

ScholarWorks@GSU

Maintaining Integrity Constraints in Semantic Web

Authors	Fang, Ming
Citation	Fang, Ming (2013). Maintaining Integrity Constraints in Semantic Web. Dissertation, Georgia State University. https://doi.org/10.57709/4003155
DOI	https://doi.org/10.57709/4003155
Download date	2026-04-10 23:35:01
Link to Item	https://hdl.handle.net/20.500.14694/4062

MAINTAINING INTEGRITY CONSTRAINTS IN SEMANTIC WEB

by

MING FANG

Under the Direction of Rajshekhar Sunderraman

ABSTRACT

As an expressive knowledge representation language for Semantic Web, Web Ontology Language (OWL) plays an important role in areas like science and commerce. The problem of maintaining integrity constraints arises because OWL employs the Open World Assumption (OWA) as well as the Non-Unique Name Assumption (NUNA). These assumptions are typically suitable for representing knowledge distributed across the Web, where the complete knowledge about a domain cannot be assumed, but make it challenging to use OWL itself for closed world integrity constraint validation. Integrity constraints (ICs) on ontologies have to be enforced; otherwise conflicting results would be derivable from the same knowledge base (KB). The current trends of incorporating ICs into OWL are based on its query language SPARQL, alternative se-

mantics, or logic programming. These methods usually suffer from limited types of constraints they can handle, and/or inherited computational expensiveness.

This dissertation presents a comprehensive and efficient approach to maintaining integrity constraints. The design enforces data consistency throughout the OWL life cycle, including the processes of OWL generation, maintenance, and interactions with other ontologies. For OWL generation, the Paraconsistent model is used to maintain integrity constraints during the relational database to OWL translation process. Then a new rule-based language with set extension is introduced as a platform to allow users to specify constraints, along with a demonstration of 18 commonly used constraints written in this language. In addition, a new constraint maintenance system, called Jena2Drools, is proposed and implemented, to show its effectiveness and efficiency. To further handle inconsistencies among multiple distributed ontologies, this work constructs a framework to break down global constraints into several sub-constraints for efficient parallel validation.

INDEX WORDS: Semantic web, OWL, Ontology, Logic programming, Integrity constraints

MAINTAINING INTEGRITY CONSTRAINTS IN SEMANTIC WEB

by

MING FANG

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2013

Copyright by
Ming Fang
2013

MAINTAINING INTEGRITY CONSTRAINTS IN SEMANTIC WEB

by

MING FANG

Committee Chair: Rajshekhar Sunderraman

Committee: Anu Bourgeois

Yanqing Zhang

Yichuan Zhao

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2013

ACKNOWLEDGEMENTS

Firstly, my specific thanks go to my advisor, Dr. Rajshekhar Sunderraman, for his guidance and precise advisement during the process of my PhD dissertation. The dissertation would not have been possible without his help.

Secondly, I would like to thank my committee members, Dr. Anu Bourgeois, Dr. Yanqing Zhang, and Dr. Yichuan Zhao for their well-appreciated support and assistance.

Finally, I want to thank my family and friends. They have been the greatest support to me during this journey and have always been there for me.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
LIST OF TABLES	viii
LIST OF FIGURES	ix
CHAPTER 1 INTRODUCTION.....	1
1.1 Background	1
1.2 Problem Statement	3
1.3 Organization.....	4
CHAPTER 2 BACKGROUND.....	6
2.1 Semantic Web and Ontology	6
<i>2.1.1 Semantic Web</i>	<i>6</i>
<i>2.1.2 Ontology.....</i>	<i>11</i>
2.2 OWL and Description Logics	12
<i>2.2.1 OWL.....</i>	<i>12</i>
<i>2.2.2 Description Logics.....</i>	<i>15</i>
2.3 Logic Programming and Logic-based Data-Language	21
<i>2.3.1 Logic Programming</i>	<i>21</i>
<i>2.3.2 Logic-based Data-Language.....</i>	<i>25</i>
2.4 The Paraconsistent Model.....	27
<i>2.4.1 Paraconsistent Relations.....</i>	<i>28</i>

2.4.2	<i>Algebraic Operators on Paraconsistent Relations</i>	29
CHAPTER 3 MAINTAINING INTEGRITY CONSTRAINTS IN RELATIONAL		
TO OWL TRANSLATION.....		
		31
3.1	Motivations	31
3.2	The paraconsistent-based translation	32
3.2.1	<i>Foreign Key Constraint</i>	33
3.2.2	<i>Total Participation Constraint</i>	36
3.3	OWL-style constraints	37
3.3.1	<i>Redundant Individual Type Constraint</i>	37
3.3.2	<i>Specific Individual Type Constraint</i>	37
CHAPTER 4 MAINTAINING INTEGRITY CONSTRAINTS IN A SINGLE OWL		
.....		
		40
4.1	Motivations	40
4.2	Syntax	41
4.2.1	<i>Rule Logic (RL) syntax</i>	41
4.2.2	<i>Description Logic (DL) syntax</i>	43
4.2.3	<i>Rule-based DL language with set extension knowledge base</i>	44
4.3	Semantics	44
4.4	An Example	45
4.5	Constraint List	46

CHAPTER 5 A HYBRID APPROACH TO CONSTRAINT REASONING IN BIO- ONTOLOGIES	50
5.1 Motivations.....	50
5.2 Proposed Architecture	52
<i>5.2.1 Loosely coupled hybrid architecture.....</i>	<i>53</i>
<i>5.2.2 Examples of Constraint Checking.....</i>	<i>59</i>
CHAPTER 6 IMPLEMENTATION AND EVALUATION OF AN IMPROVED JENA2DROOLS FOR GENERAL-PURPOSE CONSTRAINT CHECKING	62
6.1 Jena2Drools 2.0.....	62
6.2 Evaluation.....	65
6.3 Performance Analysis.....	70
CHAPTER 7 MAINTAINING INTEGRITY CONSTRAINTS IN MULTIPLE DISTRIBUTED OWLS.....	77
7.1 Motivations.....	77
<i>7.1.1 Biomedical ontologies are heavily interconnected.....</i>	<i>77</i>
<i>7.1.2 Biomedical ontologies are updated frequently.....</i>	<i>79</i>
7.2 A framework for distributed ontologies	80
CHAPTER 8 CONCLUSION AND FUTURE WORK.....	94
REFERENCES.....	96

LIST OF TABLES

Table 5.1 Statistics for sample bio-ontologies	52
Table 7.1 Contents of Patient Ontology, Treatment Ontology, and Claim Ontology.....	81
Table 7.2 The updated content of Claim Ontology	86
Table 7.3 Constraint-source table.....	87
Table 7.4 Constraint planning table	88
Table 7.5 Optimized Constraint Planning table	92

LIST OF FIGURES

Figure 1.1 An OWL evolution process. (1) RDB to OWL translation; (2) Update on a single OWL ontology; (3) Update on multiple distributed but interdependent OWL ontologies. Integrity constraints have to be maintained during these processes.....	3
Figure 2.1 The Semantic Web Stack (Adapted from Tim Berners-Lee’s slides at http://www.w3.org/2002/Talks/04-sweb/slide12-0.html).....	7
Figure 2.2 A RDF graph describing Eric Miller (from RDF Primer at http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#figure1).....	9
Figure 3.1 A small relational database with 3 foreign keys: (1) <i>student</i> pointing to <i>people</i> ; (2) <i>student</i> column in <i>take</i> pointing to <i>student</i> ; (3) <i>course</i> column in <i>take</i> pointing to <i>course</i>	33
Figure 3.2 Negative data tables.....	33
Figure 5.1 Lymphocytes and its relationship with other classes in OMIT	51
Figure 5.2 Loosely coupled hybrid architecture	53
Figure 6.1 LUBM datasets statistics	66
Figure 6.2 RDF load, RDF instance translation, and knowledgebase preparation and insertion time.	70
Figure 6.3 Comparing validation time and total running time for integrity constraint 1. 72	
Figure 6.4 Comparing validation time and total running time for integrity constraint 4. 72	
Figure 6.5 Comparing validation time and total running time for integrity constraint 5. 73	
Figure 6.6 Validation time comparison between finding a single violation and finding all violations, when load, translation, and preparation time is not taken into account.	74

Figure 6.7 Validation time comparison between finding a single violation and finding all violations, when load, translation, and preparation time is taken into account.	75
Figure 7.1 Interdependency between Biomedical Ontologies	78
Figure 7.2 Biomedical Ontologies update frequency (updates/year).....	79
Figure 7.3 Biomedical Ontologies update scale (lines/update).....	80
Figure 7.4 Ontology Schema and Constraint Checking Architecture.....	81
Figure 7.5 Internal Architecture of Constraint Checker	84
Figure 7.6 A Generalized Class Hierarchy for Patient Ontology	85

CHAPTER 1 INTRODUCTION

1.1 Background

The term Semantic Web ([BHL01]), coined by the inventor of the World Wide Web Tim Berners-Lee in 2001, refers to the web of linked data whose semantics can be understood by machines for further automatic process. In 2004, the World Wide Web Consortium (W3C) standardized OWL as the recommended language for modeling the Semantic Web. One of the challenges that associate with this new technology is maintaining integrity constraints and data consistency in Semantic Web. Complexity arises in detecting cases where instance data fail to meet the restrictions imposed by the integrity constraints, due to OWA and UNA. In OWA ([Ng05]), statements cannot be evaluated to be false if they are not explicitly stated in or inferred from the knowledge base. In NUNA ([RN03]), it is possible that two different identifiers refer to the same entity in the knowledge base.

Example 1.1 The following example in OWL Description Logic (DL) exemplify the integrity constraints issues in Semantic Web:

TBox:

$A \sqsubseteq B$

ABox:

$A(a)$

Note that even by explicitly denoting class A is subsumed by class B (i.e. all instances in class A are also in class B), a missing value of a in class B (i.e. $B(a)$) would not flag an inconsistency problem as expected in Closed World Assumption. Instead, new knowledge $B(a)$ is derived from the given information.

Several lines of approaches have been proposed to address this issue. They either integrate OWL with a different formalism such as rules or epistemic queries, or provide an alternative semantics in OWL. The rule-based approach ([EIL+04], [Mot07]) builds a hybrid knowledge base in which rules are responsible for imposing ICs on the OWL data. If the KB entails a certain rule predicate, then a violation to ICs exists. However, it is not intuitive to make two formalisms work together smoothly, and this approach may be computationally expensive when the data set gets huge, depending on rule constructs and its implementation. In addition, important aggregate types of constraints are hard, if possible, to express in this formalism. The epistemic query-based method ([CGL+07]) checks the satisfaction of ICs by asking queries against the KB. The results of these queries determine whether there is a violation or not. Unfortunately, the data complexity that inherits from this approach in expressive DLs still remains unknown. Most recently, an alternative semantics-based approach ([Tao10]) emerged as a different line of solution. In this approach, the semantics of OWL has been extended. Now OWL not only serves as an ontology modeling language, but also works as a native language to specify integrity constraints. Although the effort to provide a unified ontology and IC language has been very much appreciated, confusions in distinguishing these two may arise as well. In addition, non-traditional semantics can have interoperability issues when an application based on this approach interacts with other conventional applications. Also, the types of constraints this unified language is capable of expressing are bounded by the OWL formalism. Besides, none of these approaches consider constraints spanning multiple ontologies that are distributed and interdependent in nature.

1.2 Problem Statement

This dissertation focuses on maintaining the integrity constraints through the OWL life cycle, including the processes of OWL generation, maintenance, and interactions with other ontologies.

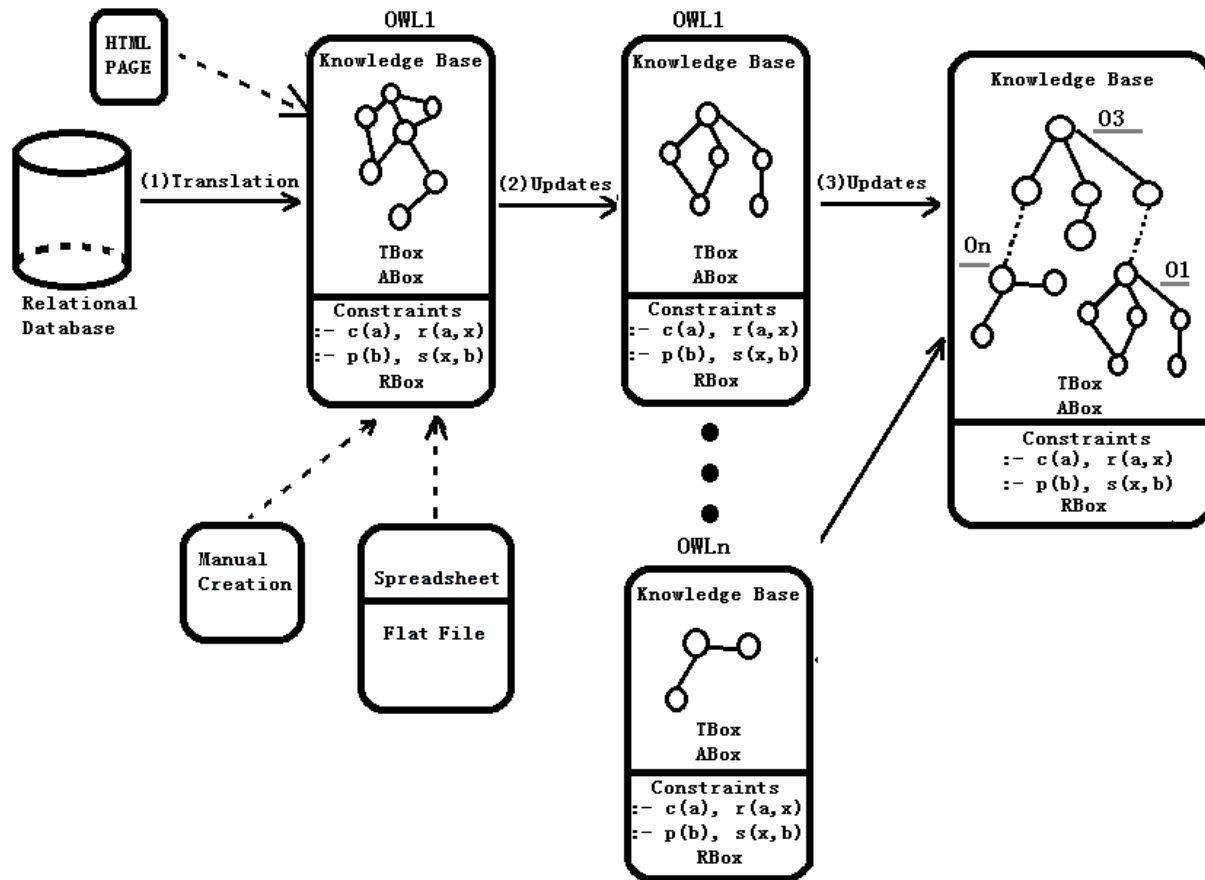


Figure 1.1 An OWL evolution process. (1) RDB to OWL translation; (2) Update on a single OWL ontology; (3) Update on multiple distributed but interdependent OWL ontologies. Integrity constraints have to be maintained during these processes.

