Distributed Modular Ontology Reasoning

by

Li Ji

Bachelor of Software Engineering, Southeast University, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor: Weichang Du, Ph.D., Computer Science
Examinining Board: Harold Boley, Ph.D., Computer Science, Chair
Huajie Zhang, Ph.D., Computer Science
Fanrui Meng, Ph.D., Forestry

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK
July, 2018
© Li Ji, 2018
Abstract

This thesis proposes algorithms for a distributed reasoning system over interface-based modular ontologies. The thesis research includes three parts: (1) The algorithm designs for distributed modular ontology reasoning, including TBox (see Glossary 5.) and ABox (see Glossary 1.) reasoning of concept, negated concept, disjunction, conjunction, subsumption, and role queries; (2) The distributed modular ontology reasoning system, comprising system functionality, architecture and functionality realization; and (3) A case study and experiments for evaluating the distributed modular ontology reasoning compared to monolithic ontology reasoning.
Acknowledgements

I would like to thank my supervisor, Dr. Weichang Du, who taught me how to study with patient guidance and made this thesis possible. I also would like to thank my parents and my family for their support and encouragement. I am indebted to Yi Ji, Peng Wang and Xiang Zhang for their kindness and help. Finally, I would like to thank all my friends for their care and support.
Table of Contents

Abstract ii
Acknowledgments iii
Table of Contents ix
List of Tables xiii
List of Figures xiii

1 Introduction 1
1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Research Objective . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 Research Problems . . . . . . . . . . . . . . . . . . . . . . . . 5
1.4 Overview of Proposed Solutions . . . . . . . . . . . . . . . . . 7
1.5 Thesis Structure . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2 Background 15
2.1 Ontology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Ontology Reasoning . . . . . . . . . . . . . . . . . . . . . . . . 21
3 Reasoning Algorithms for Simple Distributed Modular Ontologies

3.1 Simple Distributed Modular Ontology Model 37
3.2 Searching Sub- and Super-Concepts 40
3.3 TBox Reasoning with Single Concept 43
  3.3.1 Concept Reasoning 44
  3.3.2 Negation Reasoning 45
3.4 TBox Reasoning with Two Concepts 45
  3.4.1 Disjunction Reasoning 46
  3.4.2 Conjunction Reasoning 47
  3.4.3 Subsumption Reasoning 50
  3.4.4 Role Reasoning 51
3.5 ABox Reasoning 53
  3.5.1 Instance Checking with Single Concept 53
  3.5.2 Instance Checking with Constructs and Two Concepts 58
3.6 Summary 60

4 Extended Reasoning Algorithms for General Distributed Modular Ontologies

4.1 Extended Models and Reasoning 62
  4.1.1 Multiple Utilizer Model and Reasoning 62
4.1.1.1 Multiple Utilizer Model .......................... 62
4.1.1.2 Query Reasoning Using Same Reasoning
Algorithms for the Simple Model ............... 63
4.1.1.3 Query Reasoning Using Extended Reasoning
Algorithms ........................................ 65

4.1.2 Multiple Realizer Model and Reasoning .......... 68
4.1.2.1 Multiple Realizer Model .......................... 69
4.1.2.2 Query Reasoning Using Same Reasoning
Algorithms for the Simple Model ............... 69
4.1.2.3 Query Reasoning Using Extended Reasoning
Algorithms ........................................ 71

4.1.3 Realizer-Utilizer Model and Reasoning .......... 74
4.1.3.1 Realizer-Utilizer Model .......................... 75
4.1.3.2 Query Reasoning Using Same Reasoning
Algorithms for the Simple Model ............... 76
4.1.3.3 Query Reasoning Using Extended Reasoning
Algorithms ........................................ 76

4.1.4 General Model and Reasoning .......................... 80
4.1.4.1 General Model .......................... 80
4.1.4.2 Query Reasoning Using Same Reasoning
Algorithms for the Simple Model ............... 81
4.1.4.3 Query Reasoning Using Extended Reasoning
Algorithms ........................................ 82
5 Distributed Modular Ontology Reasoning System

5.1 System Functionality .................................. 91
  5.1.1 Information System ................................. 92
    5.1.1.1 Ontology Interface Database ................. 93
    5.1.1.2 Ontology Module Information Database ... 93
    5.1.1.3 Assembled Modular Ontology Information Database .................................. 95
  5.1.2 Distributed Modular Ontology Creation ............ 97
    5.1.2.1 Ontology Assemblage ......................... 97
    5.1.2.2 Assembled Modular Ontology Checking ....... 98
  5.1.3 User Query Processing ............................. 99
    5.1.3.1 Supported System Queries .................... 99
    5.1.3.2 Query User Interface ........................ 101
    5.1.3.3 Query Validation ............................. 101
    5.1.3.4 Query Reasoning .............................. 101

5.2 System Architecture .................................. 102
  5.2.1 Server Software System ........................... 104
    5.2.1.1 Information Management Subsystem .......... 105
    5.2.1.2 Modular Ontology Creation Subsystem ....... 106
    5.2.1.3 Communication Component .................... 107
5.2.2 Local Software System .................................. 107
  5.2.2.1 Query Processing Component ...................... 108
  5.2.2.2 Ontology Reasoning Component .................. 108
  5.2.2.3 Knowledge Base .................................. 109
  5.2.2.4 Communication Component ......................... 110

5.3 Functionality Realization ................................. 110
  5.3.1 Information Management Realization ................ 110
  5.3.2 Modular Ontology Creation Realization ............. 113
  5.3.3 Query Reasoning Realization ........................ 116

5.4 Summary .................................................. 119

6 Case Study and Experiments ................................. 120
  6.1 Publication Modular Ontology ......................... 120
  6.2 Reasoning Query Cases ................................ 122
    6.2.1 Concept Cases .................................. 122
    6.2.2 Negated Concept Cases ........................... 124
    6.2.3 Disjunction Cases ................................ 124
    6.2.4 Conjunction Cases ................................ 126
    6.2.5 Subsumption Cases ................................ 127
    6.2.6 Role Cases ...................................... 128
  6.3 Query Reasoning Examples .............................. 130
    6.3.1 TBox Reasoning for the Conjunctive Query .......... 130
    6.3.2 ABox Reasoning for the Concept Query ............ 133
6.4 Experiments ........................................ 135
6.5 Summary ........................................ 136

7 Conclusion ........................................ 137
7.1 Summary ........................................ 137
7.2 Contribution ..................................... 139
7.3 Future Work ..................................... 139

Bibliography ......................................... 145
Glossary ............................................. 146
Vita
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Tableaux rules for TBox reasoning</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Query reasoning using same reasoning algorithms of the simple model</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Extended conjunction reasoning algorithm</td>
<td>65</td>
</tr>
<tr>
<td>4.3</td>
<td>Extended role reasoning algorithm</td>
<td>66</td>
</tr>
<tr>
<td>4.4</td>
<td>Extended instance checking for concept algorithm</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>Query reasoning using same reasoning algorithms of the simple model</td>
<td>70</td>
</tr>
<tr>
<td>4.6</td>
<td>Extended conjunction reasoning algorithm</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>Extended role reasoning algorithm</td>
<td>73</td>
</tr>
<tr>
<td>4.8</td>
<td>Extended instance checking for concept algorithm</td>
<td>74</td>
</tr>
<tr>
<td>4.9</td>
<td>Query reasoning using same reasoning algorithms of the simple model</td>
<td>76</td>
</tr>
<tr>
<td>4.10</td>
<td>Extended conjunction reasoning algorithm</td>
<td>77</td>
</tr>
<tr>
<td>4.11</td>
<td>Extended role reasoning algorithm</td>
<td>78</td>
</tr>
<tr>
<td>4.12</td>
<td>Extended instance checking for concept algorithm</td>
<td>79</td>
</tr>
<tr>
<td>4.13</td>
<td>Query reasoning using same reasoning algorithms</td>
<td>82</td>
</tr>
</tbody>
</table>
4.14 Extended conjunction reasoning algorithm ................. 83
4.15 Extended role reasoning algorithm ....................... 84
4.16 Extended instance checking for concept algorithm ......... 86

5.1 The table of query samples ................................ 100

6.1 The table of concept cases ............................... 123
6.2 The table of negated concept cases ..................... 124
6.3 The table of disjunction cases .......................... 125
6.4 The table of conjunction cases .......................... 126
6.5 The table of conjunction cases .......................... 127
6.6 The table of role cases .................................. 128
6.7 Testing results of the query cases ....................... 135
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Lassila and McGuinness categorization</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>$EL_{tiny}$ class and property expression</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Three possible ways for employing ontologies</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>A model of interface-based modular ontology</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>An IBF modular ontology</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Intuitive shape of subsumption proofs</td>
<td>33</td>
</tr>
<tr>
<td>2.7</td>
<td>Subsumption propagation</td>
<td>34</td>
</tr>
<tr>
<td>3.1</td>
<td>A model of a simple assembled modular ontology</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>A model of a distributed assembled modular ontology</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>The output sets of the $SearchingSubConcepts$ algorithm</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>A multiple utilizer model</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>A multiple realizer model</td>
<td>69</td>
</tr>
<tr>
<td>4.3</td>
<td>A realizer-utilizer model</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>A general model</td>
<td>81</td>
</tr>
<tr>
<td>4.5</td>
<td>An example of decomposed compound query</td>
<td>88</td>
</tr>
<tr>
<td>5.1</td>
<td>The interface table</td>
<td>93</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Ontologies as the kernel of knowledge representation can provide clarified knowledge structures by ontological analysis [6]. Ontologies are important to many areas such as knowledge engineering, qualitative modelling, language engineering, database design, information modelling, information integration, object-oriented analysis, information retrieval and extraction, knowledge management and organization and agent-based systems design [12]. As a fundamental part of knowledge engineering, domain knowledge uses ontologies to analyse, model and implement [29]. Many researchers consider that ontologies, as a candidate for reuse, would
facilitate the knowledge engineering process [30]. Ontologies can improve the quality of resulting knowledge and manageable performance in knowledge engineering [30]. For language engineering, ontologies can uncover the most basic structures and categories of the commonsense world [3]. In natural language processing, ontology concepts help to generate an Upper Model about experiential semantics through gradually abstracting away from the generalized resources of lexicogrammars [3]. Moreover, ontologies also provide theories to establish agreements in information retrieval and extraction areas [11].

The rapid development and popularization of electronic devices brings a distributed way to maintain and utilize ontologies. Each of devices could require different knowledge from ontologies according to different local requirements in reality. In addition, some of knowledge could be private and inconvenient to be migrated or utilized by other devices. Distributed ontologies are an effective way to deal with these problems. Moreover, the vision of the semantic web can only be fulfilled through aggregation of ontologies from different semantic web providers [18]. In order to support interoperability in the semantic web, it will need to break ontologies into small units to form distributed ontologies [18]. Therefore, distributed ontologies should be a promising option for users of the semantic web.

Generally, there are three forms of distributed ontologies, including distributed multiple ontologies, distributed modular ontologies and
ontologies defined by distributed ontology languages. A multiple ontology system consists of multiple related individual ontologies to be used together. Multiple ontologies can facilitate communication between multiple communities [19]. Usually, multiple ontologies have similarity and semantic bridging components to help find out similarities between ontologies, and establish inter-mapping of ontologies [19] [31] [20]. In a distributed multiple ontology knowledge system, ontologies are distributed on different nodes, which can be reasoned by merging and mapping them distributedly.

Modular ontologies are built on multiple ontology modules, which emphasize extracting and managing related modules of an ontology for applications of special requirements [24]. Generally, modular ontologies are aligned by sets of semantic links between ontology modules [27]. In a distributed modular ontology system, ontology modules are distributed on different nodes, and to be reasoned distributedly. As stated in [26] [24], modules of distributed modular ontologies make sense from the perspectives of applications and systems, like improving performance. Interface-based modular ontologies [8] consist of utilizer modules, realizer modules and interfaces. In interface-based modular ontologies, the utilizer modules reuse knowledge from external realizer modules through connected interfaces.

Distributed ontology languages (DOL) provide language constructs to support defining ontologies in distributed environments, and use
dependence relationships and mappings between distributed ontology components. In other words, DOL aims for ontology interoperability, which builds on a graph of ontology languages and translations [22] [23]. Distributed ontology languages allow users to distribute ontologies with different formalisms [21]. DOL usually provides their own distributed reasoning mechanism.

In this thesis, we will focus on distributed modular ontologies. A modular ontology is defined by ontology modules without notion of distribution. Conventionally, an assembled modular ontology is assumed to be reasoned monolithically. In distributed modular ontologies, an assembled modular ontology cannot be reasoned about as a single ontology. Therefore, it is important to study how to reason distributedly with modular ontologies consisting of distributed ontology modules on distributed nodes.

### 1.2 Research Objective

This thesis investigates and designs effective distributed reasoning algorithms for reasoning about distributed modular ontologies, as well as the design and implementation of a prototype distributed ontology reasoning system.

The aim of the distributed reasoning algorithms is to accomplish reasoning with distributed ontology modules of modular ontologies. The distributed
reasoning algorithms will be able to accomplish distributed TBox reasoning and ABox reasoning, including concept, negated concept, subsumption, conjunction, disjunction reasoning, etc.

The prototype distributed ontology reasoning system will support three types of users: ontology module and interface providers, modular ontology creators, and modular ontology queriers. Using the system, an ontology module or interface provider can manage the module or interface database. A modular ontology creator can create modular ontologies by logically assembling ontology modules and interfaces. After the logically assembling procedure, a newly assembled modular ontology consists of relationships and addresses of selected ontology modules and interfaces. A modular ontology querier can query assembled modular ontologies using the system, and the system provides query results by distributed ontology reasoning based on the designed algorithms.

1.3 Research Problems

The main research challenge is that for reasoning about user queries, we cannot migrate and merge distributed ontology modules into a single monolithic ontology and reasoning with it on a single node. We need to reason about ontology modules distributedly on their local nodes to complete a query reasoning procedure.
The distributed modular ontology algorithms need to ensure that local reasoning in a local ontology module can be correctly performed and will return proper information about local reasoning results. The algorithms also should be able to combine local reasoning results and produce final query results.

The first research problem is about TBox reasoning. The involved concepts of TBox reasoning may be in different modules that cannot be moved and merged. We need to consider how to separately reason about the involved concepts in different modules.

The second research problem is about ABox reasoning. The instances of ABox reasoning can be in different modules that also cannot be moved and merged. We also need to consider how to separately check instances of concepts in different modules.

The third research problem is to determine orders of invoking different local reasoning procedures. A local module could depend on other external modules in an assembled distributed modular ontology. Such reasoning orders determine correctness and performance of distributed query reasoning on distributed ontology modules. We also need to determine what information from local reasoning results can be carried from reasoning about a local module to another local module.
Another research problem is on design of the distributed ontology reasoning system to support functionalities of three types of users, including ontology module and interface providers, modular ontology creators, and modular ontology queriers. For ontology module and interface providers, how to support the ontology modules and interfaces management. For modular ontology creators, how to support assembling and verifying modular ontologies from a set of ontology modules and interfaces. For modular ontology queriers, how to support distributed reasoning of user queries. All of these can bring new challenges to design and implementation of the system.

1.4 Overview of Proposed Solutions

In this thesis, we design distributed reasoning algorithms for reasoning with interface-based modular ontologies whose modules are distributed on different nodes. These reasoning algorithms support distributed reasoning for TBox queries, like concept, negated concept, disjunction, conjunction, subsumption, and role queries, and ABox queries, like instances checking for concept, negated concept, disjunction, conjunction, and role queries.

The concept query only consists of one concept, like $A$. The corresponding concept reasoning algorithm is used to check whether the concept query is satisfiable in an assembled modular ontology. In the concept reasoning
procedure, the algorithm checks satisfiability of $A$ in its defined module and related interfaces using a standard ontology reasoner.

For the negated concept query, like $\neg A$, we add a new named concept $NA=\neg A$ in the module containing $A$ and reasoning about $NA$ using the concept reasoning algorithm.

The disjunctive query consists of two concepts and a disjunction constructor, like $A \sqcup B$. The corresponding disjunction reasoning algorithm is utilized to check whether the disjunctive query is satisfiable in an assembled modular ontology. In the distributed disjunctive query reasoning procedure, the algorithm respectively checks satisfiability of $A$ and $B$ using the concept reasoning algorithm. When $A$ is satisfied, $B$ may not need to be tried, and vice versa. Then, the algorithm compares the computed satisfiability to return final reasoning results.

The conjunctive query consists of two concepts and a conjunction constructor, like $A \sqcap B$. The corresponding conjunction reasoning algorithm is utilized to check whether the conjunctive query is satisfiable in an assembled modular ontology. In the distributed conjunctive query reasoning procedure, the algorithm is different with the disjunction reasoning algorithm. Note that the distributed conjunction reasoning needs to consider negation relations between $A$ and $B$. For example, we assume that $A$ equals $\neg B$. In this situation, although both of $A$ and $B$ are satisfiable,
the final reasoning result of $A \cap B$ is unsatisfiable. The conjunction reasoning algorithm mainly needs to check satisfiability of super-concepts of $A$ conjunction with $B$ in $B$’s module and interface. Based on these reasoning results, the algorithm can deduce the reasoning result of $A \cap B$.

The subsumption query consists of two concepts and a subsumption constructor, like $A \sqsubseteq B$. It can be reasoned by checking $\neg (A \cap \neg B)$ using the conjunction and negation reasoning algorithms.

The role query consists of two concepts and a role constructor, like $A \sqsubseteq R. B$. The corresponding role algorithm aims to determine whether a role query is satisfiable in an assembled modular ontology. In the distributed role reasoning, we assume that $A$ and $R$ are defined in a same module. $R$ and $B$ are defined in different modules. The algorithm needs to find super-concepts of $B$ in interfaces, and uses those concepts to replace the original $B$ of the query to create new queries. After reasoning about the new queries in the related module and interface, the algorithm deduces the final result of the original role query from the reasoning results of the new queries.

The instance checking for concept query consists of one concept and one individual, like $A(a)$. The corresponding reasoning algorithm is used to check whether the query is consistent from open-world reasoning and closed-world reasoning in an assembled modular ontology. The open-world
reasoning is used to determine whether the query is consistent in an assembled modular ontology. The closed-world reasoning is used to determine whether an assembled modular ontology contains the query. In the distributed instance checking of query procedure, the algorithm firstly checks satisfiability of $A$ using the concept reasoning algorithm. If the reasoning result is unsatisfiable, the instance checking for concept algorithm will return inconsistent for the query. If the reasoning result is satisfiable, the instance checking algorithm will find super-concepts and sub-concepts of $A$ in $A$’s interface. The computed super-concepts are used for the open-world instance checking. The computed sub-concepts are used for the closed-world instance checking. Then, the algorithm reasons those computed concepts with individual $a$ in the related modules and interfaces. According to the computed reasoning results, the instance checking algorithm can deduce the final results of $A(a)$.

The instance checking for negated concept query consists of one negation constructor, one concept, and one individual, like $\neg A(a)$. Similarly to the negated concept reasoning, its corresponding reasoning algorithm can also use the instance checking for concept algorithm to compute final reasoning results.

The instance checking for disjunctive query consists of one disjunction constructor, two concepts, and one individual, like $A \sqcup B(m)$. The corresponding reasoning algorithm is used to check whether the query is
consistent in an assembled modular ontology. In the distributed instance checking for disjunctive query procedure, its corresponding reasoning algorithm uses the instance checking for concept algorithm to respectively check consistency of $A(m)$ and $B(m)$. When $A(m)$ is satisfied, $B(m)$ may not need to be reasoned, and vice versa. Then, the algorithm uses the computed sub-reasoning results to compute the final reasoning result.

