB consistency, classification, realization
Materialization	Yes (total)	Unknown	Yes
Incremental reasoning	Addition, Deletion	Unknown	Addition and deletion for ABox reasoning only. TBox updates can cause ABox to be reasoned from scratch.
Query support	CLIPS rules	Conjunctive query, SPARQL(through protégé)	SPARQL (Jena ARQ)
Rule support	CLIPS rules	SWRL (DL-safe)	Jena rules
Closed-world features	rule, query	query (through Protégé SPARQL engine), rule	query, rule
Concrete domain	XSD datatypes, User-defined datatypes datatype computation and comparison (all through CLIPS rule pre-defined functions and self-defined functions)	XSD datatypes	XSD datatypes, user-defined datatypes, datatype comparison and computation
Database support	Unknown	Unknown	Unknown
Remote interface	Unknown	Unknown	Unknown
User access	Command line (CLIPS)	Command line, GUI (through Protégé)	Unknown
Explanation	No	OWLAPI blackbox	Jena ABox explanation
Ontology manipulation	No	OWLAPI	Jena
Platform	CLIPS	Java	Java
OS	Windows	Windows, Linux, MacOS	Windows, Linux, MacOS

Table A-9: Results of the survey of OWL reasoners (cont'd)

	Bubo (UHU)	**The OWL Instance Store**	
Reasoning algorithm	Datalog engine (deductive database, XSB)	DL Tableaux and SQL database	
Reasoner type	Resolution-based	Hybrid	
Reasoner expressivity	OWL-Lite	Unknown	
Completeness	Unknown	Unknown	
Reasoning tasks	OWL entailment, conjunctive query answering	OWL entailment, conjunctive query answering	
Materialization	Unknown	Yes for ABox reasoning	
Incremental reasoning	Unknown	Unknown	
Query support	XSB queries, DB2 SQL	SQL	
Rule support	XSB rules, DB2 SQL	SQL	
Closed-world features	Unknown	rule, query (SQL support)	
Concrete domain	XSD datatypes, Datatype comparison and computation	Datatypes in SQL	
Database support	Reasoning (DB2)	Reasoning (JDBC)	
Remote interface	Unknown	DIG	
User access	Unknown	GUI	
Explanation	Unknown	Unknown	
Ontology manipulation	Unknown	Unknown	
Platform	Unknown	Java	
OS	Unknown	Windows, Linux, MacOS	

Appendix B
Scenario descriptions used in the usability experiment of RESP

Description for Knowledge-based Networking

"KBN is a semantic pub/sub message broker that uses an ontology reasoner to perform matchmaking between publications and subscriptions, i.e. a subscription can be propagated to a subscriber if it is matched by the subscription that the subscriber put. Publications arrive at the broker and are updated into a knowledge base (KB) where all knowledge is kept. Subscriptions are specified as conjunctive queries over the KB. A subscription is matched by a publication when the query that represents the subscription is resolved by the reasoner and the publication is sent to the subscriber.

The ontology (wine ontology) used for matchmaking has an expressivity of $\mathcal{SHION}$. All matched publications need to be propagated to the subscriber and therefore the underlying reasoner needs to be able to conduct complete reasoning over this ontology. Furthermore since this broker is designed to support real-time data processing, publications need to be updated into the ABox of the KB and propagated to subscribers immediately after their arrivals. Datatype values could be used in publications and subscriptions. This puts a requirement for the matchmaking algorithm to process XSD Datatypes."

The gold standard ACs for this scenario are:

Expressivity: $\mathcal{SHION}$.

Complete reasoning over $\mathcal{SHION}$.

Queries: conjunctive query.

Frequent ABox update

Concrete domain: XSD Datatype

The gold standard reasoner for this application is Pellet.

Description for BaseVISor:

"BaseVISor is a forward-chaining inference engine using RETE algorithm optimized for processing RDF triples. It supports an expressivity of R-Entailment, a set of entailment rules that supports complete RDFS semantics and a subset of OWL semantics. The vocabulary R-Entailment supports consists of the entire RDFS vocabulary and partial OWL vocabulary. The OWL vocabulary it supports includes FunctionalProperty, Restriction, InverseFunctionalProperty, onProperty, SymmetricProperty, hasValue, TransitiveProperty, someValuesFrom, sameAs, allValuesFrom, inverseOf, differentFrom, disjointWith, equivalentClass, equivalentProperty and intersectionOf. BaseVISor also provides supports for XSD datatypes.