As shown in figure 1.1, the goal of this dissertation consists of a series of sub-problems. The solution to each of the sub-problem itself requires careful consideration.

- Maintaining integrity constraints in relational to OWL translation: relational databases are expected to be an important source of OWL generation. However, most, if not all, of the vital integrity constraints on relational databases are lost in the current approaches to translating from RDB to OWL.
- Maintaining integrity constraints in a single OWL ontology: updates happen frequently in ontologies. Although current proposals can check if there is a violation to the enforced constraints whenever there is an update, they are limited by the number of types of constraints they can handle as well as the efficiency of running an inconsistency check.
- Maintaining global constraints that span multiple OWL ontologies: ontologies are distributed and interdependent in nature. One change that satisfies local constraints might trigger the violation of a global constraint. To the best of our knowledge, currently there is no integrity constraint enforcement method available at the global level.

The dissertation is intended to present practical and efficient solutions to above problems separately but under a uniform constraint-modeling framework. In this way, it is possible for an integral system to enforce the constraints throughout the OWL life cycle.

1.3 Organization

The dissertation is organized as follows. In chapter 2, we introduce the basics of Semantic Web and Ontology, OWL and Description Logics, Logic Programming and logic-based data-language, and finally the paraconsistent model. These backgrounds are essential for understanding the progress and its significance in this branch of Semantic Web research. Chapter 3 presents our approach to preserving critical integrity constraints during relational database to OWL translation, by using paraconsistent model as the underlying logic. Four common constraints, namely

foreign key constraints, total participation constraint, redundant individual type constraint, and specific individual type constraint, are explicitly encoded by our approach, to show the validity and effectiveness of the paraconsistent approach to constraint maintenance. Chapter 4 introduces Rule-based DL language with set extension, including its syntax, semantics, along with a demonstration of 18 commonly used constraints in this form. This language functions as the constraint language to allow specification of more types of constraints whose satisfaction is efficient to check. Chapter 5 examines characteristics of commonly used bio-ontologies. Then this chapter proposes a design of a hybrid architecture that integrates semantic knowledge base technologies and forward rule reasoning, called Jena2Drools, to better serve the purpose of maintaining integrity constraints in bio-ontologies. In chapter 6, an improved version of this hybrid system, Jena2Drools 2.0, can now take arbitrary legal Semantic Web data for general-purpose constraint checking. An evaluation as well as comparisons with the state-of-the-art system are also in this chapter. Chapter 7 takes the constraint checking to the global level. It provides the user with a framework that is capable of efficiently checking IC violation in parallel, by breaking down the global constraint into several sub-constraints that target at a specific site. In the last chapter, a general conclusion and a plan for future work are provided.

CHAPTER 2 BACKGROUND

2.1 Semantic Web and Ontology

2.1.1 *Semantic Web*

The term Semantic Web ([BHL01]), coined by the inventor of the World Wide Web Tim Berners-Lee in 2001, refers to the web of linked data whose semantics can be understood by machines for further automatic process. The Semantic Web inherits the expressive knowledge representation power from its predecessor Semantic Network ([Joh91]), and extends its syntactic and semantic interoperability and inference ability. Since then, numerous applications like FOAF ([BM06]), TrueKnowledge etc. have sprung rapidly in the domain of science and commerce. According to Swoogle ([DFJ+04]), a Google type of search engine for Semantic Web document, it has indexed 1.5 million Semantic Web documents of various forms including RDF, RDFS, and OWL, by the year 2006. Not only science communities like biomedical science and geoscience actively participate in Semantic Web development, but also industry leaders like Oracle, Vodafone, Amazon.com, Adobe, Yahoo and Google invest heavily in the smarter web technology. For example, Oracle developed the first RDF management system to support application integrations in areas of life sciences, enterprise applications and supply chain management. Oracle also extended the OWL support in this platform. As a leading telecommunication company in Europe, Vodafone took the initiatives to introduce RDF in describing ringtones, games, and pictures on their website, resulting a better user browsing experience and an increase in revenue.

The Semantic Web holds the mission to address the information-processing problem in the era of web information explosion. As of this writing, the WWW contains some 48 billions of web pages ([Kun2011]) whose contents are primarily in nature language. This enormous amount of human-readable data increases the difficulty for users to seek, access, utilize, and maintain

information by themselves. Users need help from machines to access the web content more intelligently and perform tasks as users demand. The Semantic Web emerged as an idea to enrich data, documents, applications, and other types of web resources with machine-understandable metadata about resources and how they are related to each other. In the Semantic Web, online resources are labeled and linked together in a meaningful way. By building a hierarchical semantic structure, the Semantic Web allows automated services to navigate through the machine-readable data for accurate search and filtering. Because the Semantic Web “understands” the content, one can also take the advantage of its reasoning capability to infer new knowledge from what has been explicitly expressed. Therefore, the Semantic Web dramatically enhances extensibility, visibility and inference ability during the knowledge sharing process, when compared with previous approaches.

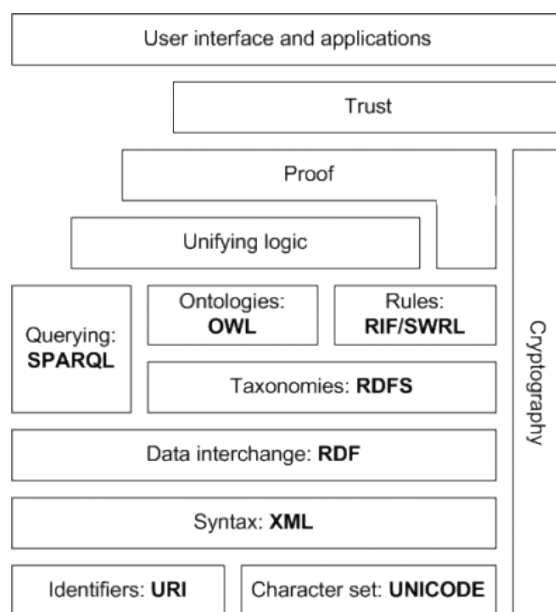


Figure 2.1 The Semantic Web Stack (Adapted from Tim Berners-Lee’s slides at <http://www.w3.org/2002/Talks/04-sweb/slide12-0.html>)

The Semantic Web is trying to achieve its goal by employing various layers of technologies, as shown in figure 2.1. Universal Resource Identifiers (URIs) are used to describe resources

and relationships, and thus comprise the vocabulary in the Semantic Web. The extensible markup language (XML) is a meta-language to define application-specific markup tags. It serves as a grammar to ensure that knowledge conforms to the XML syntax. RDF, RDFS, and OWL represent the evolution of the Semantic Web Ontology Languages.

- **RDF:** Resource Description Framework ([GJ04]), by its name, is a modeling language as well as a framework. RDF provides a reference model that is based on graphs. RDF quickly gained popularity since its release because of its simplicity. In the RDF model, a set of resources, denoted by URIs, constitute the universe of this model. A set of properties, which is essentially a group of binary predicates, is used to describe the relationships between resources. Hence, descriptions are usually in the triple-element form (i.e. subject-predicate-object). One important feature is that subject and object can be anonymous resources without names. This design assumes the knowledge about a certain domain is incomplete, in order to address the fact that the amount of information on the web is incredibly enormous. In RDF, it is possible to “allow anyone to make statements about any resource”, as part of its design goals. RDF is also equipped with its own standard query language called SPARQL, to satisfy the basic needs of retrieving information from RDF.

Example 2.1 The following code snippet from W3C describes a person named Eric Miller in RDF language that corresponds to the RDF graph in figure 2.2:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">
<contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
```

```

<contact:fullName>Eric Miller</contact:fullName>
<contact:mailbox rdf:resource="mailto:em@w3.org"/>
<contact:personalTitle>Dr.</contact:personalTitle>
contact:Person>
</rdf:RDF>

```



Figure 2.2 A RDF graph describing Eric Miller (from RDF Primer at <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/#figure1>)

- **RDFS**: RDF Schema ([Dan04]) is an augmentation to RDF to provide more expressive power for definition and classification. Some of the major additions include: (1) the ability to formally define *class* and *property*; (2) the ability to define class hierarchies and property hierarchies by making *subClassOf* and *subPropertyOf* statements about classes and properties; (3) the ability to specify the *domain* and *range* of a

property to restrict the types of resources that can participate in this property; (4) the ability to make *Datatype* statement to explicitly declare a resources as an instance of a class. RDF and RDFS together provide a light-weighted formalism to represent web resources, but they lack powerful inference ability compared to other knowledge representation languages.

- **OWL**: Web Ontology Language ([MG04]) further enriches RDF and RDFS with emphasis on the logical inference. OWL has been evolved from two independent contemporary researches, namely DARPA Agent Markup Language (DAML) and Ontology Inference Layer (OIL) ([FHH+01]), to become DAML+OIL ([HSH02]). W3C standardized OWL in 2004 as a recommendation, and later standardized OWL 2 in 2009. As a matter of fact, OWL is a family of languages whose members vary in expressive power and decidability. OWL will be discussed in great details in the next section.

Although the Semantic Web bears promising solutions to intelligently process large web data, it also faces many challenges. The vastness of the web content is unprecedented; any automated reasoning system will have to face such incredibly huge inputs. Impreciseness of defining concepts and properties like “popular” and “beautiful” is due to human subjectivity and lack of consensus. Security issues also arise from the openness and high accessibility inherited from the Semantic Web. And finally, the issue this dissertation is trying to address is inconsistency. Inconsistencies and violations to integrity constraints are unavoidable during the development of large ontologies, either by crafting from scratch or reusing ontologies from heterogeneous sources. The result is disastrous. By the Principle of Explosion, any conclusion is derivable from contradiction in logic! As such, conflicting and untrustworthy findings become useless to the us-

ers. The consistency of ontologies has to be enforced; otherwise the entire effort of building a machine-readable system is wasted in vain. Therefore, effective and efficient constraint maintenance mechanism is necessary whenever there is a change to ontologies.

2.1.2 *Ontology*

The relationship between the Semantic Web and ontologies are so close that ontologies themselves deserve a closer look at this section. As a matter of fact, ontologies are the backbone technology that facilitates the interconnections between resources in the Semantic Web. Recall from figure 2.1, W3C has standardized the OWL layer as the ontology language in the Semantic Web. Ontology originated from Greek words “onto” and “logia” that refer to the studies of being and existence in philosophy. According to [Gru93], ontology is a formal, explicit specification of a shared conceptualization. It possesses the ability to model “things” of the world in an abstract way, and the ability to define concepts and constraints in an explicit, formal, and machine-understandable way. In the field of computer science, ontologies are heavily used in artificial intelligence to promote knowledge sharing and reuse. It became popular in application areas such as natural language processing, knowledge representation and management, intelligent information integration, information retrieval, and electronic commerce. WordNet, as a well-known example, contains a thesaurus of over 100,000 terms. Ontologies enable a shared and common understanding in a specific domain between users and machines. This property of ontologies satisfies the exact need of a commonly agreed and explicitly specified infrastructure to support semantic interoperability in knowledge sharing activities between human and computers. However, ontology development has to emphasize on extensibility, visibility, and inference ability for the purpose of the Semantic Web ([DPD+05]). In short, ontology is the core of the Semantic Web.

2.2 OWL and Description Logics

2.2.1 OWL

According to a survey ([Car07]) done three years after the formal introduction of OWL, OWL topped the user adoption in semantic document development for the Semantic Web, among other 15 languages such as RDF, RDFS, Flogic, and SHOE, etc. Out of 627 respondents, more than 75% of ontologists have selected OWL to develop their ontologies. A quick search in Swoogle displayed roughly 100 thousand OWL files as of this writing. One reason for its popularity is its ability to formally describe complex concepts and relationships among concepts. More importantly, OWL provides a way to facilitate automated reasoning at both conceptual level and instance level. Although numerous ontologies, such as Infectious Disease Ontology, Chemical Information Ontology, are available in OWL form, there is still a huge demand for developing more OWL ontologies for various purposes.

Example 2.2 The following code snippet from W3C describes the domain of the property *hasBankAccount* can be either a *Person* or *Corporation*. Please note it is also a mixture of RDFS vocabulary and OWL vocabulary, as RDFS is not capable of describing disjunctions.