The instance checking for conjunctive query consists of one conjunction constructor, two concepts, and one individual, like $A \cap B(m)$. Its reasoning algorithm is used to check whether the query is consistent in an assembled modular ontology. In the distributed instance checking for conjunctive query procedure, the reasoning algorithm firstly checks satisfiability of $A \cap B$ using the conjunction reasoning algorithm. If $A \cap B$ is unsatisfiable, the algorithm will return inconsistent result for $A \cap B(m)$. If $A \cap B$ is satisfiable, the algorithm will use the instance checking for concept algorithm to check both $A(m)$ and $B(m)$, then compare the two sub-reasoning results to compute the final reasoning result.

The instance checking for role query consists of one role constructor and two individuals, like $R(a,b)$. The corresponding reasoning algorithm is used to check whether the query is consistent in an assembled modular ontology. In the distributed instance checking for role query procedure, the reasoning algorithm checks whether $a$ in domain($R$), and $b$ in range($R$) are consistent in related modules using a standard ontology reasoner. Then, the algorithm
uses the two sub-reasoning results to compute the final reasoning result, that is the algorithm checks whether $R(a,b)$ is consistent from open-world reasoning in the assembled modular ontology.

The architecture of the distributed ontology reasoning system consists of the following sub-systems: a server system and local systems. The local systems are allocated on each of distributed local nodes.

The server system has three major components, including information management, modular ontology creation, and communication components. The information management component stores ontology interfaces, information of ontology modules, and information of assembled modular ontologies. This component also provides a user interface for ontology module and interface providers to manage data stored in the information management component. The modular ontology creation component provides a user interface for ontology creators to create assembled modular ontologies from stored information of ontology interfaces and modules. The communication component supports communication from distributed local systems.

The local system has four major components, including knowledge base, query processing, ontology reasoning, and communication components. The knowledge base stores one or more ontology modules in a local node. The query processing component provides a user interface for local ontology
queriers to issue queries and receive reasoning results, and the component also processes user queries. The ontology reasoning component distributedly reasons user queries in the local node, and works with other nodes according to the designed reasoning algorithms. The communication component supports a local system to communicate with the server system and other local systems.

1.5 Thesis Structure

This thesis consists of seven chapters, including Introduction, Background, Reasoning Algorithms for Simple Distributed Modular Ontologies, Extended Reasoning Algorithms for General Distributed Modular Ontologies, Distributed Modular Ontology Reasoning System, Case Study and Experiments, and Conclusion. Chapter 2 introduces background concepts on ontology, ontology reasoning, distributed ontology and interface-based modular ontology. Chapter 3 describes the distributed ontology reasoning algorithms for TBox reasoning and ABox reasoning of simple distributed modular ontologies, including reasoning for concept, negated concept, conjunction, disjunction, subsumption, role reasoning, and instant checking for queries. Chapter 4 describes the extended distributed ontology reasoning algorithms for TBox reasoning and ABox reasoning in general distributed modular ontologies, including reasoning algorithms with
simple queries in extended models, reasoning algorithms with compound queries. Chapter 5 presents the implementation of the distributed modular ontology reasoning system. Chapter 6 shows a case study, results of experiments, and evaluation. Chapter 7 gives conclusions and future work.
Chapter 2

Background

In this chapter, we will introduce background knowledge of this thesis, including ontology definition, ontology reasoning, interface-based modular ontology, and distributed ontology.

2.1 Ontology

Neches and colleagues defined the concept of ontology: “An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary” [9].
There are different categorizations of ontologies proposed by Mizoguchi and colleagues (1995), van Heijst and colleagues (1997), and Guarino (1998) and Lassila and McGuinness (2001) [10]. Lassila and McGuinness categorization presents different types of lightweight and heavyweight ontologies as shown in Figure 2.1 [10]. The figure starts from the lightest ontology called “Controlled vocabularies” and gradually increases ontologies’ complexity to “General Logical constraints”. In the left-half part, lightweight ontologies include catalogue, UML, ER diagrams, SQL and so on. In the right-half part, heavyweight ontologies belong to AI-based languages and ontology markup languages, which mainly have three kinds of components, called concepts, roles and individuals.

Figure 2.1: Lassila and McGuinness categorization

In more detail, only individuals are described with assertional knowledge in these three components. The remainder components are built with terminological knowledge, which consists of pre-existing terms with several constructors, like conjunction, disjunction, negation, existential restriction, etc [10]. In terms of constructors, different ontology languages usually
choose different combination of constructors for different requirements, which means that a choice of combination decides the description logic of a particular ontology language [10]. For example, an $\mathcal{AL}$ language combines atomic negation, concept intersection, universal restrictions, and limited existential quantification constructors.

In general, concepts in an ontology usually describe classes of objects, which could be primitive or defined. Roles in an ontology represent binary relations between concepts. And individuals in an ontology define instances of classes. Different ontology languages have different typing constraints and expressive powers. They all have their own class and property expressions. For example, $\textit{OWL 2 EL}$ is based on description logics $\mathcal{EL}^{++}$, which is powerful for reasoning with large terminologies [1]. $\textit{EL}_{\text{tiny}}$, as the simplest definition of $\textit{OWL 2 EL}$, only requires general concept inclusions, existential restriction, conjunction, top and bottom as shown in Figure 2.2 [16]. $IName$ of Figure 2.2 represents the sets of individual names, and $CName$ means class names. $PName$ represents property names. $C$ represents any axiom of a class, like $C$, $D$ or $E$.

\[
\begin{align*}
\text{Axiom} & \quad ::= \ C \subseteq C \mid C(\text{IName}) \mid P(\text{IName}, \text{IName}) \\
C & \quad ::= \ CName \mid \top \mid \bot \mid C \cap C \mid \exists P.C \\
P & \quad ::= \ PName
\end{align*}
\]

Figure 2.2: $\textit{EL}_{\text{tiny}}$ class and property expression
Ontology design can be classified as being done in a monolithic way or a modular way [13]. Figure 2.3 [31] shows three different architectures to employ ontologies, including monolithic ontologies, multiple ontologies, and hybrid ontologies.

As the original category of ontology methodologies, a monolithic ontology is created as a single piece of artifact (Figure 2.3a), which usually leads to high cohesion and high coupling [5]. This kind of ontology methodologies were used in many early but still popular ontology systems, such as On-To-Knowledge and METHONTOLOGY [25].

In multiple ontology methodologies, each of ontologies describes its own information source (Figure 2.3b). Meanwhile, each of source ontologies can be developed without respect to other sources or their ontologies [31]. This kind of ontologies has lower coupling than monolithic ontologies. Due to lack a shared vocabulary, the multiple ontology needs an inter-ontology mapping to compare differences from involved ontologies, such as semantically equal terms, different aggregation and granularity of concepts [31].

In terms of inter-ontology mapping, Wache and colleagues [31] introduced four general approaches including defined mappings, lexical relations, top-level grounding, and semantic correspondences. Defined mappings utilize special mediator agents to customizedly translate between
Figure 2.3: Three possible ways for employing ontologies
ontologies. Lexical relations generate a mapping by informal relations between ontologies, like disjoint and covering. In top-level grounding, top-level ontology requires all lower-level ontologies having straightforward relations to avoid loss, conflicts and ambiguities. Semantic correspondences can compute an indirect mapping between ontologies by a common vocabulary of concepts in different ontologies, like semantic labels [31].

Hybrid ontologies utilize multiple ontologies with a common vocabulary (Figure 2.3c). Different hybrid ontologies have different definitions of shared vocabularies and local ontologies. In some cases, the shared vocabulary contains basic terms that belong to the primitives, which are redefined in a local ontology. In some other cases, these basic terms might comprise a general ontology or a local ontology.

Modular ontologies have higher practicability to some extent. There are two complementary scenarios perceiving modular ontologies [28]. One is that modular ontologies consist of several small and specific ontologies and relations between them. These smaller ontologies are decomposed from large-scale ontologies, which keeps semantically equivalent from original ontologies to modular ontologies. The other is that modular ontologies can be composed of a set of ontologies within a coherent network.

In [24], an ontology module is defined as: “An ontology module is a reusable component of a larger or more complex ontology, which is self-contained but
bears a definite association to other ontology modules, including the original ontology”.

The relations between ontology modules are based on the languages and logics of modules. For example, when ontology modules are written in OWL 2, we can use `owl:versionInfo` or `owl:versionIRI` to describe ontology meta data, and `owl:imports`, `owl:incompatibleWith` or another properties to illustrate their relations. Through the meta data and assertions of modules, systems choose required modules, then determine the dependencies between the modules to generate modular ontologies [17]. MOVO [17] illustrates a way to manage modular ontologies, which deals with the complexity of dependencies and mandatory exclusions between modules.

2.2 Ontology Reasoning

Ontology reasoning is used to deduce implicitly represented knowledge from the knowledge that is explicitly contained in the knowledge base [2]. By this service, we can acquire a high quality ontology design and fulfil some requirements from users, such as answering queries, and integrating multiple ontologies. Particularly, according to different requirements, we can define different functional reasoners to help design ontologies as well. For instance, some ontologies could be required to keep meaningful, which means that all concepts or roles have instances [2]. For these ontologies, we can define
special functional reasoners to help such ontologies effectively satisfy this requirement.

Ontology reasoning mainly includes classification, instance relationships and consistency checking. For classification, it generates a distinct hierarchy by classifying concepts to sub-concepts or super-concepts [2]. These subsumption relationships are also utilized on properties to deduce implicit information from them.

The type of ontology reasoning, that is not related to individuals, belongs to terminological (TBox) formalisms. The TBox of an ontology is built through declarations that are described by general properties of concepts [2]. TBox reasoning is mainly utilized to check satisfiable or subsumption relations of concepts. Satisfiability checking is to determine whether a description is satisfiable or not. For example, when a description contains a clash with terminology of knowledge base, the description will be contradictory and unsatisfiable. Subsumption checking is to find out whether one description is more general than another one [2]. TBox reasoning is useful to manage concepts of a terminology to hierarchy formalism based on its definition [2]. In addition, equivalence and disjointness are also relations between concepts, which can be checked in TBox reasoning as well.

In order to acquire decision procedures of TBox reasoning, Baader, McGuinness, Nardi and Patel-Schneider introduced useful tableau-based
algorithms that could be employed to obtain results of sound and complete satisfiability [2]. In more detail, the tableau-based algorithms apply tableaux rules to an axiom for testing its satisfiability. Until no more rules apply, when the TBox dose not contain an obvious contradiction then this axiom is satisfiable [2]. As for contradiction, \(\neg C \cap C\) is an obvious contradiction, and leads to unsatisfiability in tableau-based algorithms. Some tableaux rule examples [14] are shown in Table 2.1.

Table 2.1: Tableaux rules for TBox reasoning

<table>
<thead>
<tr>
<th>Rule</th>
<th>Axiom 1</th>
<th>Axiom 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x : ({C_1 \cap C_2, \ldots}) (\rightarrow \sqcap)</td>
<td>({C_1 \cap C_2, \ldots})</td>
<td>x : ({\exists R.C, \ldots}) (\rightarrow \exists)</td>
</tr>
<tr>
<td>x : ({C_1 \sqcup C_2, \ldots}) (\rightarrow \sqcup)</td>
<td>(\exists R.C, \ldots)</td>
<td>x : ({\forall R.C, \ldots}) (\rightarrow \forall)</td>
</tr>
<tr>
<td>x : ({\forall R.C, \ldots}) (\rightarrow \forall)</td>
<td>(\exists R.C, \ldots)</td>
<td>x : ({\forall R.C, \ldots}) (\rightarrow \forall)</td>
</tr>
<tr>
<td>x : ({\exists R.C, \ldots}) (\rightarrow \exists)</td>
<td>(\forall R.C, \ldots)</td>
<td>x : ({\forall R.C, \ldots}) (\rightarrow \forall)</td>
</tr>
<tr>
<td>x : ({\forall R.C, \ldots}) (\rightarrow \forall)</td>
<td>(\exists R.C, \ldots)</td>
<td>x : ({\forall R.C, \ldots}) (\rightarrow \forall)</td>
</tr>
</tbody>
</table>

As shown in Table 2.1, the first line is the conjunction rule, which means if an axiom \(x\) contains \(\{C_1 \cap C_2, \ldots\}\), the new \(x\) will have \(\{C_1 \cap C_2, C_1, C_2, \ldots\}\) after an action of conjunction. The second line is the disjunction rule. When an axiom is \(\{C_1 \sqcup C_2\}\), the algorithm will return multiple models \(\{C_1 \sqcup C_2, C_1\}\) and \(\{C_1 \sqcup C_2, C_2\}\) after a disjunction action. The remainder rules are existential, forall and transitive rules. The tableaux-based algorithms apply these rules for axiom \(x\) to obtain new axiom \(y\) with its new contents.
The ABox of an ontology contains extensional knowledge about the domain of interest, which has two types of assertions, concept assertions and role assertions. ABox reasoning can find out whether assertions are consistent, and whether assertions entail a particular individual that belongs to an instance of a certain concept description (or membership consistency checking) [2].

After receiving satisfiable results from reasoning of terminology, the ABox can be filled with individual assertions [2]. According to this ABox definition, if ABox reasoning does not involve with TBox reasoning, the reasoning procedure will lose important information. For example, TBox $\mathcal{T}$ contains concept C that is disjointness with concept D, and ABox $\mathcal{A}$ contains the assertions C(c) and D(c). Together with the TBox, the assertions of this ABox should be inconsistent. But if the ABox reasoning procedure was not involved with TBox reasoning, then systems could only find out the assertions without inconsistent results. Therefore, the definition of ABox consistency is that ABox $\mathcal{A}$ will be consistent with respect to TBox $\mathcal{T}$, only if an interpretation belongs to a model of both $\mathcal{A}$ and $\mathcal{T}$ [2].

Membership consistency checking of ABox has the similar reasoning procedures. In order to check whether a set of concept assertions $M$ are consistent, systems can extend related concept assertions of original ABox $\mathcal{A}$ with the assertions $M$. Then, systems can continue the consistency
checking with the expanded ABox. If the expanded ABox is consistent, then the membership consistency checking will be consistent [2].

### 2.3 Interface-Based Modular Ontology

The Interface-Based modular ontology Formalism (IBF) defines a “modular ontology” with a set of “ontology modules” and “interfaces” [8]. Ontology modules are a sub-domain of general ontologies, and the modules could exist dependencies to each other [8]. The “interfaces” are utilized to model modules’ dependencies. The interfaces also belong to general ontologies with TBox knowledge.

An ontology module can be a realizer module or a utilizer module. Realizer modules realize knowledge from interfaces, and utilizer modules use knowledge from interfaces. A realizer module consists of TBox, ABox, and a “realizer” indicator. All knowledge of a realizer module is self-contained without dependence with others. It realizes its interfaces by extension interface knowledge, which means that information sources of the realizer module subsumes their realizing interfaces. A utilizer module consists of TBox, ABox, and a “utilizer” indicator. It reuses information sources from its utilizing interfaces. A utilizer module utilizes a part of knowledge from interfaces. The reused knowledge in a utilizer module is also defined and extended to the realizer modules that implement the interfaces.
An interface-based modular ontology consists of ontology modules, interfaces, and their relations as shown in Figure 2.4. An ontology module can realize and utilize multiple interfaces, which means one module could be a realizer and utilizer module at same time. In this model, different interfaces have different knowledge domains, and a realizer module could realize several different knowledge domains in its interfaces. The different interfaces could be utilized by one or more utilizer modules. In other words, a realizer module $M$ could be related to multiple modules that utilize knowledge from $M$ through its interfaces. And a utilizer module $N$ could utilize multiple interfaces that are realized by multiple realizer modules. All in all, there could be many-to-many relations between utilizer modules and realizer modules.

Figure 2.4: A model of interface-based modular ontology
Figure 2.5 shows a simple IBF modular ontology. The interface-based modular ontology “Publication Ontology” consists of three modules, three interfaces and their relations. In this ontology, module Publisher contains information of publishers defined by TBox and ABox, which provides public TBox knowledge to interfaces PrI and PaI. In other words, interfaces PrI and PaI are realized by module Publisher by extensions of definition and assertions in PrI and PaI. Module Author describes information of authors by reusing knowledge from PaI to utilize a part of public TBox knowledge from Publisher. Author realizes interface AuI and provides knowledge to module Reader. Meanwhile, Reader also reuses knowledge from Publisher through interface PrI.

![Figure 2.5: An IBF modular ontology](image-url)
ontology files [8]. I-OWL extends the OWL ontology language with two properties \textit{utilizeInterface} and \textit{realizeInterface} that are defined to indicate utilizing and realizing relations of ontology modules [8]. For instance, module \textit{Publisher} is defined as follows:

\begin{verbatim}
<owl:Ontology>
  <iowl:realizeInterface
    rdf:resource="&IBFModularOntology;PrI.owl"/>
  <iowl:realizeInterface
    rdf:resource="&IBFModularOntology;PaI.owl"/>
</owl:Ontology>
\end{verbatim}

As shown in the example, \textit{Publisher} realizes interfaces \textit{PrI} and \textit{PaI} indicated by \textit{realizeInterface} property. Module \textit{Author} utilizes interface \textit{PaI} by using \textit{utilizeInterface} property and realizes interface \textit{AuI} by \textit{realizeInterface} property:

\begin{verbatim}
<owl:Ontology>
  <iowl:utilizeInterface
    rdf:resource="&IBFModularOntology;PaI.owl"/>
  <iowl:realizeInterface
    rdf:resource="&IBFModularOntology;AuI.owl"/>
</owl:Ontology>
\end{verbatim}

Module \textit{Reader} utilizes interfaces \textit{PrI} and \textit{AuI} by \textit{utilizeInterface} property:

\begin{verbatim}
<owl:Ontology>
  <iowl:utilizeInterface
    rdf:resource="&IBFModularOntology;PrI.owl"/>
</owl:Ontology>
\end{verbatim}
<owl:utilizeInterface
rdf:resource="&IBFModularOntology;AuI.owl"/>
</owl:Ontology>

These two properties describe realizing and utilizing relations between ontology modules and interfaces. In addition, these properties also help indicate the reused knowledge defined in interfaces in utilizer modules. In the following, we introduce an example about reusing and realizing a concept from an interface. A concept Editor defined in PaI as:

<owl:Class rdf:about="#Editor"/>

As interfaces of interface-based modular ontologies only consist of TBox without ABox, there are only concepts and roles in interfaces. The realizer module Publisher realizes concept Editor as follows:

<owl:Class rdf:about="#Editor">
  <rdfs:subClassOf rdf:resource="#Employee"/>
</owl:Class>
<Publisher:Editor rdf:about="#MaryThompson"/>

As shown in the above, Publisher adds an instance MaryThompson to Editor and extends Editor concept with subClassOf property. Next, the utilizer module Author utilizes concept Editor from the interface PaI:

<owl:ObjectProperty rdf:about="#hasEditor">
  <rdfs:domain rdf:resource="#Author"/>
  <rdfs:range rdf:resource="&IBFModularOntology;PaI.owl#Editor"/>
</owl:ObjectProperty>
<Author:&IBFModularOntology;PaI.owl#Editor rdf:about="#BenThompson"/>
**Author** as a utilizer module uses *IBFModularOntology* to represent which concepts or roles are reused in *Author* and where they come from. In this example, we can find that the reused concept *Editor* is from PaI.owl by “IBFModularOntology;PaI.owl”. Similarly to realizer modules, reused knowledge defined in interfaces also can be utilized by TBox and ABox in utilizer modules. In this case, the reused concept *Editor* in *Author* module is defined as the range of *hasEditor* relation or role. Also, a new instance *BenThompson* of *Editor* is added in *Author*. Through the reused concept *Editor* of interface *PaI*, *Author* as a related utilizer module is connected with realizer *Publisher*.

An assembled modular IBF ontology needs to be validated before reasoning. Every utilizer modules should have its related interfaces and corresponded realizer modules in an IBF ontology. For example, if an IBF modular ontology only consists of one utilizer module in extreme case, this modular ontology will be not complete and invalid for reasoning.