Rules can be authored in BaseVISor using the RIF, RuleML or BaseVISor format. Procedural attachments (either build-in or user-defined) are allowed in the BaseVISor language. Built-in procedural attachments include console output, variable binding, fact base management, equality/inequality function, common mathematical functions and Negation as Failure. In addition conjunctive queries can also be specified using BaseVISor or RuleML either in XML file outside the reasoner (using the Query tag) or in java program (using the Query class).

BaseVISor can run either embedded in Java applications or as a standalone reasoner (command lines are provided). A persistent storage package is available in BaseVISor enabling facts to be stored and reasoned using SQL-computable databases (through JDBC). It is accessible either through a standard alone batch file (the PersistentBatch program) or from within a java program. When the persistent storage is used rules and queries are evaluated as SQL statements in the database."

The gold standard RCs for the BaseVISor are:

Reasoner type: Rule-Entailment Reasoner.

Reasoning algorithm: RETE.

Reasoner expressivity: pD*.

Concrete domain: XSD datatypes.

Rule support: RuleML rules and BaseVISor rules.

Query support: BaseVISor queries

Closed-world features: Negation as Failure in rule

User access: command line

Ontology manipulation: API

Platform: J2SE.

Persistent KB: connect to SQL-compatible database through JDBC.

Appendix C
pD* Entailment and Its Implementation in Jena Rule Format

This appendix gives the full set of pD* rules used in COROR.

D* entailment rules

[lg-rdfs1: (*?v ?p ?l*), isPLiteral(*?l*), assignAnon(?l, ?b) -> (?v ?p ?b), (?b rdf:type rdfs:Literal)]

[lg-rdfs2D: (?v ?p ?l), isDLiteral(?l, ?t), assignAnon(?l, ?b) -> (?v ?p ?b), (?b rdf:type ?t)]

[rdf1: (?v ?p ?w) -> (?p rdf:type rdf:Property)]

[rdfs2: (?p rdfs:domain ?u), (?v ?p ?w) -> (?v rdf:type ?u)]

[rdfs3: (?p rdfs:range ?u), (?v ?p ?w), notLiteral(?w) -> (?w rdf:type ?u)]

[rdfs4a: (?v ?p ?w) -> (?v rdf:type rdfs:Resource)]

[rdfs4b: (?v ?p ?w), notLiteral(?w) -> (?w rdf:type rdfs:Resource)]

[rdfs5: (?v rdfs:subPropertyOf ?w), (?w rdfs:subPropertyOf ?u) -> (?v

rdfs:subPropertyOf ?u)]

[rdfs6: (?v rdf:type rdf:Property) -> (?v rdfs:subPropertyOf ?v)]

[rdfs7x: (?p rdfs:subPropertyOf ?q), (?v ?p ?w) -> (?v ?q ?w)]

[rdfs8: (?v rdf:type owl:Class) -> (?v rdfs:subClassOf rdfs:Resource)]

[rdfs9: (?v rdfs:subClassOf ?w), (?u rdf:type ?v) -> (?u rdf:type ?w)]

[rdfs10: (?v rdf:type owl:Class) -> (?v rdfs:subClassOf ?v)]

[rdfs11: (?v rdfs:subClassOf ?w), (?w rdfs:subClassOf ?u) -> (?v rdfs:subClassOf ?u)]

[rdfs12: (?v rdf:type rdfs:ContainerMembershipProperty) -> (?v rdfs:subPropertyOf rdfs:member)]

[rdfs13: (?v rdf:type rdfs:Datatype) -> (?v rdfs:subClassOf rdfs:Literal)]

P-entailment rules

[rdfp1: (?p rdf:type owl:FunctionalProperty), (?u ?p ?v), (?u ?p ?w), notLiteral(?v) -> (?v owl:sameAs ?w)]

[rdfp2: (?p rdf:type owl:InverseFunctionalProperty), (?u ?p ?w), (?v ?p ?w) -> (?u owl:sameAs ?v)]

[rdfp3: (?p rdf:type owl:SymmetricProperty), (?v ?p ?w), notLiteral(w) -> (?w ?p ?v)]

[rdfp4: (?p rdf:type owl:TransitiveProperty), (?u ?p ?v), (?v ?p ?w) -> (?u ?p ?w)]

[rdfp5a: (?v ?p ?w) -> (?v owl:sameAs ?v)]

[rdfp5b: (?v ?p ?w), notLiteral(?w) -> (?w owl:sameAs ?w)]