```
<owl:ObjectProperty rdf:ID="hasBankAccount">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Person"/>
        <owl:Class rdf:about="#Corporation"/>
      </owl:unionOf>
    </owl:Class>
```

</rdfs:domain>

</owl:ObjectProperty>

W3C further categorized OWL into three variants, namely OWL Lite, OWL DL, and OWL Full, with OWL Lite being the least expressive one, and OWL Full being the most expressive one. Since each variant strictly extends its simpler predecessor, the following statements hold:

- Every legal ontology in OWL Lite is also legal in OWL DL, and thus legal in OWL Full.
- Every legal ontology in OWL DL is also legal in OWL Full.

These three variants provide trade-offs between computational complexity and the expressive power of ontology constructs. Next, we will examine these three variants in great details.

- **OWL Lite:** As its name indicates, OWL Lite is the simplest variant in OWL family. OWL Lite disallows the usage of *owl:oneOf*, *owl:unionOf*, *owl:complementOf*, *owl:hasValue*, *owl:disjointWith*, and *owl:DataRange*. It also put restrictions on *owl:equivalentClass*, *owl:intersectionOf*, and *owl:allValuesFrom*, and much more. For a full list of OWL vocabulary and limitations of OWL Lite, please refer to ([MG04]). The reduction in expressivity results in a gain in efficiency of complete reasoners for OWL Lite. In a nutshell, OWL Lite is capable of modeling class and property hierarchies, but short of inference features. The design goal of such variant is to provide the OWL family with the interoperability with non-reasoning systems like RDFS systems and databases.

- **OWL DL:** OWL DL relaxes some of the OWL Lite restrictions by allowing conjunction, disjunction and negation in OWL DL constructs. As a matter of fact, OWL DL permits the entire OWL vocabulary but with some restrictions. An important inference mechanism, namely subsumption, is available as well. These features are carefully designed, as OWL DL is rooted in Description Logics ([BN02]) who themselves are a family of languages. Description Logics, which we shall see in details very shortly, focus on providing formal semantics within decidable inference ability. It has been shown ([HS02]) that, with the complexity between polynomial and exponential time, basic inference in most variants of Description Logics is decidable. OWL DL builds around Description Logics in such a way that OWL DL can maximize its expressivity but still stay within the range, so that decidable reasoning procedures are available for Description Logics reasoners. In other words, OWL DL represents a balance point between expressive power and computational complexity. This is exactly the reason why OWL DL draws so much attention in Semantic Web research.
- **OWL Full:** OWL Full is the most expressive variant. However, its decidability cannot be guaranteed. One major flexibility in OWL Full but missing in OWL DL and OWL Lite is that the set of classes and the set of instances can overlap. In other words, an identifier can both be a class and an instance at the same time. For example, “professor” as a class refers to the collection of individuals who hold tenure track in universities and colleges; in the meanwhile it can be an instance of the class “occupation”. In addition, the *datatype* properties are a subclass of *objectType* properties. All the OWL vocabulary can be used unrestrictedly in OWL Full. However, these features come at the price of sacrificing decidability.

2.2.2 *Description Logics*

Now the focus changes to Description Logics ([BN02]), as they are the underlying foundation of the most popular language OWL DL.

Description Logics were firstly introduced in the 1980s to overcome the ambiguities of early semantic networks. They represent a class of knowledge representation formalisms that describe the knowledge of a specific domain (a.k.a “world”) by defining classes of the domain, and then utilizing these concepts to specify properties of objects and individuals in this domain. On top of knowledge representation, Description Logics offer a featured reasoning service that allows one to infer implicit new knowledge from the explicitly described knowledge base. They possess more expressive power than propositional logic, in the meanwhile, more efficient decision procedures than first-order logic.

A knowledge base can be divided into two components: the *TBox* and the *ABox*. The *TBox* introduces the terminology, i.e. the vocabulary of an application domain, while the *ABox* specifies assertions about named individuals in terms of this vocabulary. The vocabulary consists of *concepts* (or *classes* in OWL) and *roles* (or *properties* in OWL). Concepts are collections of individuals, and roles denote binary relationships between individuals. *Atomic concepts* refer to concepts that are directly defined by their members. They serve as the building blocks for *complex concepts*. *Atomic roles* and *complex roles* can be defined similarly. Different flavors in the Description Logics are differed by the constructs they can use in building complex concepts and roles. Besides storing terminologies and assertions, reasoning services are also common to the Description Logics family.

By convention, letters A and B denote atomic concepts; letter R denotes atomic roles; and letter C and D denote concept descriptions. Next, we formally introduce the syntactical grammar and semantics for a basic description language ALC .

TBox syntax:

C	\rightarrow	A	(atomic concept)
		\top	(universal concept)
		\perp	(bottom concept)
		$\neg A$	(atomic negation)
		$C \sqcap D$	(intersection)
		$\forall R.C$	(value restriction)
		$\exists R.\top$	(limited existential quantification)
		$\neg D$	(arbitrary negation)

TBox semantics:

Let I be a set of interpretations that consist of a non-empty set Δ^I (the domain of the interpretation) and an interpretation function, which assigns to every atomic concept A a set $A^I \subseteq \Delta^I$ and to every atomic role R a binary relation $R^I \subseteq \Delta^I \times \Delta^I$. The interpretation function is extended to concept descriptions by the following inductive definitions:

\top^I	=	Δ^I
\perp^I	=	\emptyset
$(\neg A)^I$	=	$\Delta^I \setminus A^I$
$(C \sqcap D)^I$	=	$C^I \cap D^I$
$(\forall R.C)^I$	=	$\{ a \in \Delta^I \mid \forall b.(a,b) \in R^I \rightarrow b \in C^I \}$
$(\exists R.\top)^I$	=	$\{ a \in \Delta^I \mid \exists b.(a,b) \in R^I \}$

$$(\neg D)^I = \Delta^I \setminus D^I$$

Example 2.3 To give an example of what can be expressed in *ALC*, consider two atomic concepts *Person* and *Female*, and an atomic role *hasChild*, then both $Person \sqcap \exists hasChild. and $Person \sqcap \forall hasChild.Female$ are valid *ALC* concepts, denoting those people that have at least one child, and those people whose children are all female, respectively.$

Other members of Description Logics family, like *FL*-, can be obtained from *ALC* by disallowing arbitrary negation and atomic negation. If we further disallow limited existential quantifications, we can have *FL*₀.

Definition 2.1 (Terminological Axioms) *Terminological axioms* are statements about how concepts or roles are related to each other. In general, they have the form

$$C \sqsubseteq D \quad (R \sqsubseteq S) \text{ or}$$

$$C \equiv D \quad (R \equiv S),$$

where C, D are concepts and R, S are roles. Axioms of the first kind are called *inclusions*, while axioms of the second kind are called *equalities*.

Definition 2.2 (Definitions) *Definitions* are specific axioms that identify terminologies as sets of definitions by which we can introduce atomic concepts as abbreviations or *names* for complex concepts. An equality whose left-hand side is an atomic concept is a *definition*. We call a finite set of *definitions* T a *terminology* or *TBox* if no symbolic name is defined more than once.

Definition 2.3 (Specializations) *Specializations* are inclusions whose left-hand side is atomic. They are introduced for certain concepts that cannot be defined completely, but we can still state necessary conditions for concept membership using an inclusion.

Definition 2.4 (Model of Axioms) Given T a set of axioms, then interpretation I satisfies T iff I satisfies each element of T . If I satisfies an axiom (resp. a set of axioms), then we say that it is a

model of this axiom (resp. set of axioms). Two axioms or two sets of axioms are *equivalent* if they have the same models.

As the other component of a knowledge base, ABox describes a specific state of affairs of an application domain in terms of concepts and roles.

ABox syntax:

$$C(a),$$

$$R(b, c).$$

where C is a concept and R is a role, a, b, c are names for individuals.

ABox semantics:

We now give the semantics to ABox by extending interpretations to individual names. From now on, an interpretation $I = (\Delta^I, \bullet^I)$ not only maps atomic concepts and roles to a set of relations, but in addition maps each individual name a to an element $a^I \in \Delta^I$.

$$C(a)^I = a^I \in C^I$$

$$R(b, c)^I = (a^I, b^I) \in R^I$$

An interpretation *satisfies* the ABox A if it satisfies each assertion in A . In this case we say that I is a *model* of the assertion or of the ABox. I satisfies an assertion α or an ABox A *with respect to* a TBox T if in addition to being a model of α or of A , it is a model of T . Thus, a model of A and T is an abstraction of a concrete world where the concepts are interpreted as subsets of the domain as required by the TBox and where the membership of the individuals to concepts and their relationships with one another in terms of roles respect the assertions in the ABox.

Often, an analogy is established between databases on the one hand and DL knowledge bases on the other hand. The schema of a database is compared to the TBox and the instance with the actual data is compared to the ABox. However, the TBox imposes semantic relation-

ships between the concepts and roles that do not have counterparts in database semantics. In addition, the semantics of ABox differs from the usual semantics of database instances. While a database instance represents exactly one interpretation, namely the one where classes and relations in the schema are interpreted by the objects and tuples in the instance, an ABox represents many different interpretations, namely all its models. As a consequence, absence of information in a database instance is interpreted as negative information, while absence of information in an ABox only indicates lack of knowledge.

Example 2.4: If the only assertion about Peter is $hasChild(PETER, HARRY)$, then in a database this is understood as a representation of the fact that *Peter* has only one child, *Harry*. In an ABox, the assertion only expresses that, in fact, *Harry* is a child of *Peter*. However, the ABox has several models, some in which *Harry* is the only child and others in which he has brothers or sisters. Consequently, even if one also knows (by an assertion) that *Harry* is male, one cannot deduce that all of *Peter*'s children are male.

The semantics of ABox is therefore sometimes characterized as an “open-world” semantics, while the traditional semantics of databases is characterized as a “closed-world” semantics. Since ABox represents possibly infinitely many interpretations, namely its models, query answering and reasoning are more complex.

Finally, now we will briefly discuss the reasoning tasks in Description Logics.

TBox reasoning:

- **Satisfiability:** A concept C is *satisfiable* with respect to T if there exists a model I of T such that C^I is nonempty. In this case we say also that I is a model of C .
- **Subsumption:** A concept C is *subsumed* by a concept D with respect to T if $C^I \subseteq D^I$ for every model I of T . In this case we write $C \sqsubseteq_T D$ or $T \mapsto C \sqsubseteq D$.

- **Equivalence:** Two concepts C and D are *equivalent* with respect to T if $C^I = D^I$ for every model I of T . In this case we write $C \equiv_T D$ or $T \mapsto C \equiv D$.
- **Disjointness:** Two concepts C and D are *disjoint* with respect to T if $C^I \cap D^I = \emptyset$ for every model I of T .

Checking (un)satisfiability of concepts is a key inference. It has been shown that all other three reasoning tasks can be reduced to unsatisfiability check.

Proposition 2.1 (Reduction to Unsatisfiability): for concepts C and D , the following statements hold with respect to a TBox:

- 1) C is *subsumed by* $D \Leftrightarrow C \sqcap \neg D$ is *unsatisfiable*;
- 2) C and D are *equivalent* \Leftrightarrow both $(C \sqcap \neg D)$ and $(\neg C \sqcap D)$ are *unsatisfiable*;
- 3) C and D are *disjoint* $\Leftrightarrow C \sqcap D$ is *unsatisfiable*.

ABox reasoning:

- **Consistency:** An ABox A is *consistent* with respect to a TBox T , if there is an interpretation that is a model for both A and T . We simply say that A is *consistent* if it is consistent with respect to the empty TBox.
- **Instance:** It checks whether an ABox entails an assertion. We say that an assertion α is *entailed* by A and we write $A \mapsto \alpha$, if every interpretation that satisfies A , that is, every model of A , also satisfies α .
- **Retrieval:** given an ABox A and a concept C , to find all individuals a such that $A \mapsto C(a)$.
- **Realization:** given an individual a and a set of concepts, find the *most specific concepts* C from the set such that $A \mapsto C(a)$. It is the dual inference problem to retrieval.

2.3 Logic Programming and Logic-based Data-Language

2.3.1 Logic Programming

Logic programs have emerged as a very expressive formalism for knowledge representation. Deductive databases combine logic programming with relational databases to construct database systems with formulas that can also efficiently deal with large datasets. Deductive databases are more expressive than relational databases but less expressive than logic programming systems. One of differences between logic programs and deductive databases is that building up complex symbols by function symbols is allowed in logic programs, while this is not allowed in deductive databases.

We give a brief overview of general logic programs. We first look at the logic formulas, called *clauses*.

Definition 2.5 (Clause) A *clause* is a formula with universal quantification over the whole formula, that is, in the form of:

$$\forall(L_0 \vee \dots \vee L_n)$$

where for $0 \leq i \leq n$ each L_i is an atomic formula (a positive literal) or the negation of an atomic formula (a negative literal).

Definition 2.6 (General Logic Program) A *general logic program* is a set of *clauses* that are usually expressed in the form:

$$A \leftarrow B_1, B_2 \dots B_n.$$

where A, B_1, B_2, \dots, B_n are *atomic formulas* and $n \geq 0$. An atomic formula is of the form $P(t_1, t_2, \dots, t_n)$ or negation of $P(t_1, t_2, \dots, t_n)$ (noted as $\neg P(t_1, t_2, \dots, t_n)$) where P is the *predicate* symbol with finite arity $n \geq 0$ and t_1, t_2, \dots, t_n are terms. Here A is called the *head* or *conclusion* of the

rule and conjunction of $B_1 \wedge B_2 \wedge \dots \wedge B_n$ is called the *body* or *premise* of the *rule*. Rules are divided into two classes:

- 1) *EDB* (extensional database) rules are known as facts. In this situation the body is empty (i.e. $n=0$) and the implication arrow \leftarrow is omitted. The head A is implied to be always true and EDB rules normally are written at the beginning of the program. EDB is stored as relations in the database. If p is an EDB, there will be a corresponding relation, say P , in the database; and $p(a_1, \dots, a_k)$ is true if and only if there is a tuple (a_1, \dots, a_k) in the relation P .
- 2) *IDB* (intensional database) rules are evaluated using the EDB and IDB predicates in the body can be in a recursive manner to give the meaning of the program.

A deductive database system is a database system, which can make deductions based on rules and facts stored in the deductive database. A deductive database is commonly viewed as a general logic program and can have negation and express recursive views. DATALOG is the language typically used to specify facts, rules and queries in deductive databases. We now introduce the syntax of deductive databases in the DATALOG form.

Definition 2.7 (Alphabet) An *alphabet* is a finite set of symbols.

- 1) Constant symbol

The *constants* can be atoms, or numbers.

- 2) Variable symbol

The *variables* are letters or words written in capital letters.

- 3) Predicate symbol

The *predicate symbols* are letters or words.

- 4) Special symbol

Special symbols include $(,), _ , , < , \leq , > , \geq , \diamond , = , \cdot , :-$.

Definition 2.8 (Term) The set T of *terms* over a given alphabet A is the smallest set such that

- 1) any constant in A is in T ;
- 2) any variable in A is in T .

Definition 2.9 (Atomic Formula) If p is an n -ary predicate symbol and t_1, t_2, \dots, t_n are terms and at least one of terms must be a constant or a variable, $p(t_1, t_2, \dots, t_n)$ is a formula called an *atomic formula*.

Definition 2.10 (Literal) A *literal* is either a positive literal A or a negative literal $\neg A$, where A is an atomic formula.

Definition 2.11 (Rule) Deductive *rules* are expressed in the following form:

$$p :- q_1, q_2, \dots, q_n.$$

where $n \geq 1$ and p is an atomic formula and q_1, q_2, \dots, q_n are literals. If $n=0$, the body of the rule is empty, and implication symbol $:-$ will be omitted. This rule is in the EDB in the form of:

$$p.$$

Definition 2.12 (Deductive Database) A *deductive database* is a finite set of deductive rules.

Example 2.5 The following example is a deductive database consists of three EDB rules and two IDB rules with recursive negation.

$$t_0(1).$$

$$g(1, 2, 3).$$

$$g(3, 2, 5).$$

$$t(Z) :- t_0(Z).$$

$$t(Z) :- g(X, Y, Z), t(X), \neg t(Y).$$

Now we will briefly present the semantics of logic programming. To determine the set of models of a logic program, we can use the work of *Herbrand model*. To determine the Herbrand model, we first look at the concept of *ground*. A term, atom, literal, or clause is called *ground* if it contains no variables. A *ground instance* of a term, atom, literal, or clause Q is the term, atom, literal, or clause, respectively, obtained by replacing each variable in Q by a constant. Now we focus on *Herbrand models* illustrated by definitions and an example.

Definition 2.13 (Herbrand universe and Herbrand base) For a general deductive database DB the *Herbrand universe* U_{DB} is the set of all *ground* terms that can be created using constant. Note that U_P for logic programs P is the set of all *ground* terms that can be created using constant and function symbols. The *Herbrand base* B_{DB} is the set of all ground atoms created using predicate symbols and object forms.

Definition 2.14 (Herbrand interpretation) A *Herbrand interpretation* of a general deductive database DB is an interpretation I_{DB} such that:

- the domain of I_{DB} is U_{DB} ;
- a predicate mapping defines a mapping function F such that:
 - for every constant c or variable v , mapping is defined to be itself:

$$F_c : c \rightarrow U_c, F_v : v \rightarrow U_v;$$
 - for every n -ary predicate symbol p

$$F_p : p \rightarrow \text{the set of all relations on } U_p^n, \text{ that is, the set of all } n\text{-tuples of ground terms};$$
 - for every n -ary functor f , the mapping is defined as follows:

$$F_f : f \rightarrow \text{the set of all functions on } U_f^n, \text{ that is, all ground terms are composed into } f;$$

- The properties of Herbrand interpretations are as follows:
 - any subset of B_{DB} is termed I_{DB} ;
 - Atoms in the interpretation are assumed to be true and those outside the interpretation are assumed to be false.

Definition 2.15 (Herbrand model) A Herbrand interpretation is a *Herbrand model* (abbr. *model*) of a set of formulas if and only if each formula in the set is true in the Herbrand interpretation.

2.3.2 Logic-based Data-Language

The logic-based Data-Language (LDL) ([TZ86]) is a language introduced in 1980s. Its design goal is to combine the flexibility of logic programming with the high performance of the relational database technology. The query languages of relational databases are logic-based. Relational systems are considered superior to logic programming with respect to ease of use, data independence, suitability for parallel processing, and secondary storage access. On the other hand, the expressive power and functionality offered by database query language is limited compared to the logic programming languages. LDL is an effort that combines these two approaches by designing and supporting a logic-based query language that combines the power of logic programming with the advantages in relational systems.

The language LDL is equipped with the following features:

- LDL is based on pure Horn clause logic (i.e., the sequential order of execution of rules in a procedure or sub-goals within a given rule has been removed).
- Sets have been introduced as primitive data objects that can be used directly in the language rather than their simulation through lists as in logic programming.
- A form of negation is based on set-difference.

- Schema-definition and update facilities were included.

Among the above features, the introduction of set is the most interesting one to our research. The model of computation employed by LDL is that of one set at a time. Consequently, the response to a query would be to compute all of the possible answers that can be deduced from the base relations. LDL provides an explicit form of set manipulation; it enables the user to use sets as data objects in the specification of rules and facts. The advantages of having sets as a primitive in LDL include convenience, expressive power, and efficiency. Three set constructs become handy in set manipulation.

- **Set-enumeration:** LDL allows the specification of complex terms in facts and rules; these complex terms may include sets. One of the ways to specify a set is to explicitly list every set element whose order does not matter.

Example 2.6: This example specifies the relationship between a manager and his employees. Each member of the employee set is represented by a tuple, which in itself contains a set of hobbies of the employee.