In this thesis, we will design and implement the algorithms and system to reason about interface-based modular ontologies in a distributed environment. In the distributed environment, ontology modules and interfaces are located in different computing nodes without need for migration. The algorithms and system support reasoning local ontology modules and interfaces separately in their allocated local computing node and aggregate to overall reasoning results without monolithic reasoning.
2.4 Distributed Ontology

The distributed ontology is another approach to distributedly reasoning ontology [4]. The distributed ontology has its parts being stored on distributed sites, and being reasoned distributedly. In [4], a distributed ontology language called DDL (Distributed Description Language) is proposed. Mapping of DDL ontologies is to use “bridge rules” to connect local ontologies. The DDL mapping differs from traditional multi-ontology mappings. In DDL, the main correspondences are binary relations based on onto- and into-bridge rules.

Assume that we have two local ontologies $O_1$ and $O_2$ defined by DDL. $C$-objects in $O_1$ are written as $1 : C$. In the following, we will introduce the two types of bridge rules from [4]:

- into-bridge rule $1 : C \xrightarrow{\text{r}_{\text{Into}}} 2 : E$, indicating that $C$-objects in $O_1$ correspond only to $E$-objects in $O_2$
- onto-bridge rule $1 : C \xrightarrow{\text{r}_{\text{Into}}} 2 : E$, indicating that every object in $E$ of $O_2$ has a corresponding object in concept $C$ of $O_1$

The into-bridge rules state that every object or individual in concept $C$ has a corresponding object $r_{\text{Into}}(C)$ that is subsumed by concept $E$ in $O_2$ according to the $r_{\text{Into}}$ relation between $O_1$ and $O_2$. Only the part of objects in $E$ can
correspond to objects in concept $C$ of $O_1$. The onto-bridge rules state that every object in $C$ has a corresponding object $r_{\text{Onto}}(C)$ according to the $r_{\text{Onto}}$ relation between $O_1$ and $O_2$. In the onto-bridge rules, objects of $E$ in $O_2$ is subsumed by all corresponding objects $r_{\text{Onto}}(C)$.

For example, we have an into-bridge rule 1: $\text{CAT} \xrightarrow{\text{i}} 2 : \text{ANIMAL}$ in DDL. The into-bridge rule indicates that cats correspond only to animals. Not all animals are cats in this DDL so that onto-bridge rule 1: $\text{CAT} \xrightarrow{\text{onto}} 2 : \text{ANIMAL}$ cannot be included in this ontology.

A DDL ontology consists of a set of local ontologies with their bridge rules. We assume that two local ontologies $O_i$ and $O_j$, and bridge rules $\mathfrak{B}_{ij}$ form a DDL ontology. The $O_i$ ontology has concepts $A$, $B_1$ and $B_2$ and so on, and the $O_j$ ontology has concepts $G$, $M$, $H_1$, $H_2$, etc. The bridge rules $\mathfrak{B}_{ij}$ have one onto bridge rule $i : A \xrightarrow{\text{onto}} j : G$ and two into bridge rules $i : B_1 \xrightarrow{\text{onto}} j : H_1$, $i : B_2 \xrightarrow{\text{onto}} j : H_2$. In this example, any concept subsumes $A$ in $O_i$ that will have an onto-bridge rule with concept $G$ in $O_j$. Similarly, any concept is subsumed by $B_1$ in $O_i$ that will have an into-bridge rule with concept $H_1$ in $O_j$. Otherwise, If there are relations between $B_1$ and $A$ in $O_i$, then $H_1$ and $G$ in $O_j$ might imply new relations.

Generally, bridge rules could form chains of subsumptions in an atomic DDL ontology as shown in Figure 2.6 [4]. An onto-bridge rule can “switch up” from $O_2$ to $O_1$, and an into-bridge rule can switch down from $O_1$ to $O_2$. The
top part of the figure illustrates concepts of $O_1$, and the bottom part shows the concepts from $O_2$. For all concepts of the chain, its beginning has the smallest sub-set concepts in $O_2$. A right concept can subsume its left-hand concept on both top and bottom part in this figure. Therefore, bridge rule chains can deduce implicit knowledge from the chain. For example, in Figure 2.6 we can find an implicit bridge rule that $G$ is subsumed by $H$ in $O_2$ by the bridge rule chain.

![Figure 2.6: Intuitive shape of subsumption proofs](image)

For the above DDL ontology $⟨O_i, O_j, \mathfrak{B}_{ij}⟩$, Figure 2.7 [27] shows an important part of reasoning procedure with concept expression $L(m)$ where $L(m)$ is \{ $G \sqcap \neg (H_1 \sqcup H_2) \sqcap \ldots$ \}. The two functions $DT_{ab_i}$ and $DT_{ab_j}$ respectively provide the reasoning procedure of the local ontologies $O_i$ and $O_j$ of the DDL ontology.

$DT_{ab_j}$ starts reasoning from concept expression $L(m)$ or $D$ represented as node $x$. After a serials $\textit{SHIQ}-rules$ reasoning, the reasoning procedure needs to reason with $L(y)$. Its concepts $H$ and $G$ both belong to the results of the bridge rules $\mathfrak{B}_{ij}$. According to $\mathfrak{B}_{ij}$, when we combine bridge rules $i : B_{ij}$
Figure 2.7: Subsumption propagation

\[ j : H_1 \text{ and } i : B_2 \xrightarrow{\subseteq} j : H_2, \text{ it will return } i : \sqcup B \xrightarrow{\subseteq} j : \sqcup H. \]

With the addition of \( i : A \xrightarrow{\supseteq} j : G, \) we can get two unconnected lines about the subsumption chain, a switch up line from \( G \) to \( A \), and a switch down line from \( \sqcup B \) to \( \sqcup H \).

As the top part of the lines, \( A \) and \( \sqcup B \) are defined in \( O_i \), while bottom part \( G \) and \( \sqcup H \) are defined in \( O_j \).

If the reasoning algorithm can determine a new bridge rule that is \( A \subseteq \sqcup B \), then it can return that \( G \) is subsumed by \( \sqcup H \). Because, the bridge rule \( A \subseteq \sqcup B \) can connect the two lines to a subsumption chain that starts from \( j : G \) to \( i : A \) through an onto-bridge rule \( i : A \xrightarrow{\supseteq} j : G \). Then, the chain is from concept \( A \) and arrives \( B \) with the new bridge rule \( A \subseteq \sqcup B \). Next, starting from \( B \), the chain arrives \( \sqcup H \) with an into-bridge rule \( i : \sqcup B \xrightarrow{\subseteq} j : \sqcup H \). According to the subsumption chains, \( G \) is subsumed by \( \sqcup H \), and the reasoning will add \( \sqcup H \) to \( L(y) \). At this step, the algorithm will find a
clash between $\sqcup H$ and $\neg (\sqcup H)$. Therefore, it returns unsatisfiable of concept expression $D$ to system, hence the original $L(m)$ also is unsatisfiable.

In more detail, in order to reason about $A \sqsubseteq \sqcup B$, the algorithm invokes $DTab_i$ to test satisfiability of concept $A \sqcap \neg (\sqcup B)$ on $O_i$. When it returns unsatisfiable to $DTab_i$, the algorithm can clarify $A$ is subsumed by $\sqcup B$. Then the algorithm adds $\sqcup H$ to $L(y)$, and returns final result unsatisfiable. If $A \sqcap \neg (\sqcup B)$ is satisfiable, then this part of concept expression is satisfiable, and the algorithm will continue to reason with the concept expression.

Although both of the distributed IBF modular ontology and DDL use distributed reasoning procedures for user queries, they have different language constructs that lead to different distributed reasoning algorithms. Compared the language constructs, DDL uses bridge rules to connect its local ontologies. The local ontologies with bridge rules only form one DDL ontology. The distributed IBF modular ontology uses interfaces to describe dependencies between ontology modules. The ontology modules can be assembled to multiple distributed IBF modular ontologies. In the following, we will introduce distributed reasoning algorithms of the distributed IBF modular ontology.
Chapter 3

Reasoning Algorithms for Simple Distributed Modular Ontologies

In this chapter, we will introduce reasoning algorithm designs for simple distributed modular ontologies, including TBox reasoning for concept, negation, disjunction, conjunction, subsumption, and role reasoning, and ABox reasoning for instance checking with concept, negation, disjunction, conjunction, and role queries.
3.1 Simple Distributed Modular Ontology Model

In this chapter, we first consider a simple model of distributed modular ontologies. We consider that an assembled modular ontology consists of utilizer modules, realizer modules, interfaces, and non-utilizer/realizer modules or “lone modules”. In addition, we assume that any module of an assembled modular ontology cannot be both a utilizer module and a realizer module. Each realizer module only realizes one interface, and one interface can only be realized by one realizer module. Each utilizer module only reuses knowledge from one interface, and one interface can be reused by one utilizer module only. In addition, knowledge is only defined in modules, and knowledge bases of interfaces are sub-knowledge bases of their realizer modules. Figure 3.1 shows a simple assembled ontology with five modules and two interfaces.

The assembled modular ontology in Figure 3.1 has two realizer modules $\text{Module}_4$ and $\text{Module}_5$ to provide knowledge to the two utilizer modules $\text{Module}_1$ and $\text{Module}_3$ through $\text{Interface}_1$ and $\text{Interface}_2$. $\text{Module}_2$ is a non-realizer/utilizer module. This “lone module” is independent of other modules and all its knowledge is self-contained, and not used by other modules.
In terms of distributed modular ontology, we assume that each module in an assembled modular ontology is allocated on a different computing node. When we reason with a distributed assembled modular ontology using our distributed reasoning system, the system needs to reason about each module separately at its local node, as shown in Figure 3.2. Each local node only stores one module with its related interface. If a module is a “lone module”, its local node will not contain any interface. Meanwhile, there is no connection on a lone node to represent realizing and utilizing relations between modules through interfaces.

A realizer node contains a realizer module and its realized interface. A utilizer node contains a utilizer module and its utilized interface. When a utilizer module reuses knowledge from a realizer module through an interface, there
will be a “connection” between these two local nodes. Lone nodes could not have “connections” with other nodes, because no utilizer modules reuse their knowledge in the assembled modular ontology.

When the system receives a query to a distributed assembled modular ontology, it will visit local nodes which contain related modules to compute the query result. For example, a user may issue a query to the system involving three modules of the ontology shown in Figure 3.2. The system will distributedly invoke local reasoning on the three nodes, and process the local reasoning results using the reasoning algorithms. Finally, the system will return processed final results to the user. In the following, we will describe distributed reasoning algorithms for various types of queries.
3.2 Searching Sub- and Super-Concepts

As fundamental algorithms of the distributed reasoning, the searching sub- and super-concept algorithms aim to compute all sub- or super-concepts of a given concept in its related interface, utilizer and realizer modules. For example, if a concept is defined in a realizer module, the searching sub-concept algorithm will find all its sub-concepts in this realizer module, its related interface and utilizer module.

In order to find out all sub-concepts of a given concept, the `SearchingSubConcepts` algorithm needs to reason about each of the two related modules on their local nodes, starting from the module and interface that defines the given concept. In this step, the algorithm computes two sets of sub-concepts in the interface and module. Then, using the computed sub-concepts of the interface, the algorithm continues to reason about the other module to find out new sub-concepts of the given concept.

As shown in Algorithm 1, the `SearchingSubConcepts` algorithm firstly invokes functions on the local node that stores module $M(A)$ where $M(A)$ means that module $M$ containing concept $A$, in step 1 to compute all sub-concepts $J$ of $A$ in $M(A)$ and interface $I$. Next, the algorithm separates the elements of $J$ to $K$ and $H$, where elements of $K$ and $H$ are respectively defined in $I$ and $M(A)$. In step 2, using interface set $K$, the algorithm invokes functions on the other local node with module $M'$ to
Algorithm 1 SearchingSubConcepts

A is a given concept that is defined in realizer or utilizor module $M(A)$. $M'$ is the utilizor or realizer module where $M(A)$ and $M'$ are connected by interface $I$. Reasoning Order: reasoning about $M(A)$ with $I$ first, then reasoning about $M'$ with $I$.

**Step 1:** reasoning on the local node with $M(A)$

**Input:** module $M(A)$, interface $I$, concept $A$

**Output:** sub-concept sets $H$ and $K$

1: $J \leftarrow$ reasoning about $M(A)$ and $I$ to find out sub-concepts of $A$
2: for $j_i$ in $J$ do
3: \hspace{1em} $K \leftarrow$ adding $j_i$ when $I$ has this concept
4: \hspace{1em} $H \leftarrow$ adding $j_i$ when $M(A)$ has this concept
5: end for

**Step 2:** reasoning on the local node with $M'$

**Input:** module $M'$, interface $I$ and $K$

**Output:** sub-concept set $L$

1: for $k_i$ in $K$ do
2: \hspace{1em} $L \leftarrow$ reasoning about $M'$ and $I$ to find out sub-concepts of $k_i$, and only put resulting sub-concepts that are defined in $M'$ in $L$
3: end for
identify sub-concepts in $M'$ of each element in $K$, and puts those concepts defined in $M'$ to set $L$. Now, the algorithm respectively identifies all sub-concept sets $K$, $H$ and $L$ of $A$ in the interface and two modules $I$, $M(A)$ and $M'$. Figure 3.3 shows the final output data of the $SearchingSubConcepts$ algorithm.

![Diagram](image)

Figure 3.3: The output sets of the $SearchingSubConcepts$ algorithm

As shown in Figure 3.3, the final output data is sub-concepts sets $H$ in $M(A)$, $K$ in $I$, and $L$ in $M'$. After searching the two modules, the algorithm returns these three sub-concepts $H$, $K$, and $L$ to the system.

The $SearchingSuperConcepts$ algorithm aims to compute all super-concepts of a given concept. The algorithm has a similar structure to the $SearchingSubConcepts$ algorithm, which also needs to sequentially reason with each of the two related modules on their local nodes to find out super-concept sets of the given concept.

As shown in Algorithm 2, similarly to the $SearchingSubConcepts$ algorithm, the algorithm invokes functions on the local node that stores module $M(A)$ to find out super-concept sets $K$ and $H$ of $A$ in $I$ and $M(A)$ in step 1. Then,
Algorithm 2 SearchingSuperConcepts

A is a given concept that is defined in module $M(A)$. $M'$ is the utilizer or realizer module where $M(A)$ and $M'$ are connected by interface $I$. Reasoning Order: reasoning about $M(A)$ with $I$ first, then reasoning about $M'$ with $I$.

**Step 1:** reasoning on the local node stored $M(A)$

**Input:** module $M(A)$, interface $I$, concept $A$

**Output:** super-concept sets $H$ and $K$

1: $J$ ← reasoning about $M(A)$ and $I$ to find out super-concepts of $A$
2: for $j$ in $J$ do
3: $K$ ← adding $j$ when $I$ has this concept
4: $H$ ← adding $j$ when $M(A)$ has this concept
5: end for

**Step 2:** reasoning on the local node stored $M'$

**Input:** module $M'$, interface $I$ and $K$

**Output:** super-concept set $L$

1: for $k$ in $K$ do
2: $L$ ← reasoning about $M'$ and $I$ to find out super-concepts of $k$, and only put resulting super-concepts that are defined in $M'$ in $L$. 
3: end for

in step 2, using the interface’s super-concept set $K$, the algorithm invokes functions of the other local node to find out new super-concepts $L$ in the related module $M'$. Similarly to the SearchingSubConcepts algorithm, the algorithm has final output $K$ in $I$, $H$ in $M(A)$, and $L$ in $M'$.

### 3.3 TBox Reasoning with Single Concept

TBox reasoning with single concept concerns with concept reasoning, negation reasoning and their reasoning algorithms in an assembled modular ontology.
3.3.1 Concept Reasoning

The concept reasoning aims to determine whether a given concept is satisfiable in an assembled modular ontology.

In this thesis, we assume that only if a modular ontology is consistent, the algorithm can reason about a query with this assembled modular ontology. In other words, when the system reasons with a given concept in its defined module and related interface, its reasoning result will be same as reasoning with the concept in the corresponding monolithic ontology by merging all modules of the assembled modular ontology into a monolithic ontology. Thus, the ConceptReasoning algorithm can check satisfiability of a given concept on one local node to return its final reasoning result.

Algorithm 3 ConceptReasoning

\[ A \] is a given concept that is defined in module \( M(A) \), and \( I \) is its related interface for concept reasoning.

**Input:** module \( M(A) \), interface \( I \), concept \( A \)

**Output:** satisfiable or unsatisfiable

1: \( R \leftarrow \) checking satisfiability of \( A \) in \( M(A) \) and \( I \) using a standard reasoning algorithm
2: **return** \( R \)

As shown in **Algorithm 3**, the ConceptReasoning algorithm invokes functions on the local node that stores module \( M(A) \), and uses a standard reasoning algorithm to check satisfiability of \( A \) in \( M(A) \) and \( I \). After that, as the final result, the reasoning result from the standard reasoning algorithm will be returned to the system. As the overall modular ontology
with all modules are assembled consistent, we do not need to check the other related modules.

### 3.3.2 Negation Reasoning

The negation reasoning aims to determine whether a negated concept is satisfiable in an assembled modular ontology. For the negation reasoning, a negated concept \( \neg A \) can be added as a new concept into the module that contains concept \( A \), and use the concept reasoning algorithm to reason. For example, if the negation reasoning algorithm needs to reason about \( \neg A \), we can add \( C = \neg A \) to \( M(A) \). Then, the negation reasoning can utilize the \textit{ConceptReasoning} algorithm to reason about \( C \) and return the result to the system.

### 3.4 TBox Reasoning with Two Concepts

TBox reasoning with two concepts introduces distributed reasoning on two nodes. In this thesis, a distributed query means that concepts involved in the query are defined in different modules. Thus, the query must be distributedly reasoned in multiple modules of an assembled modular ontology to compute final results.
This section contains disjunction reasoning, conjunction reasoning, subsumption reasoning, and role reasoning of distributed queries.

### 3.4.1 Disjunction Reasoning

The disjunction reasoning aims to determine whether a disjunctive query is satisfiable in an assembled modular ontology. The disjunctive query consists of two concepts with a disjunction constructor, like $A \sqcup B$. In distributed reasoning, these two concepts are respectively defined in a realizer module and a utilizer module. In addition, we only concern the case that their related interface does not contain the concepts in the query.

The disjunction reasoning algorithm is based on the $\text{ConceptReasoning}$ algorithm. Using computed results of the $\text{ConceptReasoning}$ algorithm, when the disjunction reasoning algorithm $\text{DisjunctionReasoning}$ finds out both of the two concepts in the disjunctive query are unsatisfiable, then this disjunctive query will be unsatisfiable. If not, the query will be satisfiable.

As shown in Algorithm 4, the $\text{DisjunctionReasoning}$ algorithm firstly invokes functions on the node that stores module $M(A)$ in step 1, which computes satisfiability of $A$ by the $\text{ConceptReasoning}$ algorithm. If the result is satisfiable, the algorithm will directly return satisfiable as a final result. If not, the algorithm invokes functions on the other node that stores module $M(B)$. Then, using the $\text{ConceptReasoning}$ algorithm, the
Algorithm 4 DisjunctionReasoning

For query $A \sqcup B$, $A$ and $B$ are respectively defined in two modules $M(A)$ and $M(B)$, and their related interface is $I$.

**Step 1:** reasoning on the local node stored $M(A)$

**Input:** module $M(A)$, interface $I$, concept $A$

**Output:** satisfiable if $A$ is satisfiable or no output

1: $R \leftarrow$ checking satisfiability of $A$ in $M(A)$ by ConceptReasoning
2: if $R$ is satisfiable then
3:     return satisfiable
4: end if

**Step 2:** reasoning on the local node stored $M(B)$

**Input:** module $M(B)$, interface $I$, concept $B$

**Output:** satisfiable or unsatisfiable

1: $T \leftarrow$ checking satisfiability of $B$ in $M(B)$ by ConceptReasoning
2: if $T$ is satisfiable then
3:     return satisfiable
4: else
5:     return unsatisfiable
6: end if

DisjunctionReasoning algorithm gets the reasoning result of $M(B)$ in step 2, and returns this result as the final reasoning result to the system.