[rdfp6: (?v owl:sameAs ?w), notLiteral(?w) -> (?w owl:sameAs ?v)]

[rdfp7: (?u owl:sameAs ?v), (?v owl:sameAs ?w) -> (?u owl:sameAs ?w)]

[rdfp8ax: (?p owl:inverseOf ?q), (?v ?p ?w), notLiteral(?w) -> (?w ?q ?v)]

[rdfp8bx: (?p owl:inverseOf ?q), (?v ?q ?w), notLiteral(?w) -> (?w ?p ?v)]

[rdfp9: (?v rdf:type owl:Class), (?v owl:sameAs ?w) -> (?v rdfs:subClassOf ?w)]

[rdfp10: (?p rdf:type rdf:Property), (?p owl:sameAs ?q) -> (?p rdfs:subPropertyOf ?q)]

[rdfp11: (?u ?p ?v), (?u owl:sameAs ?up), (?v owl:sameAs ?vp), notLiteral(?up) -> (?up ?p ?vp)]

[rdfp12a: (?v owl:equivalentClass ?w) -> (?v rdfs:subClassOf ?w)]

[rdfp12b: (?v owl:equivalentClass ?w), notLiteral(?w) -> (?w rdfs:subClassOf ?v)]

[rdfp12c: (?v rdfs:subClassOf ?w), (?w rdfs:subClassOf ?v) -> (?v owl:equivalentClass ?w)]

[rdfp13a: (?v owl:equivalentProperty ?w) -> (?v rdfs:subPropertyOf ?w)]

[rdfp13b: (?v owl:equivalentProperty ?w), notLiteral(?w) -> (?w rdfs:subPropertyOf ?v)]

[rdfp13c: (?v rdfs:subPropertyOf ?w), (?w rdfs:subPropertyOf ?v) -> (?v owl:equivalentProperty ?w)]

[rdfp14a: (?v owl:hasValue ?w), (?v owl:onProperty ?p), (?u ?p ?w) -> (?u rdf:type ?v)]

[rdfp14bx: (?v owl:hasValue ?w), (?v owl:onProperty ?p), (?u rdf:type ?v), notLiteral(?p) -> (?u ?p ?w)]

[rdfp15: (?v owl:someValuesFrom ?w), (?v owl:onProperty ?p), (?u ?p ?x), (?x rdf:type ?w) -> (?u rdf:type ?v)]

[rdfp16: (?v owl:allValuesFrom ?w), (?v owl:onProperty ?p), (?u rdf:type ?v), (?u ?p ?x), notLiteral(?x) -> (?x rdf:type ?w)]

Appendix D
Rule-Construct Mappings

lg-rdfs1:rdfs:[]->[]

lg-rdfs2D:rdfs:[]->[]

rdf1:rdfs:[]->[rdf:Property]

rdfs2:rdfs:[rdfs:domain]->[]

rdfs3:rdfs:[rdfs:range]->[]

rdfs4a:rdfs:[]->[]

rdfs4b:rdfs:[]->[]

rdfs5:rdfs:[rdfs:subPropertyOf]->[rdfs:subPropertyOf]

rdfs6:rdfs:[rdf:Property]->[rdfs:subPropertyOf]

rdfs7x:rdfs:[rdfs:subPropertyOf]->[]

rdfs8:rdfs:[]->[rdfs:subClassOf]

rdfs9:rdfs:[rdfs:subClassOf]->[]

rdfs10:rdfs:[rdf:Property]->[rdfs:subClassOf]

rdfs11:rdfs:[rdfs:subClassOf]->[rdfs:subClassOf]

rdfs12:rdfs:[rdfs:ContainerMembershipProperty]->[rdfs:subPropertyOf]

rdfs13:rdfs:[rdfs:Datatype]->[rdfs:subClassOf,rdfs:Literal]

rdfp1:owl-lite:[owl:FunctionalProperty]->[owl:sameAs]

rdfp2:owl-lite:[owl:InverseFunctionalProperty]->[owl:sameAs]

rdfp3:owl-lite:[owl:SymmetricProperty]->[]

rdfp4:owl-lite:[owl:TransitiveProperty]->[]

rdfp5a:owl-lite:[]->[owl:sameAs]

rdfp5b:owl-lite:[]->[owl:sameAs]

rdfp6:owl-lite:[owl:sameAs]->[owl:sameAs]

rdfp7:owl-lite:[owl:sameAs]->[owl:sameAs]