```
employees_of (bill, brown,
             {(red, russell, {working, jogging, bicycling}),
             (mac, fat, {cooking, eating}),
             (graham, greene, {spy-novels}),
             ... }).
```

- **Set-generation:** Set-generation is the process of generating all of the elements of a set that meet some specifications. In standard mathematical notation this would be denoted as $s = \{x \mid p(x)\}$ where $p(x)$ is a predicate on x . In LDL, set generation is specified in rule form, as follows:

$$s(<x>) \leftarrow p(x).$$

Example 2.7: This example generates a set of items supplied by a supplier when the base relation is $suppl(Sup\#, Item\#)$.

$$item_set(Sup\#, <Item\#>) \leftarrow suppl(Sup\#, Item\#).$$

- **Partition:** The partition primitive partitions a set S into two disjoint subsets $s1$ and $s2$ having at least one element; the exact form of partitioning is transparent to the user. The partition primitive is essential for counting in set. It enables the specification of cardinality in a recursive manner, and the operation on each of the partitioned subsets can proceed in parallel.

Example 2.8: This example generates the number of items supplied by each supplier. The constructs *cardinality* and *partition* are required to do so.

$$item_count(Sup\#, Count) \leftarrow item_set(Sup\#, S), cardinality(S, Count).$$

$$cardinality(\{\}, 0).$$

$$cardinality(\{X\}, 1).$$

$$cardinality(Set, Value) \leftarrow partition(Set, Set1, Set2),$$

$$cardinality(Set1, Value1),$$

$$cardinality(Set2, Value2).$$

$$Value = Value1 + Value2.$$

2.4 The Paraconsistent Model

In this section we present key background knowledge of paraconsistent relational data model related to the dissertation. The paraconsistent relational data model extends the traditional relational model with explicit negation by allowing both positive and negative facts to be stored

in paraconsistent relations. Here we give a brief overview of this model, for a detailed description the reader is referred to [BS95].

2.4.1 Paraconsistent Relations

Paraconsistent relations are the fundamental mathematical structures underlying the model, which essentially contains two kinds of tuples, ones that definitely belong to the relation and others that do not belong to the relation.

Definition 2.16 (Paraconsistent) A *paraconsistent relation* R is defined as a pair $\langle R^+, R^- \rangle$, where R^+ and R^- are sets of tuples in the relational schema, where tuples in R^+ denote positive facts and tuples in R^- denote negative facts.

Let a (relation) scheme Σ be a finite set of attribute names. Let $\tau(\Sigma)$ denote the set of all tuples on Σ . A paraconsistent relation R on scheme Σ , is *consistent* if $R^+ \cap R^- = \emptyset$. Moreover R is called *complete* relation if $R^+ \cup R^- = \tau(\Sigma)$. If R is consistent and complete *i.e.* $R^- = \tau(\Sigma) - R^+$, then it is a *total* relation.

If a tuple falls beyond the union of R^+ and R^- , it can only be assumed as unknown. If a tuple appears in both R^+ and R^- , it indicates an inconsistency. Notice that a paraconsistent structure is strictly general than an ordinary relation. For any ordinary relation, there exists a corresponding paraconsistent relation, but not vice versa. The paraconsistent data model is able to handle incomplete information about tuples, yet still provides a way to check inconsistent data. For the above reasons, paraconsistent model might be suitable to mediate translation from relational data to OWL instances, while keeping the ability to maintain constraints and reject inconsistent data. For details about a complete set of algebraic operators on paraconsistent relations and appropriate handling of storage issues, the reader is referred to [BS95].

2.4.2 Algebraic Operators on Paraconsistent Relations

This sub-section presents some algebraic operators on paraconsistent relations related to this dissertation. To reflect the generalization of algebraic operators of ordinary relations, a dot is placed over the ordinary relation operator to obtain corresponding paraconsistent relation operator and distinguish from the ordinary operator. For example, \bowtie denotes the natural join among ordinary relations, and $\dot{\bowtie}$ denotes natural join among the paraconsistent relations. We first look at related *set-theoretic operators*.

Definition 2.17 (set-theoretic operators) Let R and S be two paraconsistent relations on scheme Σ . Then,

- the *complement* of R , denoted by $\dot{\neg} R$, is a paraconsistent relation on scheme Σ given by

$$(\dot{\neg} R)^+ = R^-, (\dot{\neg} R)^- = R^+;$$

- the *union* of R and S , denoted by $R \dot{\cup} S$, is a paraconsistent relation on scheme Σ given by

$$(R \dot{\cup} S)^+ = R^+ \cup S^+, (R \dot{\cup} S)^- = R^- \cap S^-;$$

- the *intersection* of R and S , denoted by $R \dot{\cap} S$, is a paraconsistent relation on scheme Σ given by

$$(R \dot{\cap} S)^+ = R^+ \cap S^+, (R \dot{\cap} S)^- = R^- \cup S^-;$$

- the *difference* of R and S , denoted by $R \dot{\ominus} S$, is a paraconsistent relation on scheme Σ given by

$$(R \dot{\ominus} S)^+ = R^+ \cap S^-, (R \dot{\ominus} S)^- = R^- \cup S^+.$$

Now, we look at *relation-theoretic operators*. If Σ and Δ are relation schemes such that $\Sigma \subseteq \Delta$, then for any tuple $t \in \tau(\Sigma)$, we let t^Δ denote the set $\{t' \in \tau(\Delta) \mid t'(A) = t(A), \text{ for all } A \in \Sigma\}$ of all extensions of t . We extend this notion for any $T \subseteq \tau(\Sigma)$ by defining $T^\Delta = \bigcup_{t \in T} t^\Delta$.

Definition 2.18 (Natural Join/ Join) Let R and S be partial relations on schemes Σ and Δ , respectively. Then, the (*natural*) *join* of R and S , denoted by $R \bowtie S$, is a partial relation on the scheme $\Sigma \cup \Delta$, given by

$$(R \bowtie S)^+ = R^+ \bowtie S^+, \quad (R \bowtie S)^- = (R^-)^{\Sigma \cup \Delta} \cup (S^-)^{\Sigma \cup \Delta};$$

It is important to observe that $(R \bowtie S)^-$ contains all extensions of tuples in R^- and S^- , because at least one of R and S is believed false for these extended tuples.

Definition 2.19 (Projection) Let R be a paraconsistent relation on scheme Σ , and Δ be any scheme. Then, the *projection* of R onto Δ , denoted by $\dot{\pi}_\Delta(R)$ is a paraconsistent relation on Δ given by

$$\dot{\pi}_\Delta(R)^+ = \pi_\Delta((R^+)^{\Sigma \cup \Delta}), \quad \dot{\pi}_\Delta(R)^- = \{t \in \tau(\Sigma) \mid t^{\Sigma \cup \Delta} \subseteq (R^-)^{\Sigma \cup \Delta}\},$$

where π_Δ is the usual projection over Δ on ordinary relations.

Definition 2.20 (Strong Projection) Let R be a paraconsistent relation on scheme Σ , and Δ be any scheme. Then, the *strong projection* of R onto Δ , denoted by $\dot{\mu}_\Delta(R)$ is a paraconsistent relation on Δ given by

$$\dot{\mu}_\Delta(R) = \dot{\cdot} (\dot{\pi}_\Delta(\dot{\cdot} R)).$$

CHAPTER 3 MAINTAINING INTEGRITY CONSTRAINTS IN RELATIONAL TO OWL TRANSLATION

The data of Semantic Web exist in machine readable format like RDF(S) and OWL, in order to promote data exchange on the web based on their semantics. Besides starting from scratch, one can also automatically construct OWL ontologies by translating relational databases into RDFs/OWLs, although this method tends to lose most, if not all, vital constraints from the relational databases. In this chapter, we focus on maintaining the integrity constraints in the very first process of OWL lifecycle shown in figure 1.1. We present a paraconsistent logic based method to maintain as many relational integrity constraints as possible while translating relational data into OWL. In addition, we are able to further enforce OWL-like constraints on the resulting ontologies, as they are often desirable if we want to fully exploit the expressiveness and logic inference power of OWL. Most importantly, with this method, the problem of inconsistency checking of OWL instances reduces to simple instance matching within two classes.

3.1 Motivations

Although numerous ontologies are available in the RDF/RDFS form, there is still a huge demand for developing more OWL ontologies for various purposes. With the help of developing tools such as Protégé, one can always construct ontology from scratch by first defining concepts and relationships in the TBox, followed by inserting instances of concepts and relationships into the ABox of the ontology. However, when the knowledge base gets huge, this developing process may become time-consuming and erroneous. The idea of translating relational databases into OWL automatically seems appealing, as relational resources are abundant and of great importance. Taking the contents of deep web into consideration, the market for this idea becomes several orders of magnitude larger. For one reason, the size of those hidden data is incredibly

enormous, and for the other reason, the owners of these resources might want to take advantages of the reasoning services in OWL as well. However, current approaches to converting relational information into OWL instances tend to lose most, if not all, vital constraints from the relational databases. Undoubtedly, maintaining integrity constraints in semantic web could be as important as in relational databases, especially for database users who wish to exploit automated reasoning in OWL. Integrity constraints are valuable in checking and enforcing data consistency, providing further semantics on data, and promoting semantic query optimization.

Researchers have constantly shown the important usages of mapping from relational databases to OWL in the promotion of semantic web ([Ber04], [SSV02], [Biz03]). In ([Biz03]), Bizer introduced a database to RDF mapping language called D2R MAP. In accordance, W3C published new ontology patterns that are used to capture n-ary relations in RDF and OWL, while conventional semantic web languages are only capable of representing binary relations. Although these approaches greatly facilitate the generation of OWL instances, they tend to lose vital constraint information and barely support generic mapping. On the other hand, efforts have been spent on integrating integrity constraints within RDFs ([TDB+08], [MHS07], [MHR06]). The current trend of incorporating ICs into OWL is either based on the query language SPARQL ([TDB+08]) or based on logic programming ([MHR06]). However, to the best of our knowledge, none of them combines with the relational to RDF mapping and addresses the important relational aspect of integrity constraints like our approach does.

3.2 The paraconsistent-based translation

Throughout this section, we will demonstrate our approach using the following small yet effective relational database in figure 3.1. There are four relational tables, namely *people*, *student*, *course*, and *take*. For simplicity (and without loss of generality) we will assume that each

of the "entity" tables (in this case all but the *take* table) consists of one column containing the primary key value and each "relationship" table (in this case the *take* table) consists of the primary key columns of the entity types involved in the relationship. Again, for simplicity we will assume that all relationship tables correspond to binary relationships, as the same in OWL.

<i>people</i>	<i>student</i>	<i>course</i>	<i>take</i>
a	a	s	a s
b	b	t	b t
c			

Figure 3.1 A small relational database with 3 foreign keys: (1)*student* pointing to *people*; (2)*student* column in *take* pointing to *student*; (3)*course* column in *take* pointing to *course*.

3.2.1 Foreign Key Constraint

Our method to deal with foreign keys is composed of five steps as detailed below:

Step 1:

Convert the given relational database into a paraconsistent database by adding the following tables shown in Figure 3.2, using the Relational to Paraconsistent Transformation Algorithm. Missing tuples are considered to be false under the relation predicate due to the Closed World Assumption in relational database, thus these tuples go into negative tables.

The following figure shows the negative data tables for the example database.

<i>people</i> ⁻	<i>student</i> ⁻	<i>course</i> ⁻	<i>take</i> ⁻
	c		a t
			b s
			c s
			c t

Figure 3.2 Negative data tables

Algorithm: Relational to Paraconsistent Transformation

For every entity table R

Let S be the most general super-class of R

(Note: $S = R$ if R does not have any superclass)

$$R^- = \{a \mid a \in S \text{ and } a \notin R\}$$

End For;

For every relationship table R that has two columns (say A and B)

Let C be the most general superclass of A

($C = A$ if A does not have any superclass)

Let D be the most general superclass of B

($D = B$ if B does not have any superclass)

$$R^- = \{(a, b) \mid a \in C \text{ and } b \in D \text{ and } (a, b) \notin R\}$$

End For;

Step 2:

Translate positive table(s) in the relational database into regular classes and properties in OWL. Each tuple of a positive table will become an OWL instance of a class whose name is the same as the corresponding table name in the relational data. For the example database, we will get $people(a)$, $people(b)$, $people(c)$, $student(a)$, $student(b)$, $course(s)$, $course(t)$, $take(a, s)$, $take(b, t)$.

Step 3:

Express integrity constraints as regular Description Logic axioms. For the first two foreign keys in the sample relational database, the following two DL axioms will be generated:

$$student \sqsubseteq people$$

$$\exists take.\top \sqsubseteq student$$
Step 4:

Translate the negative table(s) that corresponds to the right-hand side of the DL axioms into new OWL class(es). If the right-hand side is in the form of negation (e.g. $\neg C$), still translate the table C^- into an OWL class. For the above example, we will introduce two new OWL classes: $people^-$ and $student^-$ and introduce the instance $student^-(c)$.

Step 5 (IC Check):

Check whether every non-empty negative class and its positive counterpart are disjoint by simply running the following SPARQL query:

```
ASK{
  ? x rdf:type C.
  ? y rdf:type C-.
  FILTER ( ?x = ?y)
}
```

If yes then IC is not violated. In our case, we only need to check if $student$ and $student^-$ classes in OWL ontology are disjoint. From now on till the end of section 3.2.1, let us only focus on the second axiom. For example, if we are trying to insert $take(c, t)$ into our ontology, system will attempt to insert $student(c)$ automatically. Then a violation will occur since our ontology contains both $student(c)$ and $student^-(c)$.

Please note that the reason for not translating negative tables for the left-hand side of the DL axioms (in the case of the second axiom, $take^-$ table) into OWL classes is that we still want to maintain some flexibility for OWL. Otherwise every insertion of new data into current ontology will cause violation. For example, instead of inserting $take(c, t)$, we insert $take(a, t)$ into OWL, and our OWL is still legal under the second axiom, even though system will attempt to insert $student(a)$ automatically (and will find $student(a)$ is already present). This is consistent with our relational DB because $take(a, t)$ is a valid insertion in relational database as well. However, if $take^-$ class is also present as the result of translating negative left-hand side tables, insertion of $take(a, t)$ will raise violation because $take^-(a, t)$ will also be available in our ontology. This false positive indication of violation is obviously unreasonable.

3.2.2 Total Participation Constraint

Our methodology can also easily handle Total Participation Constraint that is present in the relational database. Consider the constraint: every student must take a course. The first two steps of our methodology to handle such constraints are exactly the same as for foreign key constraint. We now start directly from Step 3 by expressing the integrity constraint as:

$$student \sqsubseteq \forall take.course$$

As $take$ is on the right-hand side of this axiom, in Step 4, we include a $take^-$ class and its instances $take^-(a, t)$, $take^-(b, s)$, $take^-(c, t)$, and $take^-(c, s)$. In Step 5, if we want to insert $student(c)$ into OWL, we will have $take(c, t)$, or $take(c, s)$ or both. Each one of these three cases will raise violation because they all conflict with $take^-$ class.

3.3 OWL-style constraints

In addition to maintaining the integrity constraints that were enforced in the original relational database, we are also able to specify and enforce additional OWL-style constraints on the transformed RDF(S)/OWL data. We illustrate two such constraint types with examples.

3.3.1 *Redundant Individual Type Constraint*

An OWL-style constraint called redundant individual type constraint is illustrated first. This constraint specifies that an individual cannot be explicitly declared to have both class C and D (where D is a superclass of C) as its types. Suppose $C \sqsubseteq D$ is already defined in the TBox. Then, in Step 3 this constraint can be expressed as:

$$C \sqsubseteq \neg D$$

To enforce this constraint on *student* class we must check if every instance in *student* also appears in *people*. If an instance of *student* does not appear in *people*⁻, a constraint violation should be reported.

Please note the inclusion of $C \sqsubseteq D$ and $C \sqsubseteq \neg D$ at the same time may seem contradictory, but they bear different meanings. The former is a part of regular ontology and can be used to formally infer new knowledge, while the later is created solely for generating negative tables and integrity checking and it should not participate in regular reasoning services. These differences can be easily differentiated during implementation.

3.3.2 *Specific Individual Type Constraint*

An OWL-style constraint called Specific Individual Type constraint requires that the declared type of a given individual in the instance data must be the most specific one. Suppose the following class hierarchy is already defined in the TBox:

$$C_1 \sqsubseteq C_2$$

$$C_2 \sqsubseteq C_3$$

...

$$C_{i-1} \sqsubseteq C_i$$

...

$$C_{n-1} \sqsubseteq C_n$$

Then, in Step 3, this constraint can be expressed as follows:

for each $1 \leq i \leq n - 1$:

$$C_i \sqsubseteq \neg C_{i+1}$$

$$C_i \sqsubseteq \neg C_{i+2}$$

...

...

$$C_i \sqsubseteq \neg C_n$$

This type of constraint can also be easily checked by membership in positive or negative tables.

As an illustration of specific individual type constraint, consider the class hierarchy:

$$\textit{graduateStudent} \sqsubseteq \textit{student} \sqsubseteq \textit{people}$$

Let $\{a, b, c, d\}$ be the set of all individuals and let a and d be instances of *graduateStudent*. This will prohibit a and d from being stated as instances of *student* and *people* classes. Let c be an instance of *student*. This will prohibit c from being stated as an instance of *people* class. So, a valid

ABox will be:

$$\textit{people}(b)$$

$$\textit{student}(c)$$

$$\textit{graduateStudent}(a)$$

$$\textit{graduateStudent}(d)$$

If one tries to add $student(a)$, system will detect an inconsistency. This is because a , b , and d will appear in class $student^-$ due to the IC axiom:

$$graduateStudent \sqsubseteq \neg student$$

There is an evident conflict between $student(a)$ and $student^-(a)$.

Besides redundant individual type constraints and specific individual type constraints, others constraints are also definable with this approach as well. In fact, any integrity constraint of the form

$$A \sqsubseteq C \text{ (} A \text{ is subsumed by } C\text{)}$$

is enforceable in our approach, where A is either an atomic class or a compound class built from other classes using connectives such as union, intersection, negation, and etc.. An instance of A will require itself to be present under the class C . Many user-defined integrity constraints take this form.

CHAPTER 4 MAINTAINING INTEGRITY CONSTRAINTS IN A SINGLE OWL

This chapter focuses on maintaining integrity constraints within a single OWL, whenever there is an update issued to this ontology. The update we consider corresponds to the second process in the OWL lifecycle shown in figure 1.1. Enforcing constraints in a single OWL represents the majority of ongoing research interests in the field of ICs in Semantic Web. For the rest of this chapter, we will first look at the motivations of our work in this process. Then we formally introduce the syntax and semantics of Rule-based DL language with set extension. By using sample ontology, we show that the newly introduced language is effective to capture many commonly used integrity constraints.

4.1 Motivations

Several lines of approaches have been proposed to address IC issues in a single OWL. They either integrate OWL with a different formalism such as rules or epistemic queries, or provide an alternative semantics in OWL. The rule-based approach ([EIL+04], [Mot07]) builds a hybrid knowledge base (KB) in which rules are responsible for imposing ICs on the DL data. If the KB entails a certain rule predicate, then a violation to ICs exist. However, this approach is computationally expensive when the data set gets huge. In addition, important aggregate types of constraints are hard, if possible, to express in this formalism. The epistemic query-based method ([CGL+07]) checks the satisfaction of ICs by asking queries against the KB. The results of these queries determine whether there is a violation or not. Unfortunately, the data complexity that inherits from this approach in expressive DLs still remains unknown. Most recently, an alternative semantics-based approach ([Tao10]) emerged as a different line of solution. In this approach, the semantics of OWL has been extended. Now OWL not only serves as an ontology modeling language, but also works as a native language to specify integrity constraints. Although the effort to

provide a unified ontology and IC language has been very much appreciated, confusions in distinguishing these two may arise as well. In addition, non-traditional semantics can have interoperability issues when an application based on this approach interacts with other conventional applications. Also, the types of constraints this unified language is capable of expressing are bounded by the OWL formalism.

In this study, we carefully designed a language named Rule-based DL language with set extension that does not have to be omnipotent in specifying every possible integrity constraints, but rather capable of defining constraints that are mostly used by ontologists in practice. On the other hand, we were able to confine our DL base so that efficient reasoning is possible.

4.2 Syntax

Here we first introduce the Rule Logic syntax, followed by the syntax of Description Logic. At the end of this section, Rule-based DL language with set extension is defined.

4.2.1 Rule Logic (RL) syntax

The definition of Rule Logic is based on a series of definitions including alphabet, term, atomic formula, literal, and rule.

Definition 4.1 (Alphabet) The *alphabet* A of the Rule Logic consists of the following classes of symbols:

- 1) *variables* denoted by capital letters, e.g. X, Y, Z;
- 2) *constants* denoted by lower-case letters, e.g. a, b, c ;
- 3) *predicate symbols* denoted by alphanumeric identifiers in lower-case, with an associated arity ≥ 0 , e.g. pre/0;
- 4) arithmetic and comparison predicates +, -, *, /, <, <=, =, >, >=, <>;
- 5) miscellaneous symbols (,), :-, ,, <, >, **K**, *not*.

Although comparison predicates $<$ and $>$ share the same syntax with *set generation* symbols $<$ and $>$ in miscellaneous symbols, their semantics should be easy to differentiate based on the context in which they are used.

Definition 4.2 (Terms) *Terms* T can be inductively defined as the smallest set such that:

- 1) any constant in A is in T ;
- 2) any variable in A is in T ;
- 3) any object ID;
- 4) $\langle X \rangle$, where X is a variable.

An object ID is commonly used to uniquely identify an object in ontologies. One such example could be a Universal Resource Identifier (URI). Set generation operator $\langle X \rangle$ gathers all of the elements of a set under certain specification.

Definition 4.3 (Atomic Formula) *Atomic formulas* F of well-formed formulas (wff) with respect to alphabet A and terms T are defined as the smallest set such that:

- 1) $p(t_1, t_2, \dots, t_n)$ is in F if p/n is a predicate symbol in A and t_1, t_2, \dots, t_n are terms in T ;
- 2) $\mathbf{K} p(t_1, t_2, \dots, t_n)$ is in F if $p(t_1, t_2, \dots, t_n)$ is in F ;
- 3) built-in atomic formula of the form $\mathbf{K} X\theta Y$ where X and Y are variables or constants and θ is comparison predicate $<, \leq, =, >, \geq, \diamond$.

Please note special operator \mathbf{K} is interpreted under minimal knowledge semantics, and *not* (used below) is interpreted under as failure.

Definition 4.4 (Literal) *Literal* L is defined as either a positive atomic formula, or a negative atomic formula denoted as *not* $p(t_1, t_2, \dots, t_n)$ (shorthand of *not* $\mathbf{K} p(t_1, t_2, \dots, t_n)$, by convention) where $p(t_1, t_2, \dots, t_n)$ is in F .

Definition 4.5 (Rule) A *Rule* is called RL rule if it is a formula of the following form:

$$KA :- L_1, \dots, L_k.$$

where KA , called the *head* of a rule, is in L , and L_1, \dots, L_k , called the *body* of a rule, are literals. A rule with an empty body is called a *fact* and is simply written as KA . A rule containing $\langle X \rangle$ in the head is called a *grouping rule*. A rule without a head is called a *constraint rule*; the derivation of an empty head indicates a violation to a constraint.

Definition 4.6 (Well-formed Rule) A rule is *well-formed* if it obeys the following syntactic restrictions:

- 1) the body contains no occurrence of the form $\langle X \rangle$;
- 2) the head contains at most one occurrence of the form $\langle X \rangle$, such occurrence must be an argument of the head predicate symbol;
- 3) all the predicates in the body of a grouping rule are positive;
- 4) all the predicates in the rule are either associated with the K operator or not operator.

(1), (2) and (3) are related to grouping rules and they are present in the original LDL language.

4.2.2 Description Logic (DL) syntax

Let \mathcal{O} be a Description Logics (DL) based OWL knowledge base whose data complexity of checking entailment of ground literals remains in P (e.g. Horn-SHIQ, DL-lite). The knowledge base \mathcal{O} can be further divided into two parts, namely TBox and ABox of the following generic form (specific grammar depends on the particular choice of language):