### 3.4.2 Conjunction Reasoning

The conjunction reasoning aims to determine whether a conjunctive query is satisfiable in an assembled modular ontology. The conjunctive query consists of two concepts with a conjunction constructor, like $A \sqcap B$. Similarly, these two concepts also are respectively defined in a realizer module and a utilizer module. Their related interface does not contain concepts in the query.
Similarly to the disjunction reasoning, the conjunction reasoning algorithm is also based on the *ConceptReasoning* algorithm. Comparing these two reasoning procedures, an important difference is that the conjunction reasoning needs to consider negated relations of the two concepts in a conjunctive query, as explained below.

If there is a negated relation between two concepts of a conjunctive query, the reasoning result will be empty and unsatisfiable. For a distributed conjunctive query, the conjunction reasoning needs to find out connections between the two concepts to determine whether there exists a negated relation in the two related modules. In other words, using the interface connection of the two related modules, the conjunction reasoning can check whether the distributed conjunctive query is satisfiable in a distributed modular ontology.

In more detail, when the *ConjunctionReasoning* algorithm reasons about a distributed conjunctive query $A \cap B$, the algorithm needs to check all satisfiability of newly computed conjunction queries that consist of $B$ and all super-concepts of $A$ involved in the interface. The system will reason with the interface and the module containing $B$ to return a set of satisfiability results for the super-concepts. After that, if there is one unsatisfiable result, the final reasoning result of $A \cap B$ will be unsatisfiable. If the super-concepts of $A$ conjunction with $B$ has one unsatisfiable result, then $A \cap B$ must be unsatisfiable. If all of the reasoning results are
satisfiable, the concepts $A$ and $B$ in the query will not have negated relations. The system can continue to check whether $A$ and $B$ respectively are satisfiable. Only if both of them are satisfiable, $A \cap B$ will be satisfiable.

Algorithm 5  ConjunctionReasoning

For query $A \cap B$, $A$ and $B$ are respectively defined in two modules $M(A)$ and $M(B)$, and their related interface is $I$.

**Step 1:** reasoning with $M(A)$

**Input:** module $M(A)$, interface $I$, concept $A$

**Output:** $H$ and $R$

1: $H \leftarrow$ adding super-concepts of $A$ in $I$ by SearchingSuperConcepts
2: $R \leftarrow$ checking satisfiability of $A$ by ConceptReasoning

**Step 2:** reasoning with $M(B)$

**Input:** module $M(B)$, interface $I$, concept $B$, $H$

**Output:** $J$ and $T$

1: for $h_i$ in $H$ do
2: \hspace{1em} $J \leftarrow$ reasoning about $h_i \cap B$ in $M(B)$ and $I$ by a standard reasoning algorithm
3: \hspace{1em} end for
4: $T \leftarrow$ checking satisfiability of $B$ by ConceptReasoning

**Step 3:**

**Input:** $R$, $T$, $J$

**Output:** *satisfiable* or *unsatisfiable*

1: for $j_i$ in $J$ do
2: \hspace{1em} if $j_i$ is unsatisfiable then
3: \hspace{2em} return unsatisfiable
4: \hspace{1em} end if
5: \hspace{1em} end for
6: if $R$ and $T$ both are satisfiable then
7: \hspace{1em} return satisfiable
8: \hspace{1em} else
9: \hspace{2em} return unsatisfiable
10: \hspace{1em} end if

As shown in Algorithm 5, the *ConjunctionReasoning* algorithm firstly invokes functions on the local node that stores module $M(A)$ in step 1 to
compute super-concepts $H$ of $A$ in $I$ by the $SearchingSuperConcepts$ algorithm, and check satisfiability of $A$ by the $ConceptReasoning$ algorithm. Next, using the computed $H$, the algorithm invokes functions on the other local node containing $B$ to check satisfiability of each concept in $H$ conjunction with $B$ in step 2, and records the results into $J$. In this step the algorithm also checks satisfiability of $B$ in $M(B)$ by the $ConceptReasoning$ algorithm.

Finally, in step 3, the algorithm finds out whether $J$ has an unsatisfiable result. If it has one unsatisfiable result, the final reasoning result will be unsatisfiable. If all of them are satisfiable, the $ConjunctionReasoning$ algorithm will continue to check computed satisfiable results $R$ and $T$. Only if both of them are satisfiable, the final conjunction reasoning result will be satisfiable. Otherwise, the algorithm will return unsatisfiable to the system.

### 3.4.3 Subsumption Reasoning

The subsumption reasoning aims to determine whether a subsumption query is satisfiable in an assembled modular ontology. A subsumption query consists of two concepts and a subsumed relation, like $A \sqsubseteq B$.

The subsumption query $A \sqsubseteq B$ can use $A \sqcap \neg B$ to compute reasoning results, which means that the subsumption reasoning can be based on the conjunction reasoning. For example, in subsumption reasoning, we can add
a new concept $C = \neg B$ into $M(B)$, and check satisfiability of $A \cap C$ using the $ConjunctionReasoning$ algorithm to return its reasoning results.

### 3.4.4 Role Reasoning

The role reasoning aims to determine whether a role query is satisfiable in an assembled modular ontology. A role query can be written as $A \sqsubseteq \forall Ro.B$, which consists of two concepts $A$ and $B$, and a role or property $Ro$. For a distributed role query, we assume that member concepts of a role query are defined in different modules. We also assume that $Ro$ and $A$ are defined in one module, and $B$ is defined in the other module. Both $A$ and $B$ are not in the interface.

In this situation, the $RoleReasoning$ algorithm needs to compute a set of related role queries based on the original query. By checking satisfiability of the computed queries, the algorithm can compute satisfiability of the original role query.

In order to compute these related role queries, for the original query $A \sqsubseteq \forall Ro.B$, the $RoleReasoning$ algorithm firstly finds out super-concepts of $B$ in its interface. Then, the algorithm uses each of these super-concepts to replace $B$ of the original query to create a set of related role queries local to $M(A)$. Next, the $RoleReasoning$ algorithm reasons with these created role queries on one $M(A)$’s node using a standard reasoning algorithm to
compute reasoning results. Based on these reasoning results, the algorithm computes the final reasoning result and returns it to the system. For example, if the RoleReasoning algorithm can find out one unsatisfiable query of the created role queries, the original role query will be unsatisfiable. If all of them are satisfiable, the original role query will be satisfiable.

Algorithm 6 RoleReasoning
For query $A \sqsubseteq \forall Ro.B$, Ro and A are defined in module $M(Ro, A)$, B is defined in module $M(B)$, and their related interface is $I$.

**Step 1:** reasoning about $M(B)$
**Input:** $M(B)$, interface $I$, concept $B$
**Output:** $J$

1: $J \leftarrow$ adding super-concepts of $B$ in $I$ by SearchingSuperConcepts

**Step 2:** reasoning about $M(Ro, A)$
**Input:** module $M(Ro, A)$, interface $I$, concept $A$, role $Ro$, $J$
**Output:** $L$

1: for $j_i$ in $J$ do
2:   $L \leftarrow$ reasoning about $A \sqsubseteq \forall Ro.j_i$ in $M(Ro, A)$ and $I$ by a standard reasoning algorithm
3: end for

**Step 3:**
**Input:** $L$
**Output:** *satisfiable or unsatisfiable*

1: for $l_i$ in $L$ do
2:   if $l_i$ is unsatisfiable then
3:     return unsatisfiable
4: end if
5: end for
6: return satisfiable

As shown in Algorithm 6, the RoleReasoning algorithm firstly invokes functions on the local node that stores module $M(B)$ in step 1, which computes super-concepts $J$ of $B$ in $I$ by the SearchingSuperConcepts
algorithm. Next, in step 2, using computed $J$, the algorithm invokes functions of the other local node containing $M(R_0, A)$ to check satisfiability of new queries using the concepts in $J$. In step 3, the algorithm finds out unsatisfiable results in computed data of step 2. When there is one unsatisfiable result, the algorithm will return unsatisfiable as a final reasoning result. When all of them are satisfiable, then the final result will be satisfiable.

### 3.5 ABox Reasoning

In this thesis, ABox reasoning concerns instance checking with single concept and instance checking with constructs involving two concepts, including instance checking for concept, negated concept, disjunction, conjunction and role query. In the following, we will introduce their reasoning algorithms in an assembled modular ontology.

#### 3.5.1 Instance Checking with Single Concept

Instance checking with single concept concerns instance checking for concept and for negated concept algorithms.
The instance checking for concept contains open-world reasoning and closed-world reasoning. The open-world reasoning aims to determine whether an instance concept query is consistent in an assembled modular ontology. The closed-world reasoning aims to compute whether an assembled modular ontology contains an instance concept query. An instance concept query can be written as $A(a)$, which consists of concept $A$ and individual $a$.

In distributed queries, $A$ and $a$ may be in different modules, which means that the system needs to check all related modules to compute $A(a)$.

In this situation, the open-world $\text{OpenInstanceConceptChecking}$ algorithm firstly checks satisfiability of $A$ by $\text{ConceptReasoning}$. If the reasoning result is unsatisfiable, the $\text{OpenInstanceConceptChecking}$ algorithm will directly return inconsistent to the system. Otherwise, the algorithm will continue to find out super-concepts of $A$ in the interface of $M(A)$ by $\text{SearchingSuperConcepts}$ algorithm. Using the computed super-concepts, the algorithm checks their consistency with $a$ in the interface and the module that contains $a$. If the reasoning results have one inconsistency, the final result will be inconsistent. If all results are consistent, the algorithm will return consistent to the system.

As shown in Algorithm 7, the $\text{OpenInstanceConceptChecking}$ algorithm firstly invokes functions on the local node that stores module $M(A)$ in step 1, which checks satisfiability of $A$, and computes super-concepts $J$ of $A$ in $I$. Next, in step 2, using computed $J$, the algorithm invokes functions of the
Algorithm 7 OpenInstanceConceptChecking
For query \( A(a) \), \( A \) is defined in module \( M(A) \), \( a \) is defined in the other module \( M(a) \), and their related interface is \( I \).

Step 1: reasoning with \( M(A) \)
\begin{itemize}
  \item \textbf{Input}: \( M(A) \), interface \( I \), concept \( A \)
  \item \textbf{Output}: \( J \), inconsistent if \( A \) is unsatisfiable
\end{itemize}
\begin{enumerate}
  \item \( R \leftarrow \) checking satisfiability of \( A \) by \texttt{ConceptReasoning}
  \item \textbf{if} \( R \) is unsatisfiable \textbf{then}
  \item \textbf{return} inconsistent
  \item \textbf{end if}
  \item \( J \leftarrow \) adding super-concepts of \( A \) in \( I \) by \texttt{SearchingSuperConcepts}
\end{enumerate}

Step 2: reasoning about \( M(a) \)
\begin{itemize}
  \item \textbf{Input}: module \( M(a) \), interface \( I \), \( J \)
  \item \textbf{Output}: \( L \)
\end{itemize}
\begin{enumerate}
  \item \textbf{for} \( J_i \) in \( J \) \textbf{do}
  \item \( L \leftarrow \) reasoning about \( J_i(a) \) in \( M(a) \) and \( I \)
  \item \textbf{end for}
\end{enumerate}

Step 3:
\begin{itemize}
  \item \textbf{Input}: \( L \)
  \item \textbf{Output}: \textit{consistent or inconsistent}
\end{itemize}
\begin{enumerate}
  \item \textbf{for} \( l_i \) in \( L \) \textbf{do}
  \item \textbf{if} \( l_i \) is inconsistent \textbf{then}
  \item \textbf{return} inconsistent
  \item \textbf{end if}
  \item \textbf{end for}
  \item \textbf{return} consistent
\end{enumerate}
other local node containing $a$ to check consistency of $J$ with $a$. In step 3, when there is one inconsistent result after reasoning in step 2, the algorithm will return inconsistent as a final reasoning result. When all of them are consistent, then the final result will be consistent.

The closed-world $\text{ClosedInstanceConceptChecking}$ algorithm also firstly checks satisfiability of $A$ by $\text{ConceptReasoning}$. If the reasoning result is unsatisfiable, the $\text{ClosedInstanceConceptChecking}$ algorithm will directly return inconsistent to the system. Otherwise, the algorithm will continue to find out sub-concepts of $A$ in the interface of $M(A)$ by $\text{SearchingSubConcepts}$ algorithm. Using the computed sub-concepts, the algorithm checks whether $a$ is an instance of the computed sub-concepts in the ABox of the module that contains $a$. If it does not find that $a$ is an instance of the computed sub-concepts, the final result will be inconsistent. Otherwise, the algorithm will return consistent to the system.

As shown in Algorithm 8, the $\text{ClosedInstanceConceptChecking}$ algorithm firstly invokes functions on the local node that stores module $M(A)$ in step 1, which checks satisfiability of $A$, and computes sub-concepts $J$ of $A$ in $I$. Next, in step 2, using computed $J$, the algorithm invokes functions of the other local node containing $a$ to check whether $a$ is an instance of $J$. In this step, if $J_i(a)$ is found in $M(a)$ and $I$, the algorithm will add consistent to $L$. Otherwise, the algorithm will add inconsistent. In step 3, when there is one consistent result after reasoning in step 2, the algorithm will return
Algorithm 8 ClosedInstanceConceptChecking

For query \( A(a) \), \( A \) is defined in module \( M(A) \), \( a \) is defined in the other module \( M(a) \), and their related interface is \( I \).

**Step 1:** reasoning about \( M(A) \)

**Input:** \( M(A) \), interface \( I \), concept \( A \)

**Output:** \( J \), inconsistent if \( A \) is unsatisfiable

1: \( R \leftarrow \) checking satisfiability of \( A \) by ConceptReasoning
2: if \( R \) is unsatisfiable then
3: \hspace{1em} return inconsistent
4: end if
5: \( J \leftarrow \) adding sub-concepts of \( A \) in \( I \) by SearchingSubConcepts

**Step 2:** reasoning about \( M(a) \)

**Input:** module \( M(a) \), interface \( I \), \( J \)

**Output:** \( L \)

1: for \( J_i \) in \( J \) do
2: \hspace{1em} \( L \leftarrow \) adding the result of finding \( J_i(a) \) in \( M(a) \) and \( I \)
3: end for

**Step 3:**

**Input:** \( L \)

**Output:** consistent or inconsistent

1: for \( l_i \) in \( L \) do
2: \hspace{1em} if \( l_i \) is consistent then
3: \hspace{2em} return consistent
4: \hspace{1em} end if
5: end for
6: return inconsistent
consistent as a final reasoning result. When all of them are inconsistent, then the final result will be inconsistent.

The instance checking for negated concept includes open-world reasoning and closed-world reasoning based on the OpenInstanceConceptChecking and ClosedInstanceConceptChecking algorithms. Similarly to the negation reasoning, the instance checking for negated concept algorithms are same as the instance concept checking algorithms by adding new concept $C = \neg A$ into module $M(A)$. Then, $C(a)$ are reasoned using the two instance concept checking algorithms to return final results.

### 3.5.2 Instance Checking with Constructs and Two Concepts

The instance checking with constructs and two concepts concerns instance checking for disjunction, conjunction and role queries. The reasoning algorithms are based on the instance concept checking algorithms.

The instance checking for disjunctive query, like $A \sqcup B(a)$, includes open-world reasoning and closed-world reasoning. The two reasoning algorithms check $A(a)$ and $B(a)$ by the corresponded instance concept checking algorithms. The open-world instance checking for disjunctive query is based on the OpenInstanceConceptChecking algorithm, and the
closed-world instance checking for disjunctive query is based on the *ClosedInstanceConceptChecking* algorithm. For the open-world reasoning, after checking $A(a)$ and $B(a)$, if they have one consistent result, the final reasoning result will be consistent. If all checking results are inconsistent, the final reasoning result will be inconsistent. After checking $A(a)$ and $B(a)$, the closed-world reasoning algorithm is same as the open-world reasoning algorithm.

The instance checking for conjunctive query, like $A \cap B(a)$, also includes open-world reasoning and closed-world reasoning. Both of them check satisfiability of $A \cap B$ by the *ConjunctionReasoning* algorithm. If the reasoning result is unsatisfiable, the system will stop reasoning and return inconsistent. Otherwise, the system respectively checks $A(a)$ and $B(a)$ by the instance concept checking algorithms similarly to the instance checking for disjunctive query. After that, for open-world reasoning, if one of $A(a)$ and $B(a)$ is inconsistent, the final result will be inconsistent. If both of them are consistent, the final reasoning result will be consistent. After checking $A(a)$ and $B(a)$, the closed-world reasoning algorithm is same as the open-world reasoning algorithm.

For the instance checking for role query, like $R(a, b)$, we assume that $R$ and $a$ are in module $M(R, a)$, and $R$ also is in its interface $I$. The instance $b$ is defined in the other module $M(b)$. The checking algorithm separately checks whether $a$ with $R$ in $M(R, a)$ and $I$ is consistent, and $b$ with $R$ in
$M(b)$ and $I$ is consistent using a standard reasoning algorithm. If any of results is inconsistent, the final result will be inconsistent. If both of them are consistent, the final reasoning result will be consistent.

For interface-involved query reasoning, we only consider the queries with constructs and two concepts that are respectively defined in a utilizer module and its interface. Its reasoning algorithms are simpler than the above reasoning algorithms. The interface-involved query algorithms can directly check satisfiability or consistency of the queries in the utilizer module and its interface using a standard reasoning algorithm to compute the final reasoning results.

### 3.6 Summary

In this chapter, we described the distributed reasoning algorithms for simple distributed modular ontologies. After introducing a simple distributed modular model and its features, we showed the reasoning algorithms for TBox and ABox queries to a modular ontology, including the concept, negation, disjunction, conjunction, subsumption, and role query reasoning.
Chapter 4

Extended Reasoning

Algorithms for General Distributed Modular Ontologies

In this chapter, we will introduce extended reasoning algorithms for general distributed modular ontologies from extended models reasoning and compound query reasoning.
4.1 Extended Models and Reasoning

Extended models and reasoning concern the multiple utilizer model, multiple realizer model, realizer-utilizer model and general model with their reasoning. These models extend the simple distributed modular ontology of chapter 3. In this section, we will introduce these different extended models and make comparisons of their reasoning procedures with some algorithms of the simple model in chapter 3, including the concept reasoning, negation reasoning, disjunction reasoning, conjunction reasoning, subsumption reasoning, role reasoning, and instance concept checking algorithms.

4.1.1 Multiple Utilizer Model and Reasoning

This section introduces the multiple utilizer model and its query reasoning. The query reasoning includes using same reasoning algorithms for the simple model, and using extended reasoning algorithms.

4.1.1.1 Multiple Utilizer Model

The multiple utilizer model means that its realizer module is utilized by multiple utilizer modules through an interface. As shown in Figure 4.1, we illustrate a multiple utilizer model that consists of three modules and one interface.
As shown in Figure 4.1, the realizer module $M_{R1}$ or $M_R$ provides knowledge to two utilizer modules $M_{U2}$ or $M_{U1}$, and $M_{U3}$ or $M_{U2}$ through interface $Interface_1$ or $I$.