rdfp8ax:owl-lite:[owl:inverseOf]->[]

rdfp8bx:owl-lite:[owl:inverseOf]->[]

rdfp9:owl-lite:[owl:sameAs]->[rdfs:subClassOf]

rdfp10:owl-lite:[rdf:Property,owl:sameAs]->[rdfs:subPropertyOf]

rdfp11:owl-lite:[owl:sameAs]->[]

rdfp12a:owl-lite:[owl:equivalentClass]->[rdfs:subClassOf]

rdfp12b:owl-lite:[owl:equivalentClass]->[rdfs:subClassOf]

rdfp12c:owl-lite:[rdfs:subClassOf]->[owl:equivalentClass]

rdfp13a:owl-lite:[owl:equivalentProperty]->[rdfs:subPropertyOf]

rdfp13b:owl-lite:[owl:equivalentProperty]->[rdfs:subPropertyOf]

rdfp13c:owl-lite:[rdfs:subPropertyOf]->[owl:equivalentProperty]

rdfp14a:owl-lite:[owl:hasValue,owl:onProperty]->[]

rdfp14b:owl-lite:[owl:hasValue,owl:onProperty]->[]

rdfp15:owl-lite:[owl:someValuesFrom,owl:onProperty]->[]

rdfp16:owl-lite:[owl:allValuesFrom,owl:onProperty]->[]

Appendix E
A Full List of the Java Classes Added to µJena to Form the Enhanced µJena

Table E-1: Classes added to µJena to forming the enhanced µJena.

ie.tcd.cs.nembes.microjenaenh.db	***ie.tcd.cs.nembes.microjenaenh.reasoner.rulesys***
RDFRDBException	BasicFBReifier
ie.tcd.cs.nembes.microjenaenh.graph	BasicForwardRuleInfGraph
bulkUpdateHandler	BindingEnvironment
Capabilities	Builtin
NodeVisitor	BuiltinException
Node_Variable	BuiltinRegistory
ie.tcd.cs.nembes.microjenaenh.graph.compose	ClauseEntry
CompositionBase	EnhForwardRETEInfGraph
MultiUnion	EnhForwardRETEReasoner
Polyadic	ForwardRuleInfGraphI
ie.tcd.cs.nembes.microjenaenh.graph.impl	Functor
AllCapabilities	Node_RuleVariable
SimpleBulkUpdateHandler	RETERuleInfGraph
ie.tcd.cs.nembes.microjenaenh.reasoner	Rule
BaseInfGraph	RuleContext
Fgraph	RulePreprocessHook
Finder	RuleReasoner
FinderUtil	SilentAddI
IllegalParameterException	Util
InfGraph	***ie.tcd.cs.nembes.microjenaenh.util.iterator***

Reasoner	ConcatenatedIterator
ReasonerException	Filter
ReasonerFactory	FilterDropIterator
ReasonerRegistry	FilterIterator
TriplePattern	FilterKeepIterator
ValidityReport	***ie.tcd.cs.nembes.microjenaenh.vocabulary***
ie.tcd.cs.nembes.microjenaenh.reasoner.rulesys.enh	ReasonerVocabulary
LiteralStore	

Table E-2: Classes added to µJena to forming the enhanced µJena.

ie.tcd.cs.nembes.microjenaenh.reasoner.rulesys.builtins	***ie.tcd.cs.nembes.microjenaenh.reasoner.rulesys.impl***
BaseBuiltin	BindingVector
Difference	FRuleEngineI
Equal	RETEClauseFilter
GE	RETEConflictSet
GreaterThan	RETEEngine
IsLiteral	RETENode
LE	RETEQueue
LessThan	RETERuleContext
ListContains	RETESinkNode
ListEntry	RETESourceNode
ListEqual	RETETerminal
ListLength	***ie.tcd.cs.nembes.microjenaenh.shared***
ListMapAsObject	DoesNotExistException
ListMapAsSubject	RulesetNotFoundException
ListNotContains	WrappedIOException
ListNotEqual	***ie.tcd.cs.nembes.microjenaenh.util***
MakeTemp	Character
Max	Collection
Min	FileUtils
NotEqual	IteratorCollection
NotLiteral	NumberUtil
Print	OneToManyMap
Product	PrintUtil
Quotient	Tokenizer
Sum	***ie.tcd.cs.nembes.microjenaenh.reasoner.rulesys.builtins.enhbuiltins***
	AssignAnon
	IsDLiteral
	IsPLiteral