TBox:

$$A \sqsubseteq B$$

ABox:

$$A(a), B(a)$$

The reasoning services in DL constitute of TBox reasoning about intensional knowledge as well as ABox reasoning about extensional data.

4.2.3 *Rule-based DL language with set extension knowledge base*

Finally we define Rule-based DL language with set extension knowledge base k as a triple (T, A, R) , where a program R is a finite set of well-formed Rule Logic rules with the following additional properties:

- 1) any predicate in R with the same symbol of a concept or property in O (the collection of TBox T and ABox A) actually refers to the very same concept or property.
- 2) any object ID in P with the same identifier of a resource in O actually refers to the very same resource.

However, please note one can create additional predicate symbols and resources in R other than those already in the knowledge base O .

4.3 Semantics

The semantics of Rule-based DL language with set extension is largely based on the semantics of logic programming and semantics of DL we have introduced in section 2.2.2 and 2.3.1, respectively. However, we still need to specifically define the semantics of K operator.

A Rule-based DL language with set extension knowledge base k now is triple $k = (T, A, R)$, where T is a TBox, A is an ABox, and R is a set of rules of the forms defined in section 4.2.1. The semantics will be defined in such a way that it applies only to individuals in the knowledge base that provably are instances of C , but not to arbitrary domain elements, which would be the case if we dropped K . The fact that a knowledge base has knowledge about the domain can be understood in such a way that it considers only a subset W of the set of all interpre-

tations as possible states of the world. Those individuals that are interpreted as elements of C under all interpretations in W are then “known” to be in C .

Now, we define an epistemic interpretation as a pair (I, W) , where I is a first-order interpretation and W is a set of first-order interpretations. Every epistemic interpretation gives rise to a unique mapping $\bullet^{I,W}$ associating concepts and roles with subsets of Δ and $\Delta \times \Delta$, respectively. For \top , \perp , for atomic concepts, negated atomic concepts, and for atomic roles, $\bullet^{I,W}$ agrees with \bullet^I . For other constructors, $\bullet^{I,W}$ can be defined analogously. Note that for a concept C without an occurrence of \mathbf{K} , the sets $C^{I,W}$ and C^I are identical. The set of interpretations W comes into play when we define the semantics of the epistemic operator:

$$(\mathbf{K} C)^{I,W} = \bigcap_{J \in W} C^{J,W}$$

It would also be possible to allow the operator \mathbf{K} to occur in front of roles and to define the semantics of role expressions of the form $\mathbf{K} R$ analogously.

An epistemic interpretation (I, W) satisfies a rule knowledge base $k = (T, A, R)$ if it satisfies every axiom in T , every assertion in A , and every rule in R . An epistemic model for a rule knowledge base k is a maximal nonempty set W of first-order interpretations such that, for each $I \in W$, the epistemic interpretation (I, W) satisfies k .

4.4 An Example

The following small knowledge base is used to exemplify the usage of Rule-based DL language with set extension in maintaining integrity constraints. For simplicity we use strings to represent resources, but in real applications they can be replaced by URIs to avoid ambiguity.

TBox:

student \sqsubseteq *person*

graduateStudent \sqsubseteq *student*

\exists *enrolled.student* \sqsubseteq *class*

\exists *hasEmail.Email* \sqsubseteq *student*

webEnrolled \sqsubseteq *enrolled*

ABox:

person (*a*) *person* (*b*)

student(*a*) *student*(*b*)

graduateStudent(*a*)

class(*csc4710*) *class*(*csc6730*)

hasEmail(*a*, *a@gmail.com*) *hasEmail*(*b*, *b@gmail.com*)

enrolled(*csc4710*, *a*) *enrolled*(*csc4710*, *b*)

webEnrolled (*4710*, *a*)

4.5 Constraint List

In this section, we list 18 types of integrity constraints and show how our approach can properly handle these constraints, using the small example in section 4.4. These constraints are from reliable sources. Many of them are from a survey result in ([TDB+08]). In this survey, OWL engineers and ontologists were asked what integrity constraints they desired to use in their applications. Some constraints are from various literatures whose goal is to address these constraints. The rest of the constraints are derived from their relational database counterparts. A definition of a constraint is provided if the meaning of that constraint is not self-evident to the reader. Please note a derivable empty head will indicate a violation to that constraint.

1) Key Constraints

$\text{:- } \mathbf{K} \text{ hasEmail}(X, Y), \mathbf{K} \text{ hasEmail}(Z, Y), X \neq Z.$

2) Foreign Key Constraints

$\text{:- } \mathbf{K} \text{ student}(X), \text{ not } \text{person}(X).$

3) Functional Dependency: essentially the same as key constraints be properties are binary relationships in OWL.

$\text{:- } \mathbf{K} \text{ hasEmail}(X, Y), \mathbf{K} \text{ hasEmail}(Z, Y), X \triangleleft\triangleright Z.$

4) Max-cardinality Constraints: $\text{Max}(C, n, R): \{x \mid \#\{y \mid (x, y) \in R^{\text{IP}}\} \leq n\} \supseteq C^{\text{IC}}$

$\mathbf{K} \text{ student_set}(\text{csc4710}, \langle X \rangle) \text{ :- } \mathbf{K} \text{ enroll}(\text{csc4710}, X).$

$\text{:- } \mathbf{K} \text{ student_set}(\text{csc4710}, S), \mathbf{K} \text{ cardinality}(S, \text{Count}), \mathbf{K} \text{ count} > 40.$

5) Min-cardinality Constraints: $\text{Max}(C, n, R): \{x \mid \#\{y \mid (x, y) \in R^{\text{IP}}\} \geq n\} \supseteq C^{\text{IC}}$

$\mathbf{K} \text{ student_set}(\text{csc4710}, \langle X \rangle) \text{ :- } \mathbf{K} \text{ enroll}(\text{csc4710}, X).$

$\text{:- } \mathbf{K} \text{ student_set}(\text{csc4710}, S), \mathbf{K} \text{ cardinality}(S, \text{Count}), \mathbf{K} \text{ count} < 10.$

6) Functionality Constraints: $\text{Func}(C, Q): \{x \mid \#\{y \mid (x, y) \in R^{\text{IP}}\} \leq 1\} \supseteq C^{\text{IC}}$

$\mathbf{K} \text{ student_set}(\text{csc4710}, \langle X \rangle) \text{ :- } \mathbf{K} \text{ enroll}(\text{csc4710}, X).$

$\text{:- } \mathbf{K} \text{ student_set}(\text{csc4710}, S), \mathbf{K} \text{ cardinality}(S, \text{Count}), \mathbf{K} \text{ count} \geq 1.$

7) Totality Constraints: $\text{Total}(C, Q): \{x \mid \#\{y \mid (x, y) \in R^{\text{IP}}\} = 1\} \supseteq C^{\text{IC}}$

$\mathbf{K} \text{ student_set}(\text{csc4710}, \langle X \rangle) \text{ :- } \mathbf{K} \text{ enroll}(\text{csc4710}, X).$

$\text{:- } \mathbf{K} \text{ student_set}(\text{csc4710}, S), \mathbf{K} \text{ cardinality}(S, \text{Count}), \mathbf{K} \text{ count} \triangleleft\triangleright 1.$

8) SubProperty-chain Constraints: $\text{SubPChain}(C, p_1, \dots, p_n, q)$ enforces that, for each object o of type C , if there is a chain of properties p_1, \dots, p_n starting from o , then this chain always references a node that is also directly referenced via property q of o .

$\text{:- } \mathbf{K} \text{ class}(X), \mathbf{K} \text{ enrolled}(X, Y), \mathbf{K} \text{ hasEmail}(Y, Z), \text{ not } \text{collects}(X, Z).$

9) Singleton Constraints: $\text{Single}(C)$ enforces that there is exactly one object of class C .

$\mathbf{K} \text{ person_set}(\langle X \rangle) \text{ :- } \mathbf{K} \text{ person}(X).$

- :- \mathbf{K} person_set(csc4710, S), \mathbf{K} cardinality(S, Count), \mathbf{K} count > 1.
- 10) Sub-class Constraints:** $\text{SubC}(C,D) : C^{\text{IC}} \subseteq D^{\text{IC}}$
- :- \mathbf{K} student(X), *not* person(X).
- 11) Sub-property Constraints:** $\text{SubP}(R,S) : R^{\text{IP}} \subseteq S^{\text{IP}}$
- :- \mathbf{K} webEnrolled(X, Y), *not* enrolled(X, Y).
- 12) Property Domain Constraints:** $\text{PropD}(R, C) : \{x \mid \exists y : (x, y) \in R^{\text{IP}}\} \subseteq C^{\text{IC}}$
- :- \mathbf{K} enrolled(X, Y), *not* student(Y).
- 13) Property Range Constraints:** $\text{PropR}(R, C) : \{y \mid \exists x : (x, y) \in R^{\text{IP}}\} \subseteq C^{\text{IC}}$
- :- \mathbf{K} enrolled(X, Y), *not* class(X).
- 14) Expected Individual Type Constraints:** The declared type of a given individual in the instance data must meet the expectation of the referenced ontologies.
- :- \mathbf{K} enrolled(X, Y), *not* class(X).
- 15) Specific Individual Type Constraints:** The declared type of a given individual in the instance data must be the most specific one
- :- \mathbf{K} student(X), \mathbf{K} person(X).
- :- \mathbf{K} graduateStudent(X), \mathbf{K} person(X).
- :- \mathbf{K} graduateStudent(X), \mathbf{K} student(X).
- 16) Redundant Individual Type Constraints:** An individual cannot be explicitly declared to have both C and C's superclass as its type.
- :- \mathbf{K} student(X), \mathbf{K} person(X).
- 17) Uniqueness Constraints:** An instance that is expected to be unique cannot have two or more individuals in the data set.
- :- \mathbf{K} hasEmail(X, Z), \mathbf{K} hasEmail(Y, Z), $Y \neq Z$.

18) User-defined Constraints: A student cannot take more than 4 classes:

K class_set($\langle C \rangle$, X):- K enroll(C, X).

: - K class_set(S, X), K cardinality(S, Count), K count > 4.

Please note for this simple sample ontology, we see different integrity constraints possess the same expression. However, the semantics behind these constraints are totally different. Their differences should be evident as the ontology becomes more complex.

The purpose of the above list is not to exhaust the possible constraints our approach can handle, but rather to show the capability to express many useful integrity constraints in real applications without much of increase in computational complexity. Our goal is to cover usefulness and efficiency instead of completeness in IC research in OWL. Users are allowed to express application-specific constraints with this formalism as well.

CHAPTER 5 A HYBRID APPROACH TO CONSTRAINT REASONING IN BIO- ONTOLOGIES

This chapter continues focusing on maintaining integrity constraints within a single OWL, with implementation details on Bio-ontologies. The update we consider corresponds to the second process in the OWL lifecycle shown in figure 1.1. In this study, we examined characteristics of commonly used bio-ontologies. Then this chapter proposes a hybrid architecture integrating semantic knowledge base technologies and forward rule reasoning, called Jena2Drools, to better serve the purpose of maintaining integrity constraints in bio-ontologies. A design and a prototype implementation are introduced. As a preliminary result, our demonstration shows the feasibility of efficient, composable, and easy-to-use bio-ontology constraint checking method from expressive rules reasoning

5.1 Motivations

Although numerous ontologies, such as Disease Ontology¹, Symptom Ontology², Gene Ontology³, are available in OWL form, there is still a huge demand for developing more OWL ontologies for various purposes in Bio-medical field. We have surveyed 82 bio-medical ontologies that are publicly available from Ontology Lookup Service⁴ and Bio-portal⁵. Close analysis revealed some common features among these ontologies. In general, these ontologies vary in size, with the number of terms ranging from 10 to 679478. But almost all of them emphasize on concept classification in TBox, while having little to none individual description in ABox. This

¹ <http://disease-ontology.org/>

² http://symptomontologywiki.igs.umaryland.edu/wiki/index.php/Main_Page

³ <http://www.geneontology.org/>

⁴ <http://www.ebi.ac.uk/ontology-lookup/ontologyList.do>

⁵ <http://bioportal.bioontology.org/ontologies>

means the reasoning service for biomedical ontologies will mostly focus on TBox reasoning, instead of instance checking (i.e. checking whether an assertion is entailed by an ABox) or realization (i.e. finding the most specific concept that an individual belongs to) in ABox reasoning. The following is a snippet about Lymphocytes in OMIT⁶ (Ontology for MicroRNA Target) ontology. Its relationships with other classes are shown in figure 5.1 (Lymphocytes is in the green outlined box). An arrow edge denotes an *subClassOf* relation.

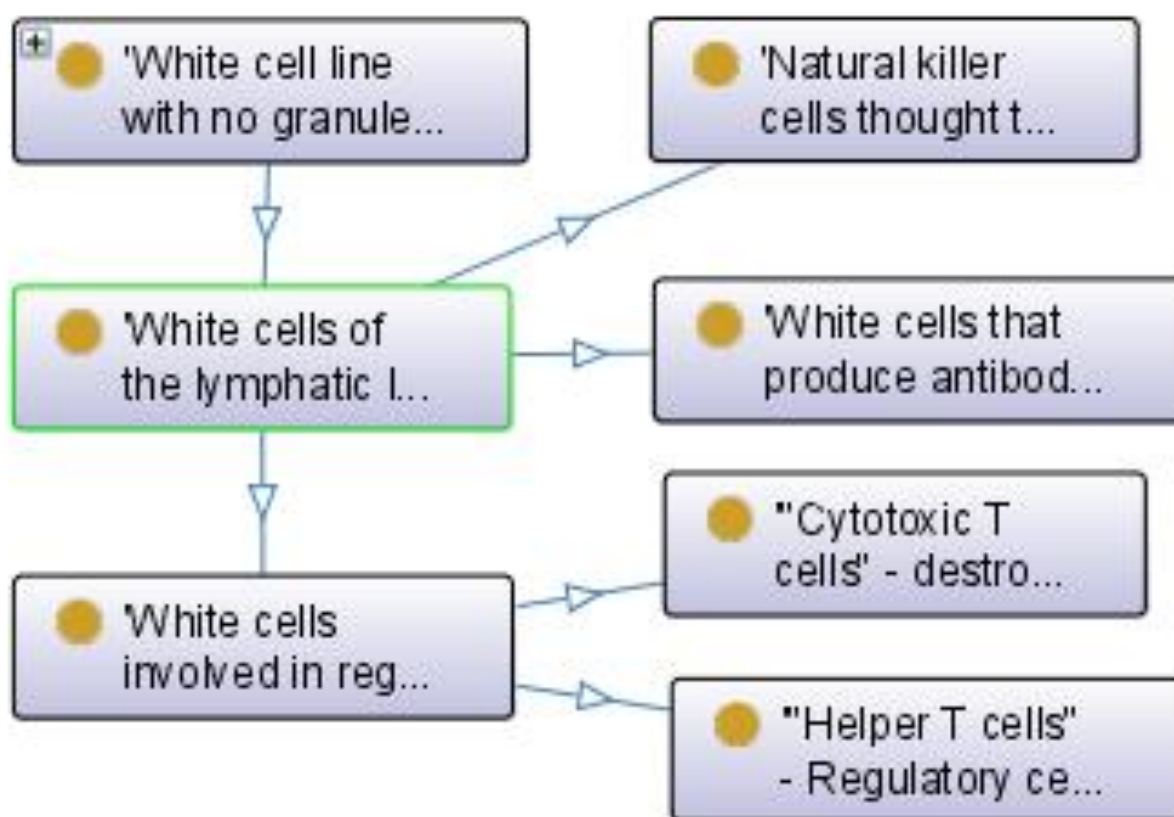


Figure 5.1 Lymphocytes and its relationship with other classes in OMIT