4.1.1.2 Query Reasoning Using Same Reasoning Algorithms for the Simple Model

In the multiple utilizer model, the single concept, and disjunction reasoning algorithms are similar to the algorithms of the simple model. For the other distributed query reasoning involved two concepts $A$ and $B$, when any concept of $A$ and $B$ is defined in realizer module $M_R$, the reasoning algorithms are same as the simple model. In the following, we illustrate these query reasoning algorithms in Table 4.1.
Table 4.1: Query reasoning using same reasoning algorithms of the simple model

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>A or ( \neg A )</td>
<td>( M_U 1(A) ), ( M_U 2(A) ), or ( M_R(A) )</td>
<td>ConceptReasoning</td>
</tr>
<tr>
<td>( A \sqcap B )</td>
<td>((M_U 1(A), M_R(B)), (M_U 2(A), M_R(B))), or ((M_U 1(A), M_U 2(B)))</td>
<td>DisjunctionReasoning</td>
</tr>
<tr>
<td>( A \sqcap \neg B ) or ( A \sqsubseteq B )</td>
<td>((M_U 1(A), M_R(B))), or ((M_U 2(A), M_R(B)))</td>
<td>ConjunctionReasoning</td>
</tr>
<tr>
<td>( A \sqsubseteq \forall R \cdot B )</td>
<td>((M_U 1(A), M_R(B))), or ((M_U 2(A), M_R(B)))</td>
<td>RoleReasoning</td>
</tr>
</tbody>
</table>

Similarly to the simple model in chapter 3, the single concept reasoning, including the concept and negation reasoning, also involves in the module containing the single concept. As shown in Table 4.1, the single concept reasoning algorithms only involve in one module \( M_U 1(A) \), \( M_U 2(A) \), or \( M_R(A) \). The algorithms use the ConceptReasoning algorithm to compute final reasoning results.

The disjunction reasoning algorithm is based on the concept reasoning algorithm so that it is same as the DisjunctionReasoning algorithm in the simple model. For the conjunction, subsumption, and role query that their two concepts \( A \) and \( B \) are in utilizer modules \( M_U 1(A) \) or \( M_U 2(A) \), and in realizer module \( M_R(B) \), their reasoning algorithms only involve in a utilizer module and a realizer module without reasoning about the other utilizer module of the multiple utilizer model. These reasoning algorithms have the same module conditions as the algorithms in the simple model that also involve in a utilizer module and a realizer module. Thus, these reasoning algorithms are the same as the algorithms in the simple model.
For the query reasoning involved two concepts $A$ and $B$ that are in utilizer module $M_{U1}$ and $M_{U2}$, the reasoning algorithms need to extend the algorithms of the simple model. Moreover, the instance checking for concept query also needs to extend the algorithms of the simple model.

For the conjunction reasoning, the two concepts $A$ and $B$ of the query are respectively in utilizer modules $M_{U1}(A)$ and $M_{U2}(B)$. The two modules reuse common knowledge of realizer module $M_R$ through $I$. $A$ and $B$ are created connections through $M_R$ and $I$. In the simple model, two concepts of the conjunctive query are created connections only through an interface. Its reasoning algorithm finds out a super-concept set through the interface. In the multiple utilizer model, we need an extended conjunction reasoning algorithm to compute super-concepts through $M_R$ and $I$ as shown in Table 4.2.

Table 4.2: Extended conjunction reasoning algorithm

<table>
<thead>
<tr>
<th>Query Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
</table>
| $A \sqcap B$ ($M_{U1}(A), M_{U2}(B)$) | a) reasoning about $A$ in $M_{U1}(A)$ and $B$ in $M_{U2}(B)$  
  b) searching super-concepts $C$ of $B$ in $M_{U1}(A)$ through $M_{U2}(B)$-$I$-$M_R$-$I$-$M_{U1}(A)$  
  c) checking $A \sqcap C$ in $M_{U1}(A)$ |

As shown in Table 4.2, the conjunction reasoning algorithm reasons about $A$ and $B$ in their defined modules $M_{U1}(A)$ and $M_{U2}(B)$, then finds super-concepts $C$ of $B$ in $M_{U1}(A)$ through reasoning with modules and
interfaces from $M_U \cdot 2(B)$ to $I$, $M_R$, $I$, and end up with $M_U \cdot 1(A)$. Next, the algorithm reasons about $M_U \cdot 1(A)$ to check conjunctions of $A$ and $C$. If the reasoning results have an unsatisfiable, the algorithm will return unsatisfiable. Otherwise, the algorithm will return satisfiable.

As previously mentioned, the subsumption reasoning is based on the conjunction reasoning. The subsumption algorithm of the multiple utilizer model can use the extended conjunction reasoning algorithm to compute final reasoning results.

For the role reasoning, the two concepts $A$ and $B$ of the query are respectively in utilizer modules $M_U \cdot 1(Ro, A)$ and $M_U \cdot 2(B)$, and the role constructor $Ro$ is in $M_U \cdot 1(Ro, A)$. Similarly to the conjunction reasoning, $A$ and $B$ are created connections through $M_R$ and $I$. In the simple model, the two concepts are created connections through an interface. Its algorithm finds out super-concepts of $B$ by searching the module containing $B$ and its interface. In the multiple utilizer model, the role reasoning algorithm finds the super-concepts of $B$ in $M_U \cdot 1(Ro, A)$ after searching module $M_U \cdot 2(B)$ containing $B$, $M_R$, and $I$ as shown in Table 4.3.

Table 4.3: Extended role reasoning algorithm

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \sqsupseteq Ro.B$</td>
<td>($M_U \cdot 1(Ro, A)$, $M_U \cdot 2(B)$)</td>
<td>a) searching super-concepts $C$ of $B$ in $M_U \cdot 1(Ro, A)$ through $M_U \cdot 2(B)$-I-$M_R$-I-$M_U \cdot 1(Ro, A)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) checking $A \sqsupseteq Ro.C$ in $M_U \cdot 1(Ro, A)$</td>
</tr>
</tbody>
</table>
As shown in Table 4.3, the role reasoning algorithm finds super-concepts $C$ of $B$ in $MU_1(Ro, A)$ through reasoning with modules and interfaces from $MU_2(B)$ to $I$, $MR$, $I$, and end up with $MU_1(Ro, A)$. Next, the algorithm reasons about $MU_1(Ro, A)$ to check role queries that consists of $C$, $A$ and $Ro$. If the reasoning results have an unsatisfiable, the algorithm will return unsatisfiable. Otherwise, the algorithm will return satisfiable.

Similarly to the simple model, the instance checking for queries in the multiple utilizer model is based on its instance checking for concept algorithm and TBox reasoning algorithms. In the following, we introduce the extended algorithm for the instance checking for concept.

For the instance checking for concept, concept $A$ of the query is defined in a module. Except for reasoning with this module, in the simple model, the algorithm needs to reason about the other module to compute final reasoning results. In the multiple utilizer model, the algorithm reasons two related modules to compute final reasoning results. In the following, we will introduce the instance checking for concept algorithm from open-world reasoning and closed-world reasoning as shown in Table 4.4.

As shown in Table 4.4, the closed-world algorithm of instance checking for concept finds out sub-concepts $C$ of $A$ in modules $MU_1(A)$, $MU_2$, and $MR$ through searching interface $I$. Then, the algorithm computes whether $s$ is an instance of $C$. If the algorithm finds out that $C$ has an instance $s$ in modules
Table 4.4: Extended instance checking for concept algorithm

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
</table>
| $A(s)$ | $M_U1(A)$, $M_U2$ and $M_R$ | **Closed-world** algorithm:  
a) searching sub-concepts $C$ of $A$ in $M_U1(A)$, $M_U2$, and $M_R$ through $I$  
b) checking $C(s)$ in $M_U1(A)$, $M_U2$, and $M_R$  
**Open-world** algorithm:  
a) searching super-concepts $C$ of $A$ in $M_U1(A)$, $M_U2$, and $M_R$ through $I$  
b) checking $C(s)$ in $M_U1(A)$, $M_U2$, and $M_R$ |

$M_U1(A)$, $M_U2$, or $M_R$, the algorithm will return consistent. Otherwise, the algorithm will return inconsistent.

The open-world algorithm computes super-concept $C$ of $A$ in $M_U1(A)$, $M_U2$, and $M_R$ through searching $I$. Then, the algorithm checks consistency of $C(s)$ in $M_U1(A)$, $M_U2$, and $M_R$. If the reasoning results have an inconsistent, the algorithm will return inconsistent. Otherwise, the algorithm will return consistent.

### 4.1.2 Multiple Realizer Model and Reasoning

This section introduces the multiple realizer model and its query reasoning. The query reasoning includes using same reasoning algorithms for the simple model, and using extended reasoning algorithms.
4.1.2.1 Multiple Realizer Model

The multiple realizer model means that its utilizer module reuses multiple realizer modules through interfaces. As shown in Figure 4.2, we illustrate a multiple realizer model that consists of three modules and two interfaces.

![Figure 4.2: A multiple realizer model](image)

As shown in Figure 4.2, realizer modules $M_{R1}$ or $M_{R2}$ provide knowledge to utilizer module $M_U$ through interfaces $I_1$ or $I_2$.

4.1.2.2 Query Reasoning Using Same Reasoning Algorithms for the Simple Model

Similarly to the multiple utilizer model, in the multiple realizer model, the single concept, and disjunction reasoning algorithms are similar to the
algorithms of the simple model. The other distributed query reasoning algorithms involved two concepts that any of two concepts $A$ and $B$ is in utilizer module $M_U$ are same as the algorithms of the simple model. In the following, we illustrate these query reasoning algorithms in Table 4.5.

Table 4.5: Query reasoning using same reasoning algorithms of the simple model

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \lor \neg A$</td>
<td>$M_{R1}(A), M_{R2}(A)$, or $M_U(A)$</td>
<td>$ConceptReasoning$</td>
</tr>
<tr>
<td>$A \sqcup B$</td>
<td>$(M_{R1}(A), M_U(B))$, or $(M_{R2}(A), M_U(B))$,</td>
<td>$DisjunctionReasoning$</td>
</tr>
<tr>
<td></td>
<td>$\lor (M_{R1}(A), M_{R2}(B))$</td>
<td></td>
</tr>
<tr>
<td>$A \sqcap \forall Ro.B$</td>
<td>$(M_{R1}(A), M_U(B))$, or $(M_{R2}(A), M_U(B))$</td>
<td>$ConjunctionReasoning$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Similarly to the simple model, the single concept reasoning, including concept and negation reasoning, also involves in the module containing the query and its interface. As shown in Table 4.5, the algorithms are same as the algorithms of the simple model, and use the $ConceptReasoning$ algorithm to return final results.

The disjunction reasoning is based on the single concept reasoning. Its algorithm is same as the $DisjunctionReasoning$ algorithm of the simple model in chapter 3.

As shown in Table 4.5, for the conjunction, subsumption, and role query that their two concepts $A$ and $B$ are in realizer module $M_{R1}(A)$ or $M_{R2}(A)$, and in utilizer module $M_U(B)$, their reasoning algorithms involve in a realizer module and a utilizer module without reasoning about the other realizer.
module of the multiple realizer model. These reasoning algorithms also have same module conditions of the algorithms in the simple model. Thus, these algorithms are same as the algorithms in the simple model.

4.1.2.3 Query Reasoning Using Extended Reasoning Algorithms

For the query reasoning involved two concepts $A$ are $B$ that are in realizer modules $M_{R1}$ and $M_{R2}$, the reasoning algorithms need to extend the algorithms of the simple model. Moreover, the instance checking for concept query also needs to extend the algorithms of the simple model.

For the conjunction reasoning, the two concepts $A$ and $B$ of the query are respectively in realizer modules $M_{R1}$ and $M_{R2}$. The two modules provide knowledge to utilizer module $M_{U}$ through $I_{1}$ and $I_{2}$. $A$ and $B$ are created connections through $M_{U}$, $I_{1}$ and $I_{2}$. In the simple model, the two concepts are created connections only through an interface. Its conjunction reasoning algorithm finds out a super-concept set in the interface. In the multiple realizer model, we need an extended conjunction reasoning algorithm to compute super-concepts through $M_{U}$, $I_{1}$ and $I_{2}$ as shown in Table 4.6.

Table 4.6: Extended conjunction reasoning algorithm

<table>
<thead>
<tr>
<th>Query Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \sqcap B$ ($M_{R1}(A), M_{R2}(B)$)</td>
<td>a) reasoning about $A$ in $M_{R1}(A)$ and $B$ in $M_{R2}(B)$</td>
</tr>
<tr>
<td></td>
<td>b) searching super-concepts $C$ of $B$ in $M_{R1}(A)$</td>
</tr>
<tr>
<td></td>
<td>through $M_{R2}(B)$-$I_{2}$-$M_{U}$-$I_{1}$-$M_{R1}(A)$</td>
</tr>
<tr>
<td></td>
<td>c) checking $A \sqcap C$ in $M_{R1}(A)$</td>
</tr>
</tbody>
</table>
As shown in Table 4.6, the conjunction reasoning algorithm reasons about $A$ and $B$ in their defined modules $M_{R1}(A)$ and $M_{R2}(B)$, then finds out super-concepts $C$ of $B$ in $M_{R1}(A)$ after sequentially reasoning with $M_{R2}(B)$, $I_2$, $M_U$, $I_1$, and $M_{R1}(A)$. Next, the algorithm reasons about $M_{R1}(A)$ to check conjunctions of $A$ and $C$. If the reasoning results have an unsatisfiable, the algorithm will return unsatisfiable. Otherwise, the algorithm will return satisfiable.

Similarly to the other model, the subsumption reasoning is based on the conjunction reasoning. The subsumption algorithm of the multiple realizer model uses the extended conjunction reasoning algorithm to compute final reasoning results.

For the role reasoning, the two concepts $A$ and $B$ of the query are respectively in realizer modules $M_{R1}(Ro, A)$ and $M_{R2}(B)$, and the role constructor $Ro$ is in $M_{R1}(Ro, A)$. Similarly to the conjunction reasoning, $A$ and $B$ are created connections through $M_U$, $I_1$, and $I_2$. In the simple model, the two concepts are created connections through an interface. Its algorithm finds out super-concepts of $B$ by searching the module containing $B$ and its interface. In the multiple realizer model, the role reasoning algorithm computes the super-concepts of $B$ in $M_{R1}(Ro, A)$ after searching module $M_{R2}(B)$, $M_U$, $I_1$, and $I_2$ as shown in Table 4.7.
As shown in Table 4.7, the role reasoning algorithm finds super-concepts $C$ of $B$ in $M_{R_1}(Ro, A)$ by sequentially searching module $M_{R_2}(B)$, $I_2$, $M_U$, $I_1$, and $M_{R_1}(Ro, A)$. Then, the algorithm checks role query $A \sqsubseteq \forall Ro.C$ in module $M_{R_1}(Ro, A)$. If the reasoning results have an unsatisfiable, the algorithm will return unsatisfiable. Otherwise, the algorithm will return satisfiable.

Similarly to the simple model, the instance checking for queries in the multiple realizer model is based on its instance checking for concept algorithm and TBox reasoning algorithms.

For the instance checking for concept, concept $A$ of the query is defined in a module. Except for reasoning with this module, in the simple model, the algorithm needs to reason about the other module to compute final reasoning results. In the multiple realizer model, the algorithm reasons with two related modules to compute final reasoning results. In the following, we will introduce the instance checking for concept algorithm from open-world and closed-world reasoning as shown in Table 4.8.

As shown in Table 4.8, the closed-world algorithm finds out sub-concepts $C$ of $A$ in $M_{R_1}(A)$, $M_{R_2}$, and $M_U$ through $I_1$ and $I_2$. Then, the algorithm

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
</table>
| $A \sqsubseteq \forall Ro.B$ | $(M_{R_1}(Ro, A), M_{R_2}(B))$ | a) searching super-concepts $C$ of $B$ in $M_{R_1}(Ro, A)$ through $M_{R_2}(B)$-I$_2$-M$_U$-I$_1$-M$_{R_1}(Ro, A)$
| | | b) checking $A \sqsubseteq \forall Ro.C$ in $M_{R_1}(Ro, A)$ |

73
Table 4.8: Extended instance checking for concept algorithm

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
</table>
| A(s)   | \( M_{R1}(A) \), \( M_{R2} \) and \( M_{U} \) | **Closed-world** algorithm:  
  a) searching sub-concepts \( C \) of \( A \) in \( M_{R1}(A) \), \( M_{R2} \) and \( M_{U} \) through \( I_{1} \) and \( I_{2} \)  
  b) checking \( C(s) \) in \( M_{R1}(A) \), \( M_{R2} \) and \( M_{U} \)  
**Open-world** algorithm:  
  a) searching super-concepts \( C \) of \( A \) in \( M_{R1}(A) \), \( M_{R2} \) and \( M_{U} \) through \( I_{1} \) and \( I_{2} \)  
  b) checking \( C(s) \) in \( M_{R1}(A) \), \( M_{R2} \) and \( M_{U} \) |

computes whether \( s \) is an instance of \( C \). If the algorithm finds that \( C \) has \( s \) in \( M_{R1}(A) \), \( M_{R2} \), or \( M_{U} \), the algorithm will return consistent. Otherwise, the algorithm will return inconsistent.

The open-world algorithm finds out super-concepts \( C \) of \( A \) in \( M_{R1}(A) \), \( M_{R2} \), and \( M_{U} \) through \( I_{1} \) and \( I_{2} \). Then, the algorithm computes consistency of \( C(s) \) in \( M_{R1}(A) \), \( M_{R2} \), and \( M_{U} \). If the reasoning results have an inconsistent, the algorithm will return inconsistent. Otherwise, the algorithm will return consistent.

### 4.1.3 Realizer-Utilizer Model and Reasoning

This section introduces the realizer-utilizer model and its query reasoning. The query reasoning includes using same reasoning algorithms for the simple model, and using extended reasoning algorithms.
4.1.3.1 Realizer-Utilizer Model

The realizer-utilizer model has realizer-utilizer modules. The realizer-utilizer module simultaneously is a utilizer module and a realizer module. As shown in Figure 4.3, we illustrate a realizer-utilizer model that consists of three modules and two interfaces.

As shown in Figure 4.3, realizer module $Module_1$ or $M_R$ provides knowledge to realizer-utilizer module $Module_2$ or $M_{RU}$ through interface $Interface_1$ or $I_1$, and realizer-utilizer module $M_{RU}$ provides knowledge to utilizer module $Module_3$ or $M_U$ through interface $Interface_2$ or $I_2$. 
4.1.3.2 Query Reasoning Using Same Reasoning Algorithms for the Simple Model

In the realizer-utilizer model, the single concept, and disjunction reasoning algorithms are similar to the algorithms of the simple model as shown in Table 4.9.

Table 4.9: Query reasoning using same reasoning algorithms of the simple model

<table>
<thead>
<tr>
<th>Query Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \lor \neg A$</td>
<td>$M_R(A), M_{RU}(A),$ or $M_U(A)$</td>
</tr>
<tr>
<td>$A \lor B$</td>
<td>$(M_R(A), M_U(B)),$ $(M_{RU}(A), M_U(B)),$ or $(M_R(A), M_{RU}(B))$</td>
</tr>
</tbody>
</table>

Similarly to the other model, the single concept reasoning in the realizer-utilizer model also involves in the module containing the query. As shown in Table 4.9, its algorithms are same as the algorithms of the simple model, and use the ConceptReasoning algorithm to return final results. The disjunction reasoning is based on the single concept reasoning so that its algorithm is same as the DisjunctionReasoning algorithm of the simple model.

4.1.3.3 Query Reasoning Using Extended Reasoning Algorithms

For the query reasoning involved two concepts $A$ and $B$ that are in realizer module $M_R(A)$ and utilizer module $M_U(B)$ in the model, the reasoning algorithms need to extend the algorithms of the simple model. Moreover,
the instance checking for concept query also needs to extend the algorithms of the simple model.

For the conjunction reasoning, the two concepts $A$ and $B$ of the query are respectively in $M_R(A)$ and $M_U(B)$. The realizer-utilizer module $M_{RU}$ uses knowledge from $M_R$ through interface $I_1$, and provides knowledge to $M_U$ through interface $I_2$. $A$ and $B$ are created connections through $M_{RU}$, $I_1$, and $I_2$. In the simple model, the two concepts are created connections only through an interface. Its conjunction reasoning algorithm finds out a super-concept set in the interface. In the realizer-utilizer model, we need an extended conjunction reasoning algorithm to compute super-concepts through $M_{RU}$, $I_1$, and $I_2$ as shown in Table 4.10.