```

<owl:Class rdf:about="#Lymphocytes">
  <rdfs:label>White cells of the lymphatic lineage</rdfs:label>

```

⁶ <http://omit.cis.usouthal.edu/>

```

<rdfs:subClassOf rdf:resource="#Agranulocyte"/>
<owl:disjointWith rdf:resource="#Monocytes"/>
</owl:Class>

```

OMIT was proposed to handle the challenge of predicting the relationship between miRNAs and their targeting genes for cancer research. Similar to OMIT, the vast majority of biomedical ontologies listed on Ontology Lookup Service and BioPortal not only share the hierarchical structure, but also share their focus on terminology and relationship description in TBox. Table 5.1 summarizes statistics of some representative bio-ontologies:

Table 5.1 Statistics for sample bio-ontologies

Ontology	# of classes	#of individuals	# of properties
GO ³	37051	0	8
OMIT ¹⁰	319	0	18
CL ⁷	2003	0	9
Diagnostic Ont ⁸	96	6	9
DOID ¹	8623	0	19
TRANS ⁹	28	0	6
SYMP ²	934	0	9

5.2 Proposed Architecture

Here in this section we first describe our hybrid architecture in details, followed by discussions on implementation. Examples of Drools rules for constraint checking are also demonstrated here. We adopted the rule-based approach to handling the constraint problem. Here we

⁷ <https://lists.sourceforge.net/lists/listinfo/obo-cell-type>

⁸ <http://bioportal.bioontology.org/ontologies/3013>

⁹ <http://bioportal.bioontology.org/ontologies/1094>

present our preliminary results of implementing a hybrid system that combines knowledge base management and rule-based reasoner. To this end, we begin with Drools¹⁰, a powerful, expressive, and state-of-the-art rule-based reasoner that implements full and enhanced Rete-oo algorithm. On the other hand, Jena¹¹, a Java framework for building Semantic Web applications, is employed to manage bio-ontologies and to loosely interact with other components. It is essential that we provided a bridge component that faithfully translate OWL model into Java model for Drools reasoning. We argue this system suits bio-ontology constraint checking well, because limited to none ABox reasoning in bio-ontologies permits efficient reasoning about constraints written in more expressive Drools rule form.

5.2.1 Loosely coupled hybrid architecture

The prototype consists of five main components as depicted in figure 5.2.

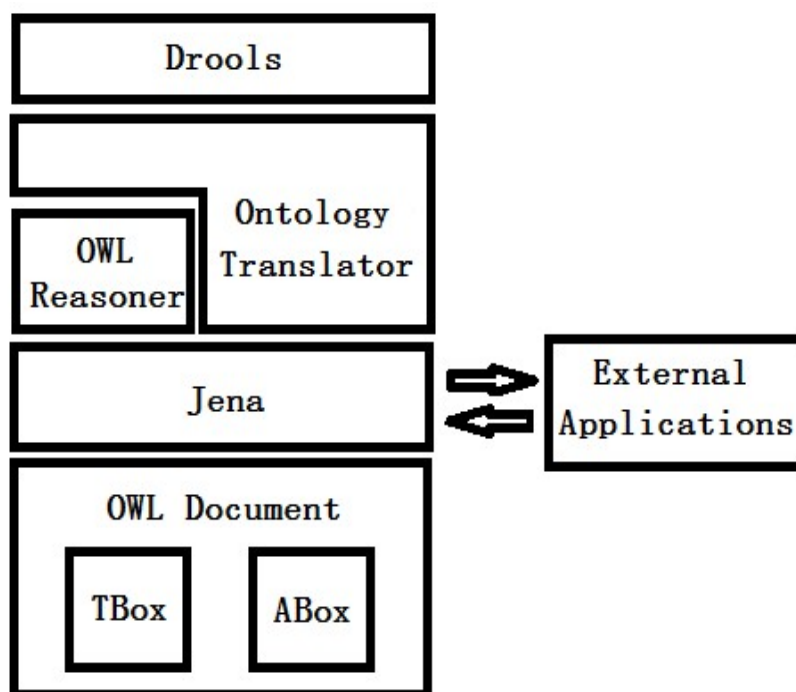


Figure 5.2 Loosely coupled hybrid architecture

¹⁰ <http://www.jboss.org/drools>

¹¹ <http://jena.apache.org/>

Drools is a mature and widely used production rule system in which rules are expressed in the following form:

when

[condition-list]

then

[action-list]

end

In our system, the actions in action-list can either be derivations of new facts from custom rules and knowledge base, or indications of integrity constraint violation if the premises are true, followed by reverting the ontology back to its previous valid state.

Compared to other reasoners including some that implement rule-based systems, Drools possesses the following advantages that make it an ideal tool for constraint checking in biomedical ontologies.

- Drools is interpreted under CWA in nature. This feature is aligned with the need of “closing” part of the ontology for constraint checking.
- An article [WRJ+09] surveyed 28 modern OWL reasoners, and found rule-based reasoners are fast, expressive, and easy to modularize. Indeed, in Drools, implied facts are derived and stored, so it is optimized for retrieval.
- Drools implements forward-chaining reasoning where the reception of new data automatically triggers new inferences. This makes the engine better suited to bio-ontologies whose updates are frequent over time.

- Drools permits negation in the rule premises, while negation is not supported in many rule formalisms and their reasoners, such as Jena and the W3C recommended Semantic Web Rule Language (SWRL).
- Drools is adapted to object oriented programming. In Drools, asserted facts are simply Java objects that can be accessed and changed through their methods and properties. This feature alleviates the complexity of integrating the rule engine with external applications.
- Drools allows rules to be expressed in natural language by mapping such sentences to Java objects. In addition, it supports online collaborative management of knowledge base. Thus Drools is friendly to domain experts that are not familiar with underlying implementation.

This system is capable of handling various species of Semantic Web documents, such as RDF, RDFs, and OWL. Since we are specifically interested in how Drools is suited to maintain integrity constraints in biomedical ontologies, we only tested with OWL documents, as these ontologies are available in OWL format. As shown in previous section, biomedical OWL documents usually contain no individuals in ABox, therefore, reasoning service in these ontologies is a pure T-box reasoning, such as subsumption (i.e. determining whether a concept is a sub-concept of another or not) and classification (i.e. consolidating and reorganizing a set of concepts into the minimal hierarchy). This special feature in biomedical ontology reasoning permits efficient validation of constraints that are intended to leverage the expressive power of Drools in this hybrid system. Please note our approach is also capable of handling ontologies with non-empty ABox, as shown later in this section.

Jena in this system is a Java framework for building Semantic Web applications. It contains Application Programming Interface (API) to read, process, write, manage, and query RDF model in various formats like XML, N-triples and Turtle formats. Ontology API for RDFS and OWL ontologies are also included. Ontologies can either be stored in memory, or stored in persistent storage such as TDB and SDB. Jena framework is well-known for its flexibility. Jena uses its interface to facilitate the connection and interoperability with other external applications, including third party reasoners. In addition, Jena comes with a very limited built-in OWL reasoner. For example, Jena does not support the well-founded negation, even in its simpler form like predicate-stratified negation. Such flexibility and limitation make the integration with more sophisticated rule engine like Drools desirable.

It is important to note that although rule engines and ontology reasoners are currently available, few of them are capable of handling rule reasoning and OWL reasoning at the same time. Besides Jena, other examples under this category include Hammurapi Rules¹², Algernon¹³, and SweetRules¹⁴. Despite their effort of unifying two types of reasoning under the same framework, they are less customizable, less widespread than the components of our loosely coupled solution. Loosely integrated approach, on the other hand, can combine the strengths from two systems, each of which is specialized at providing one aspect of the solution, while leaving opportunities for flexible integration with other applications. Compared to these unified systems, our hybrid system also has the advantage of operating under CWA that is suitable for constraint checking, as well as more expressive power such as negation-as-failure.

However, loose integration of ontology and rule-based system poses several issues, in order to handle mutual interactions between components, and to align their models.

¹² <http://www.hammurapi.com/>

¹³ <http://algernon-j.sourceforge.net/>

¹⁴ <http://sweetrules.semwebcentral.org/>

- The rule engine should be capable to import data via Jena from existing ontology, as well as to define ontology model internally.
- The rule engine should be capable of querying the information described and derived from the ontology model, since ontology usually contains implicit information.
- In general, the premises expressed in the rules should be defined using the concepts from the ontology TBox and evaluated against entities in the ontology ABox.

In our system, we handle these issues by building an ontology-to-facts translator, to mediate translation between OWL ontology and Drools facts. This translator is essential, because Jena internally stores OWL documents as RDF graph model, while Drools can only operate on Java object model. Thus this translator serves as a tool for the automatic translation of concepts, relations and individuals from an OWL document into Java classes, properties, and instances. The JavaBeans can then be directly asserted as facts into the working memory of Drools for forward-chaining reasoning. Note our ontology translator is capable of translating stored RDF model in Jena directly, as well as translating derived data from Jena built-in reasoner or other OWL reasoners.

The following algorithm details the mapping mechanism from ontology to Java facts. Please note that this mapping can also work with general ontologies whose ABox is not empty. Although this mapping can possibly map a TBox property and an ABox relation that is the instance of that TBox property to the same Java class, they can be differentiated by appending different labels during class instance creation.

Algorithm: Mapping from Semantic Model to Java ModelTBox:

For each distinct concept in TBox:

 Create a Java class with the same name as the concept;

For each distinct property in TBox:

 Create a Java class with the same name as the property; also, create two attributes for this class;

For each Subject-Object pair of this property in TBox:

 Create an object of this class; set two attributes to the name of Subject and Object, respectfully;

ABox:

For each individual in ABox:

 Create an object of the class that corresponds to the concept this individual belongs to;

For each relation in ABox:

 Create an object of the class that corresponds to the property this relation belongs to;

For implementation, the prototype used Java 1.6. The Drools rule engine 5.3 was used as the rule engine, and Jena 2.6.4 was used for ontology processing and management. To validate the effectiveness of our system on heterogeneous datasets, we tested on Breast Cancer Grading Ontology¹⁵, Breast Tissue Cell Line Ontology¹⁶, Cell Line Ontology¹⁷, Computational Neurosci-

¹⁵ <http://bioportal.bioontology.org/ontologies/1304>

¹⁶ <http://bioportal.bioontology.org/ontologies/1438>

ence Ontology¹⁸, Diagnostic Ontology¹², Ontology for Drug Discovery Investigations¹⁹, Gene Regulation Ontology²⁰, Infectious Disease Ontology²¹, Neomark Oral Cancer Ontology²², Neural Motor Recovery Ontology²³, and Ontology for MicroRNA Target Prediction. The number of classes in these datasets ranges from 30 to 8934. For demonstration, we show example of constraint checking by Drools using the OMIT ontology.

5.2.2 Examples of Constraint Checking

The following examples assume the hybrid system has already loaded the ontology into Jena, and performed the ontology to Java Model transformation. All prefixes are removed in these examples for the sake of better readability.

Example 5.1: Consider the rule below that specifies the constraint that a concept cannot be the subclass of two disjoint classes. In the setting where no Jena or OWL external reasoners are involved, Drools is still capable of derive all the implicit subclasses of a concept, instead of just the immediate subclass that is explicitly stated in ontology. Variables begins with a \$ sign.

rule “obtain all subclasses”

salience 10 //higher priority

when

subClassOf (\$s : subject, \$o : object)

exists ((subClassOf (\$o: subject, \$t : object) **or** (SubConceptOf(\$o, \$t)))

then

insert (new SubConceptOf(\$s, \$t))

¹⁷ <http://sourceforge.net/projects/clo-ontology/>

¹⁸ <https://github.com/INCF/Computational-Neurosciences-Ontology--C.N.O.->

¹⁹ <http://code.google.com/p/ddi-ontology/>

²⁰ <http://www.ebi.ac.uk/Rebholz-srv/GRO/GRO.html>

²¹ http://infectiousdiseaseontology.org/page/Main_Page

²² <http://www.neomark.eu/portal/>

²³ <http://cis.usouthal.edu/~huang/>

```

insert (new SubConceptOf($s, $o))
insert (new SubConceptOf($o, $t))

//Assume SubConceptOf class are created to store all subclass relationships
//explicitly

end

rule “check disjoint superclasses”

salience 5 //lower priority

when

    subConceptOf ($s : subject, $o1 : object)
    subConceptOf ($s : subject, $o2 : object)
    disJointClass($o1 : subject, $o2 : object)

then

    //raise a flag of constraint violation revert to previous state

end

```

subClassOf (\$s : subject, \$o : object) will match through all the objects of subClassOf class, and assign the value of property subject to \$s, and assign the value of corresponding property object to \$o. In implementation, Java Map data structure is used to avoid data duplication.

Example 5.2: Consider a real-time or near real-time biomedical application that uses ontology and reasoning as its information source. Since reasoning can be time-consuming thus not be able to meet the real-time requirement, it is desirable to explicitly store inferred knowledge to trade for efficiency. The example below is used to specify the constraint that every subclass relationship has to be explicitly stored. Upon any state change to the knowledge base, this constraint checking can be triggered. Assume prior to updates, the knowledge base is valid (i.e. all subclass

relationships are explicitly stored). This example demonstrates the expressive power of negation-as-failure, which is not available in rules like Jena Rules and SWRL.