Table 4.10: Extended conjunction reasoning algorithm

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
</table>
| $A \cap B$ | $(M_R(A), M_U(B),$ and $M_{RU})$ | a) reasoning about $A$ in $M_R(A)$ and $B$ in $M_U(B)$  
 b) searching super-concepts $C$ of $B$ in $M_R(A)$ through $M_U(B)$-$I_2$-$M_{RU}$-$I_1$-$M_R(A)$  
 c) checking $A \cap C$ in $M_R(A)$ |

As shown in Table 4.10, the conjunction reasoning algorithm checks $A$ and $B$ in their defined modules $M_R(A)$ and $M_U(B)$, then finds super-concepts $C$ of $B$ in $M_R(A)$ after sequentially searching $M_U(B)$, $I_2$, $M_{RU}$, $I_1$, and $M_R(A)$. Next, the algorithm reasons about $M_R(A)$ to check conjunctions of $A$ and $C$. If the reasoning results have an unsatisfiable, the algorithm will return unsatisfiable. Otherwise, the algorithm will return satisfiable.
Similarly to the other models, the subsumption reasoning is based on the conjunction reasoning. The subsumption algorithm of the realizer-utilizer model uses the extended conjunction reasoning algorithm to compute final reasoning results.

For the role reasoning, concept $A$ and role constructor $Ro$ are in $M_R(Ro, A)$, and $B$ is in $M_U(B)$. Similarly to the conjunction reasoning, $A$ and $B$ are created connections through $M_{RU}$, $I_1$, and $I_2$. In the simple model, the two concepts are created connections through an interface. Its algorithm finds out super-concepts of $B$ by searching the module containing $B$ and its interface. In the realizer-utilizer model, the role reasoning algorithm computes the super-concepts of $B$ through $M_{RU}$, $I_1$, and $I_2$ as shown in Table 4.11.

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \sqsubseteq \forall Ro.B$</td>
<td>$(M_R(Ro, A))$, $M_U(B)$, $M_{RU}$</td>
<td>a) searching super-concepts $C$ of $B$ in $M_R(Ro, A)$</td>
</tr>
<tr>
<td>$A \sqsubseteq \forall Ro.C$</td>
<td></td>
<td>b) checking $A \sqsubseteq \forall Ro.C$ in $M_R(Ro, A)$</td>
</tr>
</tbody>
</table>

As shown in Table 4.11, the role reasoning algorithm computes super-concepts $C$ of $B$ in $M_R(Ro, A)$ through sequentially searching $M_U(B)$, $I_2$, $M_{RU}$, $I_1$, and $M_R(Ro, A)$. Then, the algorithm checks role query $A \sqsubseteq \forall Ro.C$ in module $M_R(Ro, A)$. If the reasoning results have an unsatisfiable, the algorithm will return unsatisfiable. Otherwise, the algorithm will return satisfiable.
Similarly to the simple model, the instance checking for queries in the realizer-utilizer model is based on its instance checking for concept algorithm and TBox reasoning algorithms.

For the instance checking for concept, concept $A$ of the query is defined in a module. Except for reasoning with this module, in the simple model, the algorithm only needs to reason about the other module to compute final reasoning results. In the realizer-utilizer model, its algorithm needs to reason with two related modules to compute final reasoning results. In the following, we will introduce the instance checking for concept algorithm from open-world reasoning and closed-world reasoning as shown in Table 4.12.

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(s)$</td>
<td>$M_R(A)$, $M_{RU}$ and $M_U$</td>
<td><strong>Closed-world</strong> algorithm:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a) searching sub-concepts $C$ of $A$ in $M_R(A)$, $M_{RU}$ and $M_U$ through $I_1$ and $I_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) checking $C(s)$ in $M_R(A)$, $M_{RU}$ and $M_U$</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Open-world</strong> algorithm:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a) searching super-concepts $C$ of $A$ in $M_R(A)$, $M_{RU}$ and $M_U$ through $I_1$ and $I_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) checking $C(s)$ in $M_R(A)$, $M_{RU}$ and $M_U$</td>
</tr>
</tbody>
</table>

As shown in Table 4.12, the closed-world algorithm finds out sub-concepts $C$ of $A$ in $M_R(A)$, $M_{RU}$, and $M_U$ through $I_1$ and $I_2$. Then, the algorithm computes whether $s$ is an instance of $C$. If the algorithm finds that $C$ has $s$ in $M_R(A)$, $M_{RU}$, or $M_U$, the algorithm will return consistent. Otherwise, the algorithm will return inconsistent.
The open-world algorithm finds out super-concepts $C$ of $A$ in $M_R(A)$, $M_{RU}$, and $M_U$ through $I_1$ and $I_2$. Then, the algorithm computes consistency of $C(s)$ in $M_R(A)$, $M_{RU}$, and $M_U$. If the reasoning results have an inconsistent, the algorithm will return inconsistent. Otherwise, the algorithm will return consistent.

4.1.4 General Model and Reasoning

This section introduces the general model and its query reasoning. The query reasoning includes using same reasoning algorithms for the simple model, and using extended reasoning algorithms.

4.1.4.1 General Model

The general model combines the extended models including the multiple utilizer model, multiple realizer model and realizer-utilizer model into a new generalized extension model. We assume that the general model is a non-cyclical model, which means that any two modules of the model cannot reuse each other at the same time. As shown in Figure 4.4, we illustrate a general model that consists of five modules and three interfaces.

As shown in Figure 4.4, utilizer module $Module_5$ or $M_U2$ reuses knowledge from realizer module $Module_1$ or $M_R$ and realizer-utilizer module $Module_3$.
or $M_{RU}2$ through $Interface_1$ or $I_1$ and $Interface_3$ or $I_3$. The other utilizer module $Module_4$ or $M_U1$ also reuses knowledge from $Module_3$ through $Interface_3$. In addition, $Module_3$ reuses knowledge from realizer-utilizer module $Module_2$ or $M_{RU}2$ through $Interface_2$ or $I_2$. $Module_2$ reuses knowledge from $Module_1$ through $Interface_1$.

### 4.1.4.2 Query Reasoning Using Same Reasoning Algorithms for the Simple Model

In the general model, the single concept, and disjunction reasoning algorithms are similar to the algorithms of the simple model as shown in Table 4.13.
Table 4.13: Query reasoning using same reasoning algorithms

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>A or ¬A</td>
<td>$M_R(A)$, $M_U1(A)$, $M_U2(A)$, $M_{RU1}(A)$, or $M_{RU2}(A)$</td>
<td>ConceptReasoning</td>
</tr>
<tr>
<td>A $\sqcup$ B</td>
<td>$(M_R(A), M_U1(B)), (M_R(A), M_U2(B))$, or $(M_R(A), M_{RU1}(B))$, etc.</td>
<td>DisjunctionReasoning</td>
</tr>
</tbody>
</table>

Similarly to the other model, the concept and negation reasoning in the general model also involve in the module containing the single query. The algorithms are same as the algorithms of the simple model, and also use the `ConceptReasoning` algorithm to compute final results. The disjunction reasoning is based on the single concept reasoning so that its algorithm is same as the `DisjunctionReasoning` algorithm of the simple model.

### 4.1.4.3 Query Reasoning Using Extended Reasoning Algorithms

For the query reasoning involved two concepts $A$ and $B$ that are respectively in module $M_i(A)$ and module $M_j(B)$ of the general model, the reasoning algorithms need to extend the algorithms of the simple model. Moreover, the instance checking for concept query also needs to extend the algorithms of the simple model.

For the conjunction reasoning, the two concepts $A$ and $B$ of the query are respectively in $M_i(A)$ and $M_j(B)$. $M_i(A)$ could reuse or provide knowledge in $M_j(B)$ from multiple modules and interfaces. $A$ and $B$ are created connections through the modules between $M_i(A)$ and $M_j(B)$. In the simple
model, the two concepts are created connections only through an interface. Its conjunction reasoning algorithm finds out a super-concept set in the interface. In the general model, we need an extended algorithm to compute super-concepts through the related modules between $M_i(A)$ and $M_j(B)$.

The related modules could form multiple paths to connect $M_i(A)$ and $M_j(B)$. For example, as shown in Figure 4.4, $Module_1$ and $Module_5$ are connected through two paths, including from $Module_1$ to $Interface_1$, and end up with $Module_5$, and from $Module_1$ to $Interface_1$, $Module_2$, $Interface_2$, $Module_3$, $Interface_3$, and end up with $Module_5$. As shown in Table 4.14, the extended conjunction reasoning algorithm computes super-concepts along the paths between $M_i$ and $M_j$.

Table 4.14: Extended conjunction reasoning algorithm

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
</table>
| $A \sqcap B$ | $M_i(A)$, $M_j(B)$, and modules between $M_i(A)$ and $M_j(B)$ | a) reasoning about $A$ in $M_i(A)$ and $B$ in $M_j(B)$  
b) finding $M_k$ for each path from $M_i$ to $M_j$  
c) searching super-concepts $A_S$ of $A$ in all $M_k$ through reasoning from $M_i(A)$ to $M_k$  
d) searching super-concepts $B_S$ of $B$ in all $M_k$ through reasoning from $M_j(B)$ to $M_k$  
e) checking $A_S \sqcap B_S$ in $M_k$ |

As shown in Table 4.14, the conjunction reasoning algorithm checks $A$ and $B$ in $M_i(A)$ and $M_j(B)$. Then, for each path from $M_i(A)$ to $M_j(B)$, the algorithm finds a module $M_k$ in the path. Next, it finds all super-concepts $A_S$ of $A$ and $B_S$ of $B$ in all modules $M_k$. After that, the algorithm checks conjunctions of each pair of $A_S$ and $B_S$ in $M_k$. If the reasoning results
have an unsatisfiable, the algorithm will return unsatisfiable. Otherwise, the algorithm will return satisfiable.

Similarly to the other models, the subsumption reasoning is based on the conjunction reasoning. The subsumption algorithm of the general model uses the extended conjunction reasoning algorithm to compute final reasoning results.

For the role reasoning, two concepts $A$ and $B$ of the query are respectively in $M_i(A, Ro)$ and $M_j(B)$, and the role constructor $Ro$ is in $M_i(A, Ro)$. Similarly to the conjunction reasoning, $A$ and $B$ are created connections through multiple paths that consist of related modules and interfaces. In the simple model, the two concepts are created connections through an interface. Its algorithm finds out super-concepts of $B$ by searching the module containing $B$ and its interface. In the general model, the algorithm computes super-concepts of $B$ by searching along all paths from $M_i(A, Ro)$ to $M_j(B)$ as shown in Table 4.15.

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
</table>
| $A \sqsubseteq\forall Ro.B$ | $M_i(A, Ro)$, $M_j(B)$, and modules between $M_i(A)$ and $M_j(B)$ | a) searching super-concepts $C$ of $B$ in $M_i(A, Ro)$ through reasoning with all paths from $M_j(B)$ to $M_i(A, Ro)$  
  b) checking $A \sqsubseteq\forall Ro.C$ in $M_i(A, Ro)$ |

As shown in Table 4.15, the role reasoning algorithm computes super-concepts $C$ of $B$ in $M_i(A, Ro)$ by searching along all paths from
$M_j(B)$ to $M_i(A, Ro)$. Then, the algorithm checks $A \sqsubseteq \forall Ro.C$ in $M_i(A, Ro)$.

If the reasoning results have an unsatisfiable, the algorithm will return unsatisfiable. Otherwise, the algorithm will return satisfiable.

Similarly to the simple model, the instance checking for queries in the general model is based on its instance checking for concept algorithm and TBox reasoning algorithms.

For the instance checking for concept, concept $A$ of the query is defined in a module. Except for reasoning with this module, in the simple model, the algorithm only reasons about the other module to compute the final reasoning results. In general model, the algorithm checks multiple related modules to compute final reasoning results. The related modules realize and/or utilize knowledge from the module defined $A$. For example, as shown in Figure 4.4, we assume that $A$ is defined in $Module_1$. Its related modules include $Module_2$, $Module_3$, $Module_4$, and $Module_5$ of the model. In the following, we will introduce the instance checking for concept algorithm from open-world reasoning and closed-world reasoning as shown in Table 4.16.

As shown in Table 4.16, the closed-world algorithm finds out sub-concepts $C$ of $A$ in $M_i(A)$ and another related modules through interfaces. Using the computed $C$, the algorithm computes whether $s$ is an instance of $C$. If the algorithm finds that $C$ has $s$ in $M_i(A)$ or related modules, the algorithm will return consistent. Otherwise, the algorithm will return inconsistent.

85
### Table 4.16: Extended instance checking for concept algorithm

<table>
<thead>
<tr>
<th>Query</th>
<th>Involved Modules</th>
<th>Reasoning Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(s)$</td>
<td>$M_i(A)$ and related modules</td>
<td><strong>Closed-world</strong> algorithm:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a) searching sub-concepts $C$ of $A$ in $M_i(A)$ and related modules through interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) checking $C(s)$ in $M_i(A)$ and related modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Open-world</strong> algorithm:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a) searching super-concepts $C$ of $A$ in $M_i(A)$ and related modules through interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) checking $C(s)$ in $M_i(A)$ and related modules</td>
</tr>
</tbody>
</table>

The open-world algorithm finds out super-concepts $C$ of $A$ in $M_i(A)$ and related modules through interfaces. Then, the algorithm computes consistency of $C(s)$ in $M_i(A)$ and related modules. If the reasoning results have an inconsistent, the algorithm will return inconsistent. Otherwise, the algorithm will return consistent.

## 4.2 Compound Query and Reasoning

A compound query consists of multiple concepts and constructors. Simple queries introduced earlier that are concept, negation, disjunctive, conjunctive, and role query only involve one or two concepts and a single constructor. A compound query can be decomposed into multiple simple queries. For example, a compound query $(A \sqcap B) \sqcap C$ can be decomposed to $A \sqcap B$, and $P \sqcap C$ where $P$ is $A \sqcap B$.
In a general modular ontology, compound queries could involve concepts in different modules, and be distributedly reasoned to compute their reasoning results. In the following, we will introduce reasoning procedures for compound queries in general modular ontologies.

There is a simple way to distributedly reason with a compound query. When the system receives a compound query, it will change types of related modules from realizer and utilizer to realizer and realizer-utilizer, and create a new utilizer module with this compound query as the axiom. Meanwhile, this new utilizer module will reuse knowledge from all modules related to this query. In order to successfully reuse knowledge, the system will export new interfaces from the related modules to be utilized by the new utilizer module. Thus, the system extends the original modular ontology with the added utilizer module for reasoning of the query.

For example, a simple ontology extension process for reasoning the compound query \((A \cap B) \cap C\) is illustrated as follows. We assume that \(A\), \(B\) and \(C\) are respectively defined in realizer module \(M_R\), utilizer module \(M_{U1}\), and utilizer module \(M_{U2}\). \(M_{U1}\) and \(M_{U2}\) reuse \(M_R\) through interface \(I_1\). When the system receives the query, it will create a new utilizer module \(M_{U3}\) with axiom \((A \cap B) \cap C\). Utilizer modules \(M_{U1}\) and \(M_{U2}\) are changed to realizer-utilizer modules \(M_{RU1}\) and \(M_{RU2}\). \(M_R\), \(M_{RU1}\) and \(M_{RU2}\) will export new interfaces \(I_2\), \(I_3\) and \(I_4\), which contain knowledge about \(A\), \(B\) and \(C\). Then, \(M_{U3}\) reuses knowledge from \(M_R\), \(M_{RU1}\) and \(M_{RU2}\) through \(I_2\), \(I_3\) and \(I_4\).
Now, the system has a new modular ontology for the compound query, which contains modules $M_R$, $M_{RU1}$, $M_{RU2}$, $M_{U3}$, and interfaces $I_1$, $I_2$, $I_3$, and $I_4$.

After the system extends the modular ontologies for compound query reasoning, it will begin to reason with the compound query using the algorithms introduced earlier. Firstly, the system decomposes the compound query to simple queries, which is similar to a binary tree as shown in Figure 4.5.

Then, starting from leaves of a compound query tree, the system reasons about the simple queries from bottom-up using the distributed algorithms described in section 4.1.4.

![Figure 4.5: An example of decomposed compound query](image)

For example, for the compound query $(A \cap B) \cap C$, after the decomposition process, the system acquires the simple queries as shown in Figure 4.5, and starts to reason from the bottom of the tree. Firstly, the system will check whether the query $A \cap B$ is satisfiable by the extended $ConjunctionReasoning$ algorithm for the general model, then we assume that $A \cap B$ is equal to a
new concept $P$ in utilizor module $M_{U3}$. Next, the system finds out super-concepts of $P$ in interfaces $I_1$ and $I_4$, where the two interfaces are directly related to module $M_{RU2}$ that defines $C$. Finally, the system can continue use the extended *ConjunctionReasoning* algorithm to compute final results by reasoning about $C$ with the computed super-concepts of $P$.

### 4.3 Summary

In this chapter, we described the extended reasoning algorithms for general distributed modular ontologies based on the reasoning algorithms for simple distributed modular ontologies described in chapter 3. We showed the extended reasoning algorithms from two different perspectives. One is from the extended modular ontology models with simple queries, and the other is from the general model with compound queries.

The extended modular ontology models with simple queries have more types of relationships between modules than the simple model introduced in chapter 3. In this part, we introduced these extended models and their reasoning algorithms for same simple or canonical queries as in chapter 3. Then, we compared similarity and difference between these extended algorithms and the algorithms of chapter 3.
The general modular ontology model with compound queries involves with combinations of all the extended models as well as reasoning about compound queries. In this part, we introduced compound queries, and their reasoning algorithms in the extended and general models based on the reasoning algorithms given in chapter 3.
Chapter 5

Distributed Modular Ontology Reasoning System

In this chapter, we will introduce system functionality, system architecture and functionality realization for the distributed modular ontology reasoning system.

5.1 System Functionality

The distributed modular ontology reasoning system allows ontology providers to provide ontology modules and interfaces, and ontology creators to assemble ontology modules and interfaces to create modular ontologies. After checking
the created ontologies, the system can process user queries to user-selected modular ontologies, reason about them, and show reasoning results.

The distributed modular ontology reasoning system supports three types of functionality, including information system, distributed modular ontology creation, and user query processing.

5.1.1 Information System

In the system, the information system stores ontology interfaces, information of ontology modules, and information of assembled modular ontologies. The system users can search available interfaces, information of ontology modules and information of assembled modular ontologies using the information system. The ontology module and interface providers can add interfaces and information of ontology modules into the information system.

The information system contains the ontology interface database, ontology module information database, and assembled modular ontology information database.
5.1.1.1 Ontology Interface Database

The ontology interface database stores a set of ontology interfaces. As shown in Figure 5.1, the ontology interface database consists of Interface table. The table contains InterfaceName, Description, and Interface. InterfaceName stores interfaces’ names. Description stores simple descriptions of interface contents. Interface is form of TBox in OWL.

![Interface Table](image)

Figure 5.1: The interface table

In addition, the ontology interface database supports a user interface for ontology interface providers to add and edit interfaces in the database, and other components of the system to retrieve interfaces.

5.1.1.2 Ontology Module Information Database

As mentioned before, in this system, all ontology modules are distributedly stored in different nodes. The ontology module information database only stores information about ontology modules, including realizing and/or utilizing interfaces as well as their locations or nodes. Between ontology
interfaces and modules, there are many-to-many relationships. Figure 5.2 illustrates the design of ontology module information database.

Figure 5.2: The design of ontology module information database

As shown in Figure 5.2, the design describes ontology module information database, including tables \textit{ModuleInformation} and \textit{Interface}. There are many-to-many utilized and realized relations between \textit{ModuleInformation} and \textit{Interface}. In a modular ontology, a module could utilize and/or realize multiple interfaces. An interface could be utilized and/or realized by multiple modules.

\textit{ModuleInformation} table contains \textit{ModuleName}, \textit{Description}, \textit{Location}, and \textit{ExportedConcepts}. \textit{ModuleName} stores modules’ names. \textit{Description} stores simple descriptions of modules. \textit{Location} stores storage information of modules. \textit{ExportedConcepts} stores some concept names of modules that can be used to help users issue queries.

In addition, the ontology module information database supports a user interface for ontology module providers to add and edit the information of
modules in the database, and other components of the system can retrieve module information from the database.

Also, before adding information of a new module into the database, the system needs to check whether the new module is valid. The checking functions include computing whether a new module already had its related interfaces in the ontology interface database, and whether a new module and related interfaces are consistent.

5.1.1.3 Assembled Modular Ontology Information Database

The assembled modular ontology information database stores information about assembled modular ontologies. An assembled modular ontology could consist of multiple modules and interfaces. Also one module or interface could belong to multiple assembled modular ontologies. Figure 5.3 shows the design of assembled modular ontology information database.

As shown in Figure 5.3, the database stores assembled modular ontology information, which involves tables Interface, ModuleInformation and ModularOntologyInformation. There are many-to-many relations between ModuleInformation and ModularOntologyInformation. A modular ontology could consist of multiple modules. A modules could belong to multiple modular ontologies. Moreover, tables Interface and ModuleInformation of this database are sub-set of these two tables in the module information
ModularOntologyInformation could use a part of modules and interfaces of the module information database.

ModularOntologyInformation table stores information of assembled modular ontologies, including AssembledOntologyName, ModuleName, and Interfaces. AssembledOntologyName stores assembled modular ontologies’ names. ModuleName stores names of constituent modules of assembled modular ontologies. Interfaces stores utilized and/or realized interfaces of each module in assembled modular ontologies.

The system utilizes the assembled modular ontology information database to retrieve information of assembled modular ontologies, and helps other components to add new information of assembled modular ontologies into the database.
5.1.2 Distributed Modular Ontology Creation

The distributed modular ontology creation functions are for modular ontology creators to create distributed assembled modular ontologies. They include the ontology assemblage sub-function and assembled modular ontology checking sub-function.

5.1.2.1 Ontology Assemblage

The ontology assemblage sub-function allows a modular ontology creator to view information of ontology modules and ontology interfaces, and assemble them to create a modular ontology. Figure 5.4 illustrates the flow diagram of ontology assemblage.

As shown in Figure 5.4, the modular ontology creator firstly views information of available modules and interfaces from the information database. After viewing, the creator can use the ontology assemblage user interface to select required ontology modules and interfaces, and assemble them into a new modular ontology.

This assembling procedure does not need to relocate any module or interface from the databases. The ontology assemblage only records information of constituted modules and interfaces in new assembled modular ontologies.
5.1.2.2 Assembled Modular Ontology Checking

The assembled modular ontology checking sub-function checks whether a newly assembled modular ontology from the ontology assemblage is valid, which includes completeness checking and consistency checking.

The assembled modular ontology checking firstly checks whether the newly created ontologies from the ontology assemblage are complete. That is each utilizer module in an assembled modular ontology must have its all required realizer modules and interfaces also in the assembled modular ontology.

The consistency checking checks whether a completely assembled modular ontology is consistent. If its checking result is consistent, the distributed
modular ontology creation will write information of the new modular ontology into the assembled modular ontology information database. Otherwise, it will reject to add information of this ontology into the assembled modular ontology information database, and return error messages to the creator.

5.1.3 User Query Processing

The user query function allows a user to select an assembled modular ontology and issue queries to the ontology, reasoning about queries and show reasoning results to the user.

5.1.3.1 Supported System Queries

Supported system queries function provides available query samples and concepts of queries for users. Users can retrieve the supported system queries to issue available queries.

The query samples include simple or canonical query samples and compound query samples. Both of them provide DL formats (see Glossary 2.) and input formats (User Query). Users can issue an available query based on the input formats of queries as shown in Table 5.1.
<table>
<thead>
<tr>
<th>Query Type</th>
<th>User Query</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept query</td>
<td>$A$</td>
<td>$A$</td>
</tr>
<tr>
<td>Negated concept query</td>
<td>$\neg A$</td>
<td>$\neg A$</td>
</tr>
<tr>
<td>Disjunctive query</td>
<td>$A \lor B$</td>
<td>$A \lor B$</td>
</tr>
<tr>
<td>Conjunctive query</td>
<td>$A \land B$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>Subsumption query</td>
<td>$A$ subsumed $B$</td>
<td>$A \sqsubseteq B$</td>
</tr>
<tr>
<td>Role query</td>
<td>$R(A, B)$</td>
<td>$A \sqsubseteq \forall R.B$</td>
</tr>
<tr>
<td>Instance for concept query</td>
<td>$A(a)$</td>
<td>$A(a)$</td>
</tr>
<tr>
<td>Instance for negated concept query</td>
<td>$\neg A(a)$</td>
<td>$\neg A(a)$</td>
</tr>
<tr>
<td>Instance for disjunctive query</td>
<td>$A \lor B(s)$</td>
<td>$A \lor B(s)$</td>
</tr>
<tr>
<td>Instance for conjunctive query</td>
<td>$A \land B(s)$</td>
<td>$A \land B(s)$</td>
</tr>
<tr>
<td>Instance for role query</td>
<td>$R(a, b)$</td>
<td>$R(a, b)$</td>
</tr>
<tr>
<td>Compound query</td>
<td>$(A \land B) \lor \neg C$</td>
<td>$(A \land B) \lor \neg C$</td>
</tr>
</tbody>
</table>

As shown in Table 5.1, the query types include the concept query, negated concept query, disjunctive query, conjunctive query, subsumption query, role query, instance for concept query, instance for negated concept query, instance for disjunctive query, instance for conjunctive query, instance for role query, and compound query.

The available concepts of queries include all contents of interfaces, and contents of a local module, and exported concepts of the module information database. Query users can retrieve the available concepts and issue queries based on the concepts.
5.1.3.2 Query User Interface

The query user interface supports queriers to enter queries and view reasoning results. The query input of the query user interface allows queriers to directly enter queries for selected assembled modular ontologies. The result display of the query user interface shows reasoning results of valid issued queries. The reasoning results for TBox queries show satisfiability of the queries. The reasoning results for ABox queries show consistency of the queries.

5.1.3.3 Query Validation

The query validation checks whether user queries are valid. If checking results are invalid, the query validation will return error messages. The validation checking includes syntax checking and semantics checking without reasoning. The syntax checking checks whether a given query has incorrect syntax. The semantics checking checks whether a given query could be reasoned in the selected assembled modular ontology. If the given query does not belong to the supported queries by the system, the query will be rejected.

5.1.3.4 Query Reasoning

The query reasoning function of the system reasons with valid queries, and returns reasoning results to queriers. The reasoning includes the concept
reasoning, negation reasoning, conjunction reasoning, disjunction reasoning, subsumption reasoning, role reasoning, and instance checking reasoning by the reasoning algorithms described in chapter 3 and chapter 4.

5.2 System Architecture

The distributed modular ontology reasoning system supports users to provide ontology modules and interfaces, create new modular ontologies, issue queries to modular ontologies, and receive reasoning results. The system consists of one server and multiple local nodes as shown in Figure 5.5.

Figure 5.5: An architecture of distributed modular ontology reasoning system

The server mainly manages the databases, and the local nodes contain local ontology modules and process distributed reasoning. In the architecture shown in Figure 5.5, each of the local nodes can retrieve information from the
server for reasoning about queries. The local nodes also need to communicate with the other related local nodes in order to perform distributed reasoning. The server and local nodes shown in Figure 5.5 run server software and local software respectively. All of local nodes run same software in the distributed modular ontology reasoning system.

Figure 5.6 shows the architecture of the server software system and the local software system. The whole system can be used by the three actors, including the ontology module interface providers, ontology creators, and ontology queriers.

Figure 5.6: The architecture for the server software system and local software system
As shown in Figure 5.6, the ontology module interface providers use the server software system to provide ontology interfaces and information of ontology modules to the system. The ontology creators use the server software system to create assembled modular ontologies from ontology interfaces and modules in the system. The ontology queriers use the local software system to issue queries and receive reasoning results by distributed reasoning. In the following, we will introduce the components of these two software systems in more detail.

5.2.1 Server Software System

The server software system supports the ontology module interface providers to manage the information databases, including searching, retrieving, adding and deleting ontology interfaces and information of ontology modules. The server software system also provides information in the databases to the local software systems and other components. It also supports the ontology creators to create modular ontologies. The server software system contains information management subsystem, modular ontology creation subsystem, and communication component.
5.2.1.1 Information Management Subsystem

The information management subsystem manages the ontology interface database, ontology module information database, and assembled modular ontology information database in the system. The information management subsystem also supports the ontology module interface providers to manage ontology interfaces and information of ontology modules, and assists other components to retrieve information from the databases.

The information management subsystem contains ProviderUserInterface, OntologyInterfaceDatabase, OntologyModuleInformationDatabase, and AssembledModularOntologyInformationDatabase.

ProviderUserInterface is a user interface for the ontology module interface providers. The providers can use ProviderUserInterface to search, add, and delete information in the OntologyModuleInformationDatabase and OntologyInterfaceDatabase.

OntologyInterfaceDatabase stores all ontology interfaces and their associated information, including interface name, description, and interface itself in form of OWL TBox.

OntologyModuleInformationDatabase stores information of ontology modules, including module name, description, location, exported concept names, and utilized and realized interfaces by the module.
AssembledModularOntologyInformationDatabase stores information of assembled modular ontologies, including assembled ontology name, description, IDs of constituted modules and their utilized and realized interfaces, and relations between constituted modules through interfaces.

5.2.1.2 Modular Ontology Creation Subsystem

The modular ontology creation subsystem supports the ontology creators to create or assemble modular ontologies by using information of ontology modules and interfaces in the information management subsystem.

CreatorUserInterface supports the ontology creators to retrieve and display information of ontology modules and interfaces in the information management subsystem, and select their required modules and interfaces to create assembled modular ontologies.

ModularOntologyVerification checks whether a newly created modular ontology from the ontology creator is complete and consistent and has all necessary ontology modules and interfaces. If the newly created modular ontology satisfies the conditions of completeness and consistency, ModularOntologyVerification will save the information of the newly created
modular ontology into the assembled modular ontology information database in the information management subsystem. Otherwise, ModularOntologyVerification will only return an error message to the creator.

5.2.1.3 Communication Component

The communication component has system interfaces for local software systems to retrieve information on interfaces, modules, assembled modular ontologies from the server software system.

5.2.2 Local Software System

The local software system supports the ontology queriers to issue queries and receive results after reasoning. The local software system needs to retrieve information from the server software system, and could need to communicate with other local software systems to perform query reasoning procedures. The local software system consists of query processing component, ontology reasoning component, knowledge base, and communication component.
5.2.2.1 Query Processing Component

The query processing component mainly supports the ontology queriers to issue queries and view their reasoning results through the user interface. This component also supports to check and process the issued queries. The query processing component contains *QuerierUserInterface* and *QueryHandling* sub-components.

*QuerierUserInterface* allows the ontology queriers to create system-supported queries, and displays whether the created queries are valid. *QuerierUserInterface* also shows reasoning results of valid queries to the ontology queriers.

*QueryHandling* checks whether user queries are valid according to the rules of the supported system queries. *QueryHandling* also processes queries by applying suitable reasoning algorithms implemented in the ontology reasoning component of this and other local software systems.

5.2.2.2 Ontology Reasoning Component

The ontology reasoning component of the local software system is utilized to help remote invocations of reasoning components on other local software systems, and reason with processed queries with an ontology module on the local software system.
The ontology reasoning component contains \textit{DMORAI} (see Glossary 3.) and \textit{StandardOntologyReasoner} sub-components.

\textit{DMORAI} implements the reasoning algorithms described in chapter 3 and chapter 4, including the concept reasoning, negation reasoning, conjunction reasoning, disjunction reasoning, subsumption reasoning, role reasoning, and instance checking reasoning algorithms.

\textit{DMORAI} uses \textit{StandardOntologyReasoner}. In this system, we use the Pellet reasoning engine as \textit{StandardOntologyReasoner} to implement functions of \textit{DMORAI}.

5.2.2.3 Knowledge Base

The knowledge base on a local software system consists of one or more ontology modules allocated to this local software system. An ontology module can only be reasoned on its own local software system. A local ontology module can only be accessed by the ontology queriers and components of the same local software system, and components and users of other local software systems have no access to this module.
5.2.2.4 Communication Component

The communication component supports a local software system to communicate with the server software system and other local software systems. It has system interfaces for retrieving information from the server software system and distributed remote reasoning invocations of other local software systems.

5.3 Functionality Realization

In this section, we will introduce functionality realization of the distributed modular ontology reasoning system, including information management, modular ontology creation, and query reasoning.

5.3.1 Information Management Realization

The information management subsystem manages ontology interfaces, information of ontology modules and assembled modular ontologies in the server software system. Its main functional execution involves with a series of database operations, including searching, retrieving, adding and deleting information in databases \textit{AssembledModularOntologyInformationDatabase}, \textit{OntologyInterfaceDatabase}, and \textit{OntologyModuleInformationDatabase}. 
The information management subsystem also supports the modular ontology creation subsystem to add information of created assembled modular ontologies into \textit{AssembledModularOntologyInformationDatabase}, and supports other components to search and retrieve information from the database.

An ontology interface provider can directly search, retrieve and add information in \textit{OntologyInterfaceDatabase}. Meanwhile, the information management subsystem supports components of other subsystems to search and retrieve ontology interfaces in the database.

The ontology module providers and other components can also directly search, retrieve information in \textit{OntologyModuleInformationDatabase} through the information management subsystem.

For \textit{OntologyModuleInformationDatabase}, the adding ontology module operation from an ontology module provider is different. It needs a series of operations to complete the whole procedure as shown in Figure 5.7.

As shown in Figure 5.7, when an ontology module provider provides information of an ontology module in \textit{ProviderUserInterface}, \textit{ProviderUserInterface} begins to check whether there are all related interfaces of this module given in the module information in \textit{OntologyInterfaceDatabase}.  

111
Figure 5.7: A sequence diagram of adding module information in *OntologyModuleInformationDatabase*
The related interfaces include realized interfaces and utilized interfaces of a
given module. If the newly added module realizes certain interfaces, then
\textit{OntologyInterfaceDatabase} should contain these realized interfaces. If the
newly added module utilizes certain interfaces, \textit{OntologyInterfaceDatabase}
should contain these utilized interfaces.

If \textit{OntologyInterfaceDatabase} does not contain some related interfaces, then
\textit{ProviderUserInterface} will return an error message to the module provider
and stop the adding procedure.

If \textit{OntologyInterfaceDatabase} has all related interfaces of a newly added
module, then \textit{ProviderUserInterface} will get the related interfaces from the
database and check consistency of the interfaces with the new module. If
the checking result is inconsistent, \textit{ProviderUserInterface} will return an
error message to the provider and stop the adding procedure. If the
checking result is consistent, \textit{ProviderUserInterface} will add the
information of the new module into \textit{OntologyModuleInformationDatabase}.

\section*{5.3.2 Modular Ontology Creation Realization}

The modular ontology creation subsystem supports the ontology creators to
create or assemble modular ontologies by using ontology interface and module
information in the information management subsystem. Its main functional
execution involves \textit{CreatorUserInterface}, \textit{ModularOntologyVerification}, and
accessing to all the databases in the information management subsystem. In the following, Figure 5.8 shows a sequence diagram of creating a modular ontology.

![Figure 5.8: A sequence diagram of creating a modular ontology](image)

As shown in Figure 5.8, an ontology creator firstly gets ontology module and interface information from \textit{OntologyModuleInformationDatabase} and \textit{OntologyInterfaceDatabase} through \textit{CreatorUserInterface}. According to the information, the creator selects required modules and interfaces to create an assembled modular ontology. After this assembling procedure, \textit{CreatorUserInterface} sends the created modular ontology to
ModularOntologyVerification to check whether the modular ontology is complete and consistent.

ModularOntologyVerification uses the interface and module information of an assembled modular ontology given by CreatorUserInterface to compute whether the ontology contains all necessary modules and interfaces.

If in the newly created modular ontology, each of utilizer modules has utilized interfaces, and each of utilized interfaces has realizer modules which realize the interface in the ontology, then the newly assembled ontology is considered to have all necessary modules and interfaces. Then, ModularOntologyVerification continues to check consistency.

If any of utilizer modules has no utilized interfaces, or any of utilized interfaces has no realizer module in the created modular ontology, then this newly created modular ontology will be incomplete. Then, ModularOntologyVerification returns an error message to the creator and CreatorUserInterface.

When the assembled modular ontology is complete after checking, ModularOntologyVerification will check consistency of the newly created assembled ontology. Only if the checking result is consistent, ModularOntologyVerification will add information of the new created modular ontology into AssembledModularOntologyInformationDatabase through the information management subsystem.
The consistency checking of assembled ontologies will be future work. In this thesis, we only check whether a newly created modular ontology contains all necessary modules and interfaces. If a newly assembled ontology satisfies this condition, \textit{ModularOntologyVerification} will directly add information of this ontology into \textit{AssembledModularOntologyInformationDatabase}.

### 5.3.3 Query Reasoning Realization

The query reasoning of local software systems supports the ontology queriers to build queries to assembled modular ontologies and receive their results after reasoning. In more detail, a local software system mainly checks validation of the issued queries, processes the valid queries for reasoning, and invokes multiple local systems to distributedly reason with the queries.

Its main functional execution involves \textit{QuerierUserInterface}, \textit{QueryHandling}, \textit{DMORAI}, \textit{StandardOntologyReasoner}, local ontology modules, and \textit{Communication} of the local software systems. It also involves \textit{Communication} and the databases in the information management subsystem of the server software system. In the following, Figure 5.9 shows a sequence diagram of query reasoning.

As shown in Figure 5.9, an ontology querier can get information of an assembled modular ontology and supported system queries through
Figure 5.9: A sequence diagram of query reasoning
QuerierUserInterface. For getting such information, QuerierUserInterface gets information of an assembled modular ontology from AssembledModularOntologyInformationDatabase in the information management subsystem of the server software system, and also gets information of ontology modules and interfaces from OntologyInterfaceDatabase and OntologyModuleInformationDatabase in the information management subsystem of the server software system, and gets the involved ontology module on the local software system.

After selecting an assembled modular ontology, the ontology querier can build a query in QuerierUserInterface on the local software system based on the supported query types, exported concept names of all involved modules, involved ontology interfaces, and local ontology module on the same local software system of the querier.

Then, QuerierUserInterface sends the created query to QueryHandling on the local software system. QueryHandling checks whether the syntax and semantics of the query are valid. If the query is invalid, QueryHandling will return an error message to the querier and stop reasoning. If the checking result is satisfiable, QueryHandling will send the query to DMORAI on the same local software system.

Next, DMORAI on the local software system generates a map for distributed reasoning. This map provides information to distributedly invoke reasoning
functions on all related local software systems to reason about all related ontology modules with sub-queries on this and other local software systems.

After generating the map, DMORAI reasons with the processed query with the local module and related interfaces. Then, DMORAI invokes other local software systems to conduct the reasoning procedures until completing reasoning on all related local software systems. Before returning the final results to the ontology querier, DMORAI processes sub-results from all local software systems to compute final reasoning results.

5.4 Summary

In this chapter, we first introduced system functionality of the distributed modular ontology reasoning system, including information databases, distributed modular ontology creation, and user query processing. Then, we described the system architecture with the server software system and local software systems. Next, in the functionality realization section, we used the sequence diagrams to show realizations of the information management, modular ontology creation, and query reasoning.
Chapter 6

Case Study and Experiments

In this chapter, we introduce a case study on one distributed modular ontology with different query cases, reasoning executions of some queries, and experiments.

6.1 Publication Modular Ontology

In this section, we describe a distributed modular ontology case, called *Publication*, using modular ontology description logic [7] and SWOOP tools [15]. As an example of distributed modular ontologies, this ontology consists of two modules and one interface, which are *Author*, *Publisher* and *PI*. Figure 6.1 shows knowledge of them.
Figure 6.1: Publication modular ontology
As shown in Figure 6.1, utilizer module Author uses some knowledge from interface PI, and the realizer module Publisher realizes PI. In other words, Publisher provides some public knowledge to interface PI. In more detail, the realizer module Publisher describes publishers including their publications, employees, locations, etc. The utilizer module Author describes authors by using knowledge from interface PI, such as $PI : Publication$, $PI : Editor$, $PI : ContractContent$ and so on.

6.2 Reasoning Query Cases

The reasoning query has eleven different types according to the reasoning algorithms. The design of queries is based on the conditions of queries. In the following, we will present concept query cases from TBox and ABox.

6.2.1 Concept Cases

The concept cases for TBox reasoning have two situations. One is a concept defined in a module, and the other is a concept defined in an interface. The concept cases for ABox reasoning only have one situation that is the concept and individual are defined in two different modules. We illustrate examples of the concept query cases as shown in Table 6.1.
Table 6.1: The table of concept cases

<table>
<thead>
<tr>
<th>Case No.</th>
<th>Query</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>case1</td>
<td>Author</td>
<td>Whether Publication has Author</td>
</tr>
<tr>
<td>case2</td>
<td>Editor</td>
<td>Whether Publication has Editor</td>
</tr>
</tbody>
</table>
| case3    | AssisEditor(Mary)  | Open-world reasoning: whether Mary could be an instance of AssisEditor  
                      | Closed-world reasoning: whether Mary is an instance of AssisEditor         |

As shown in Table 6.1, case1 and case2 are TBox concept queries, and for checking satisifiability of concepts or whether Publication has the concepts. The concept Author in case1 describes persons who can publish written work, and is defined in the utilizer module Author. Editor in case2 describes persons who edit written work from Author, and involves interface PI and the realizer module Publisher.

For ABox concept query case3, the system will check whether Mary could be an instance of AssisEditor in Publication for open-world reasoning, and whether Mary is an instance of AssisEditor in Publication for closed-world reasoning. AssisEditor, as a concept, describes assistants of editors, and is defined in the realizer module Publisher. Mary is an individual, and is defined in the utilizer module Author.
6.2.2 Negated Concept Cases

The negated concept cases are similar to the above concept cases. A negated concept can be seen as a new concept to reason. In this section, we give one case of the negated concept for TBox reasoning, and one case for ABox reasoning as shown in Table 6.2.

Table 6.2: The table of negated concept cases

<table>
<thead>
<tr>
<th>Case No.</th>
<th>Query</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>case4</td>
<td>not Author</td>
<td>Whether Publication has (\neg\text{Author})</td>
</tr>
</tbody>
</table>
| case5    | not Author(John)    | Open-world reasoning: whether John could be an instance of \(\neg\text{Author}\)  
                            | Closed-world reasoning: whether John is an instance of \(\neg\text{Author}\) |

As shown in Table 6.2, case4 is a TBox query, and for checking satisfiability or whether Publication has \(\neg\text{Author}\). case5 is an ABox query, and for checking whether John could be an instance of \(\neg\text{Author}\) in Publication for open-world reasoning, and whether John is an instance of \(\neg\text{Author}\) in Publication for closed-world reasoning.

6.2.3 Disjunction Cases

In this section, we give the disjunction cases for TBox reasoning with two different situations, one involves two modules, and the other involves an interface and a module. The disjunction cases for ABox reasoning only have
one situation that the query involves two modules. We illustrate examples of the disjunctive query cases as shown in Table 6.3.

Table 6.3: The table of disjunction cases

<table>
<thead>
<tr>
<th>Case No.</th>
<th>Query</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>case6</td>
<td>AU:subEditor or PU:PublisherSignature</td>
<td>whether Publication has AU:subEditor ⊔ PU:PublisherSignature</td>
</tr>
<tr>
<td>case7</td>
<td>PI:BookNumber or PU:Person</td>
<td>whether Publication has PI:BookNumber ⊔ PU:Person</td>
</tr>
<tr>
<td>case8</td>
<td>AU:subEditor or PU:PublisherSignature (Peter)</td>
<td>Open-world reasoning: whether Peter could be an instance of AU:subEditor ⊔ PU:PublisherSignature \ Closed-world reasoning: whether Peter is an instance of AU:subEditor ⊔ PU:PublisherSignature</td>
</tr>
</tbody>
</table>

As shown in Table 6.3, case6 and case7 are TBox disjunctive queries. Both of the queries are for checking satisfiability or whether Publication has the disjunctive queries. The concept AU:subEditor in case6 describes editors working with authors. AU describes the concept defined in the utilizer module Author. The concept PU:PublisherSignature in case6 describes signatures of publishers for contracts. PU describes the concept defined in the realizer module Publisher. The concept PI:BookNumber in case7 describes book numbers of publications. PI describes the concept defined in the interface PI. The concept PU:Person in case7 describes persons, and is defined in the realizer module Publisher.

The ABox disjunctive query case8 consists of case6 and individual Peter. The system will check whether Peter could be an instance of case6 in

125
Publication for open-world reasoning, and whether Peter is an instance of case6 in Publication for closed-world reasoning.

### 6.2.4 Conjunction Cases

Similarly to the disjunction cases, the conjunction cases for TBox reasoning have two different situations, one involves two modules, and the other involves one interface and one module. The conjunction cases for ABox reasoning only have one situation that the query involves two modules. We illustrate examples of the conjunctive query cases as shown in Table 6.4.

<table>
<thead>
<tr>
<th>Case No.</th>
<th>Query</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>case9</td>
<td>PU:Publisher and AU:AuthorSignature</td>
<td>whether Publication has PU:Publisher (\sqcap) AU:AuthorSignature</td>
</tr>
<tr>
<td>case10</td>
<td>(not PU:Contract) and PI:Payment</td>
<td>whether Publication has (\neg) PU:Contract (\sqcap) PI:Payment</td>
</tr>
</tbody>
</table>
| case11   | PU:Publisher and AU:AuthorSignature (Peter) | Open-world reasoning: whether Peter could be an instance of PU:Publisher \(\sqcap\) AU:AuthorSignature  
Closed-world reasoning: whether Peter is an instance of PU:Publisher \(\sqcap\) AU:AuthorSignature |

As shown in Table 6.4, case9 and case10 are TBox conjunctive queries, and for checking satisfiability or whether Publication has the conjunctive queries. The concept PU:Publisher in case9 describes publishers, and is defined in the realizer module Publisher. AU:AuthorSignature describes
signatures of authors for contracts, and is defined in the utilizer module Author. The negated concept “not PU:Contract” in case10 describes knowledge of the module Publisher that is not the concept Contract. Contract describes contracts between publishers and authors about publications. PI:Payment describes payments that is a sub-set of contracts, and is defined in PI.

The ABox query case11 consists of case9 and individual Peter. The system will check whether Peter could be an instance of case9 in Publication for open-world reasoning, and whether Peter is an instance of case9 in Publication for closed-world reasoning.

6.2.5 Subsumption Cases

The subsumption cases are similar to the conjunction cases. As mentioned before, a subsumption query can be seen as conjunctive queries to reason. There is one case of the subsumption queries for TBox reasoning as shown in Table 6.5.

<table>
<thead>
<tr>
<th>Case No.</th>
<th>Query</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>case12</td>
<td>AU:AuthorSignature</td>
<td>whether AU:AuthorSignature is subsumed by PU:Contract</td>
</tr>
</tbody>
</table>

As shown in Table 6.5, case12 is a subsumption TBox query, and for
checking whether \( AU : AuthorSignature \) is subsumed by \( PU : Contract \) in 
\textit{Publication}. The involved concepts of case12 are respectively defined in the utilizer module \textit{Author} and realizer module \textit{Publisher}.

\section*{6.2.6 Role Cases}

Similarly to the conjunction cases, there are two role cases for TBox reasoning, one involves two modules, and the other involves one interface and one module. The role cases for ABox reasoning only have one situation that the query involves two modules. We illustrate examples of the role query cases as shown in Table 6.6.

Table 6.6: The table of role cases

<table>
<thead>
<tr>
<th>Case No.</th>
<th>Query</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>case13</td>
<td>( AU:SignBookContract ) ( (AU:Author, ) ( PU:PublisherSignature) )</td>
<td>whether ( AU:Author ) is a subset of whom can sign book contracts with ( PU:PublisherSignature )</td>
</tr>
<tr>
<td>case14</td>
<td>( PU:HasPublisher ) ( (PU:ContemporaryLiterature, ) ( PI:Publisher) )</td>
<td>whether ( PU:ContemporaryLiterature ) is a subset of what have the relation HasPublisher with ( PI:Publisher )</td>
</tr>
<tr>
<td>case15</td>
<td>( PI:HasPublisher ) ( (Annie, ) ( NewsPress) )</td>
<td>whether there could be the relation ( PI:HasPublisher ) between Annie and NewsPress</td>
</tr>
</tbody>
</table>

As shown in Table 6.6, \textit{case13} and \textit{case14} are TBox role queries. Both of the queries are for checking satisfiability. The involved concepts in \textit{case13} are respectively defined in the utilizer module \textit{Author} and realizer module \textit{Publisher}. The role \textit{AU:SignBookContract} in \textit{case13} describes that authors
sign book contracts with publishers. The query of case13 means that \textit{AU:Author} is a subset of whom can sign book contracts with \textit{PU:PublisherSignature}.

The concept \textit{PU:ContemporaryLiterature} in case14 describes contemporary literatures of publications, and is defined in the realizer module \textit{Publisher}. The concept \textit{PI:Publisher} is defined in the interface \textit{PI}. The role \textit{PU:HasPublisher} describes that there is a \textit{HasPublisher} relation between publications and publishers, and is defined in the realizer module \textit{Publisher}. The query of case14 means that \textit{PU:ContemporaryLiterature} is a subset of what have the relation \textit{HasPublisher} with \textit{PU:PublisherSignature}.

For ABox role query case15, the system checks whether there could be the relation \textit{PI:HasPublisher} between \textit{Annie} and \textit{NewsPress} in \textit{Publication} for open-world reasoning. The role, \textit{PI:HasPublisher}, in case15 is defined in the interface \textit{PI}. The two individuals, \textit{NewsPress} and \textit{Annie}, are defined in the realizer module \textit{Publisher} and the utilizer module \textit{Author} respectively. \textit{Annie} is an instance of \textit{Author}. \textit{NewsPress} is an instance of \textit{Publisher}. 

129
6.3 Query Reasoning Examples

In this section, we will explain several representative query executions of the query cases given in section 6.2, including one case of the conjunctive queries for TBox reasoning and one case of the concept queries for ABox reasoning.

6.3.1 TBox Reasoning for the Conjunctive Query

In the following, we will use the conjunctive query case $PU:Publisher \sqcap AU:AuthorSignature$ to show the reasoning process.

This case involves the realizer module $Publisher$, utilizor module $Author$ and interface $PI$, where $Publisher$ and $Author$ are respectively located on two local software systems, called $P$ and $A$. The concept $Publisher$ is defined in module $Publisher$ located on $P$, and the concept $AuthorSignature$ is defined in module $Author$ located on $A$. The reasoning process is shown in Figure 6.2.

As shown in Figure 6.2, after the query is issued on local system $A$, $DMORAI$ of the local software system $A$ invokes the generating map function to compute the map that indicates to invoke reasoning functions in $P$ and $A$. 
Figure 6.2: The activity diagram of the conjunction reasoning
According to the map, DMORAI of A invokes its satisfiability checking function for the concept AuthorSignature, and the searching super-concepts reasoning function for AuthorSignature in the utilizer module Author and interface PI. Then, this DMORAI records the satisfiability of AuthorSignature, and computed super-concepts ContractContent and Thing of AuthorSignature in interface PI.

Next, DMORAI of A invokes the conjunction reasoning function with the computed super-concepts and concept Publisher in the DMORAI on the local software system P according to the generated map. The conjunction reasoning function in DMORAI of P computes each of ContractContent and Thing conjunction with Publisher in module Publisher and interface PI on local system P. After that, DMORAI of P returns the computed satisfiability to DMORAI of A.

Then, DMORAI of A processes all the computed satisfiability from checking AuthorSignature on A, and the conjunction reasoning function on Publisher on P. If there is one unsatisfiable of the computed satisfiability, DMORAI of A will return unsatisfiable to the querier. If all of satisfiability results are satisfiable, then DMORAI of A will return satisfiable to the querier.
6.3.2 ABox Reasoning for the Concept Query

We will use the concept query case \textit{AssisEditor}(Mary) to show the ABox open-world reasoning process.

This case involves the realizer module \textit{Publisher}, utilizer module \textit{Author}, and interface \textit{PI}. Similarly, \textit{Publisher} and \textit{Author} are respectively located on two local software systems, called \textit{P} and \textit{A}. The concept \textit{AssisEditor} is defined in the realizer module \textit{Publisher}, and individual \textit{Mary} is defined in the utilizer module \textit{Author}. The reasoning process is shown in Figure 6.3.

Figure 6.3: The activity diagram of the instance concept checking
As shown in Figure 6.3, after the query is issued on the local software system $A$, $DMORAI$ of $A$ invokes the generating map function to compute the map that indicates to invoke reasoning functions on $P$ and $A$.

According to the map, $DMORAI$ of $A$ invokes the satisfiability checking function with the concept $AssisEditor$ in $DMORAI$ on the local software system $P$. The satisfiability checking function in $DMORAI$ of $P$ computes the satisfiability of $AssisEditor$ in the realizer module $Publisher$ and interface $PI$ on the local software system $P$. If the computed satisfiability of $AssisEditor$ is unsatisfiable, $DMORAI$ of $A$ will return inconsistent and stop reasoning. Otherwise, $DMORAI$ of $A$ will invoke the searching super-concepts reasoning function in $DMORAI$ on the local software system $P$ for the concept $AssisEditor$.

The searching super-concepts reasoning function in $DMORAI$ of $P$ computes super-concepts of $AssisEditor$ in module $Publisher$ and interface $PI$ on the local software system $P$. $DMORAI$ of $A$ records the computed super-concepts $Editor$ and $Thing$ of $AssisEditor$ in interface $PI$ from $P$. Next, $DMORAI$ of $A$ invokes the consistency checking function to compute each of $Editor$ and $Thing$ with individual $Mary$ in the module $Author$ and interface $PI$ on the local software system $A$. If there is one inconsistent of the computed consistency, $DMORAI$ of $A$ will return inconsistent to the querier. Otherwise, $DMORAI$ will return consistent.
6.4 Experiments

In this section, we show all the test results of query cases of section 6.2. In order to validate and evaluate our distributed reasoning system, we integrate modules of the modular ontology Publication to a monolithic ontology OnePublication. We use a standard ontology reasoner to reason with the monolithic ontology OnePublication in order to compare query reasoning results. In the following, we illustrate the testing data of all the query cases as shown in Table 6.7.

Table 6.7: Testing results of the query cases

<table>
<thead>
<tr>
<th>Case No.</th>
<th>Distributed Reasoning</th>
<th>Monolithic Reasoning</th>
<th>Reasoning Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>case1</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case2</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case3</td>
<td>consistent</td>
<td>consistent</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case3</td>
<td>inconsistent</td>
<td>inconsistent</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case4</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case5</td>
<td>inconsistent</td>
<td>inconsistent</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case6</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case7</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case8</td>
<td>consistent</td>
<td>consistent</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case8</td>
<td>inconsistent</td>
<td>inconsistent</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case9</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case10</td>
<td>unsatisfiable</td>
<td>unsatisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case11</td>
<td>consistent</td>
<td>consistent</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case11</td>
<td>inconsistent</td>
<td>inconsistent</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case12</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case13</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case14</td>
<td>satisfiable</td>
<td>satisfiable</td>
<td>TBox reasoning</td>
</tr>
<tr>
<td>case15</td>
<td>consistent</td>
<td>consistent</td>
<td>TBox reasoning</td>
</tr>
</tbody>
</table>

The Distributed Reasoning column in Table 6.7 shows the reasoning results for modular ontology Publication using the distributed reasoning
algorithms of chapter 3 in this thesis. The Monolithic Reasoning column shows the reasoning results for monolithic ontology OnePublication using the standard ontology reasoner Pellet. The Reasoning Types column shows the reasoning types, like TBox reasoning, open-world ABox reasoning, or closed-world ABox reasoning. As shown in Table 6.7, for each of the query cases, the results shown in the two columns are same. These results validate the correctness of distributed reasoning algorithms given in the thesis.

6.5 Summary

In this chapter, we introduced a modular ontology Publication for testing the distributed reasoning algorithms. We described several different types of query cases and examples. We chose two representative query cases to describe their reasoning executions. We showed the experimental test results for the query cases.
Chapter 7

Conclusion

7.1 Summary

This thesis describes the design and implementation of distributed modular ontology reasoning, including reasoning for concept, negation, conjunction, disjunction, subumption, role reasoning, and instance checking for concept, negation, conjunction, disjunction, and role.

We present the distributed reasoning algorithm designs in chapter 3 and chapter 4. In these two chapters, we mainly introduce the reasoning algorithms for processing queries involving distributed ontology modules on different local software systems. In chapter 3, we describe the reasoning algorithms for simple distributed modular ontologies with only one-to-one...
relations between constituted utilizer modules and realizer modules. In chapter 4, we describe the extended reasoning algorithms for general distributed modular ontologies with one-to-many and many-to-many relations between constituted ontology modules.

In chapter 5, we describe the design and implementation of the distributed modular ontology reasoning system, including system functionality, system architecture, and functionality realization. The system functionality introduces main functions of the system. The system architecture introduces the three system actors and components of the system to support functions of the system. The functionality realization describes the three main execution processes of the system, including the information management, assembled modular ontology creation, and query reasoning.

In chapter 6, we show experiments with diverse query cases for testing reasoning algorithms of chapter 3. We explain query executions of two query cases from reasoning and invocation aspects in more detail. In order to evaluate the performance and validate the correctness of the distributed reasoning, we use an equivalent monolithic ontology with standard ontology reasoning to compare with a distributed modular ontology with distributed reasoning. By testing the same query cases by monolithic and distributed reasoning, we show that the distributed reasoning of chapter 3 has correct reasoning results.
7.2 Contribution

In this thesis, our designed distributed modular ontology reasoning algorithms provide an approach to distributed ontology reasoning. We also design and implement the distributed modular ontology reasoning system for users to effectively and practically reasoning modular-ontology based distributed ontologies.

7.3 Future Work

In future work, we plan to improve performance of our system and the reasoning algorithms with real-world ontologies. We will add agents to the system transfer data for more efficiency and flexibility. We will improve the reasoning algorithms by adding qualified number restrictions and some property restrictions. In future work, we will also utilize complex and real modular ontologies to improve our system implementation.


Glossary

ABox (Assertion) are TBox-compliant statements about individuals belonging to those concepts. [32]

DL Description logic.

DMORAI Distributed modular ontology reasoning algorithm implementation.

I-OWL Interface-based Web Ontology Language.

TBox (Terminology) describe a conceptualization, a set of concepts and properties for these concepts. [33]
Vita

Candidate’s full name: Li Ji
University attended:
September 2015 - July 2018
Faulty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada

September 2009 - May 2013
Bachelor of Software Engineering
Software Engineering
Southeast University
Nanjing, Jiangsu, China

Publications:

L. Ji, W. Du: Distributed Modular Ontology Reasoning. (submitted)

Conference Presentations: none