```

rule "check implicit subclass relation"

salience 10

when

    SubClassOf ($s : subject, $o : object)

    SubClassOf ($o : subject, $t : object)

    not SubClassOf ($s : subject, $t : object)

then

    //raise a flag of constraint violation revert to previous state

End

```

Please note that example 5.2 is independent from example 1, so as their salience. Example 5.1 uses `SubConceptOf` to store explicit and derived subclass relationships for subsequent disjointness checking, while example 5.2 only check if every subclass relationship is captured explicitly in `SubClassOf`.

CHAPTER 6 IMPLEMENTATION AND EVALUATION OF AN IMPROVED JENA2DROOLS FOR GENERAL-PURPOSE CONSTRAINT CHECKING

In this chapter, we present an improved version of Jena2Drools that now can take Semantic Web data in common formats like RDFS, RDF, and OWL, for general-purpose constraint checking. In section 6.1, we introduce the improved version over the earlier of Jena2Drools described in Chapter 5. Then in section 6.2, we describe the evaluation methods that we have used on the well-known Lehigh University Benchmark (LUBM) for Semantic Web. A performance analysis of our system is then presented in section 6.3.

6.1 Jena2Drools 2.0

The improved Jena2Drools version 2.0 still adopts the same architecture used in the earlier version shown in figure 5.2. The major improvement over the earlier version is that Jena2Drools 2.0 now can accept any arbitrary syntactically legal Semantic Web data as input in common forms, including RDFS, RDF, and OWL. In contrast, the earlier version of Jena2Drools was tested with biomedical ontologies whose ABox is usually very small while their TBox mostly contains subsumption relationships.

The expansion of applicability comes from a vastly different approach to constructing the Java model from the Semantic Web model. In Jena2Drools 1.0, both concepts and properties are modeled as Java classes, and each class that represents a property contains two fields corresponding to the subject and object of the property. This approach suits simple properties like subsumption well. In the improved version, only concepts are modeled as Java classes. These classes are essentially object wrapper class that wraps around the Resources in the RDF model. In other words, these Java classes are Object-oriented representation that is internally still backed by the RDF model. Properties, on the other hand, are now modeled as fields in Java classes

whose represented concepts are the subjects of this property. The getters and setters provide means to access and change the backend RDF model through the object wrapper class. In this way, our RDF to Java translator can handle more complex Semantic Web constructs, and thus can be used as a part of the general-purpose constraint checking system.

The Jena2Drools 2.0 system reads the Semantic Web data files and user-specified integrity constraints as inputs. As the outputs, the user can choose to have this system report either a single violation or all the violations to constraints. The entire validation process can be further broken down into the following sequential steps:

- **Generating Class Files:** Jena2Drools 2.0 first reads schema files in RDFS or OWL format as inputs, and then constructs a RDF model out of these schema files. The concepts and properties in this model are then analyzed before automatic generation of Java class files from the RDF model. As described earlier, each outputted Java class is an object-oriented representation of a concept in the RDF model, while fields within each class relate to the properties of that concept. Since schema file updates are far less frequent than instance file, this class file generation step can be considered as a one-time overhead. Also, compared to instance files, the size of schema files is usually much smaller, in terms of the number of statements. Therefore, the time to generate the Java classes is small when compared with the time during the translation step where instance files are translated to Java objects.
- **Loading Instance Data:** The instance data in RDF or OWL are then loaded into this system. Usually the size of instance data varies from time to time, and it is a key factor that accounts for the loading time.

- **Translating RDF Model to Java Model:** This is the core step in the system. Jena2Drools 2.0 will look at the RDF model built from the schema and instance files, and map each resource in the RDF model to the automatically generated Java classes by creating corresponding Java object. These Java Objects are Plain Old Java Objects (POJOs) so that they can be subsequently consumed by Drools. This translation step is relatively time-consuming and its running time depends on the size of the instance data. However, it is important to note that this translation step only needs to run one time. Once the complete Java model (i.e. Java classes and their objects) is in the system, any updates to the RDF model will also be carried over accordingly to the Java model, without going over the entire translation step again.
- **Preparing Knowledge Base and Inserting Java Objects:** In this step, a default empty knowledge base is created for Drools. Then this knowledge base is populated with the Java objects created in last step. After insertion, this Drools knowledge base stores equivalent knowledge as in the RDF model.
- **Validating Knowledge Base with Integrity Constraints:** Integrity constraints are expressed as rules in Drools rule language. The rule language, together with its rule engine, can reason under Close World Assumption and Negation-as-Failure. User has the flexibility to choose whether to find all the violations or just one at a time. In addition to listing all violations, the system is also equipped with the capability of repairing the knowledge base.

6.2 Evaluation

To evaluate Jena2Drools2.0, we analyzed the performance of this system by using datasets of different sizes and modeling integrity constraints for the data. The total running time is broken down into four parts as described in last section: RDF load, RDF translation, knowledge base preparation and Java objects insertion, and finally validation. We also compared the performance of our system with OWL 2 DL-based system [Tao12] for constraint checking, using the same datasets and integrity constraints.

Datasets: The Lehigh University Benchmark (LUBM)²⁴ is a standard benchmark that is intended to evaluate Semantic Web systems in a systematic way over a large dataset that commits to a single realistic ontology in the university domain. Because the contents in the ontology is relevant to the integrity constraints being tested, here we quote the descriptions of the ontology from the LUBM website:

“(Class and property names are underlined. Class names are capitalized and property names in italic.)

In each University

- 15~25 Departments are *subOrganization* of the University

In each Department:

- 7~10 FullProfessors *worksFor* the Department
- 10~14 AssociateProfessors *worksFor* the Department
- 8~11 AssistantProfessors *worksFor* the Department
- 5~7 Lecturers *worksFor* the Department
- one of the FullProfessors is *headOf* the Department
- every Faculty is *teacherOf* 1~2 Courses
- every Faculty is *teacherOf* 1~2 GraduateCourses
- Courses taught by faculties are pairwise disjoint
- 10~20 ResearchGroups are *subOrganization* of the Department
- UndergraduateStudent : Faculty = 8~14 : 1
- GraduateStudent : Faculty = 3~4 : 1

²⁴ <http://swat.cse.lehigh.edu/projects/lubm/>

- every Student is *memberOf* the Department
- 1/5~1/4 of the GraduateStudents are chosen as TeachingAssistant for one Course
- The Courses the GraduateStudents are TeachingAssistant of are pairwise different
- 1/4~1/3 of the GraduateStudents are chosen as ResearchAssistant
- 1/5 of the UndergraduateStudents have a Professor as their *advisor*
- every GraduateStudent has a Professor as his *advisor*
- every UndergraduateStudent *takesCourse* 2~4 Courses
- every GraduateStudent *takesCourse* 1~3 GraduateCourses
- every FullProfessor is *publicationAuthor* of 15~20 Publications
- every AssociateProfessor is *publicationAuthor* of 10~18 Publications
- every AssistantProfessor is *publicationAuthor* of 5~10 Publications
- every Lecturer has 0~5 Publications
- every GraduateStudent co-authors 0~5 Publications with some Professors
- every Faculty has an *undergraduateDegreeFrom* a University,
a *mastersDegreeFrom* a University, and a *doctoralDegreeFrom* a University
- every GraduateStudent has an *undergraduateDegreeFrom* a University”

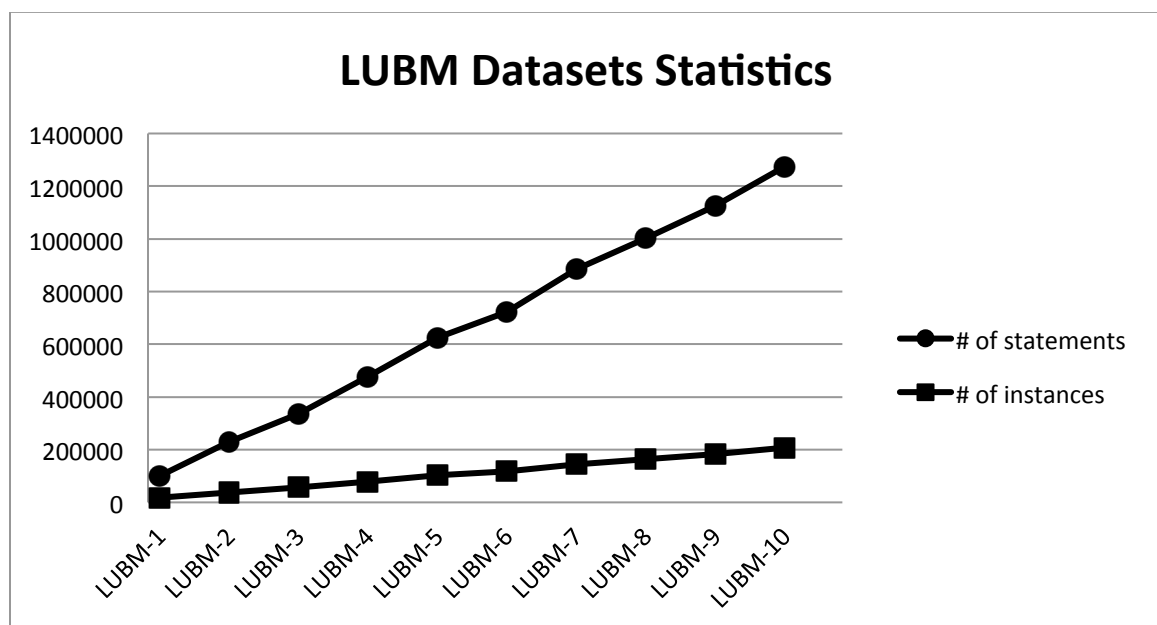


Figure 6.1 LUBM datasets statistics

We have used 10 LUBM datasets that represent 1 to 10 universities. LUBM provides data generator that produces synthetic OWL instance data over the University ontology described above. The size of the datasets ranges from around 100 thousand statements to more than 1.2 million statements. We also plotted the number of statements and the number of distinct Java objects generated by Jena2Drools over the number of universities in each dataset, and observed a linear relation between them.

Integrity Constraints: We modeled five representative integrity constraints shown below. Please note that IC1, IC4, and IC5 are taken from [Tao12] for performance comparison. IC2 and IC3 are added to demonstrate the variety of constraints Jena2Drools 2.0 can handle. These ICs are only for performance evaluation purpose. They vary in size and complexity, but they are not intended to represent all possible ICs.

IC 1: each publication has to be one of the following: Book, JournalArticle,

ConferencePaper, or TechnicalReport

rule "publication"

dialect "java"

when

 \$pub:Publication()

 eval(! (\$pub instanceof Book))

 eval(! (\$pub instanceof JournalArticle))

 eval(! (\$pub instanceof ConferencePaper))

 eval(! (\$pub instanceof TechnicalReport))

then

 System.out.println("Found one constraint violation with publication: " + \$pub);

```
drools.halt();
```

```
end
```

```
# IC 2: each author of an publication has to be a faculty
```

```
rule "publicationAuthor"
```

```
dialect "java"
```

```
when
```

```
    $pub:Publication()
```

```
    $author:Person() from $pub.getPublicationAuthor()
```

```
    eval(! ($author instanceof Faculty))
```

```
then
```

```
    System.out.println("Found one constraint violation with author: " + $author);
```

```
    drools.halt();
```

```
end
```

```
# IC 3: each faculty can not for the same university in which he got his doctoral degree
```

```
rule "degree"
```

```
dialect "java"
```

```
when
```

```
    $faculty:Faculty()
```

```
    $dep:Department() from $faculty.getWorksFor()
```

```
    $phdUniv:University() from $faculty.getDoctoralDegreeFrom()
```

```
    eval(("http://www." + $dep.toString()).substring($dep.toString().indexOf
```

```
        ("University"))) .equals($phdUniv.toString()))
```

```
then
```

```

System.out.println("Found one constraint violation with degree: faculty " +
    $faculty.getName() + " works for and got PHD degree from the same uni-
    versity "+ $phdUniv);

drools.halt();

```

end

IC 4: full professors only teach graduate course

rule "fullProfessorGraduateCourse"

when

```

$fullPro:FullProfessor()
$course:Course() from $fullPro.getTeacherOf();
eval(! ($course instanceof GraduateCourse))

```

then

```

System.out.println( "Found one constraint viololation with full professor + " +
    $fullPro + " teaching " + $course);

drools.halt();

```

end

IC 5: assistant professors cannot teach less than 3 courses

rule "assistantProfessor3course"

when

```

$assistantPro:AssistantProfessor()
eval($assistantPro.getTeacherOf().size() < 3);

```

then

```

System.out.println( "Found one constraint violation with assistant professor " +
    $assistantPro + " teaching only " + $assistantPro.getTeacherOf().size() +"
    courses.");

drools.halt();

end

```

6.3 Performance Analysis

To gain insights into the performance of Jena2Drools 2.0, we evaluated the RDF load time, RDF instance translation time, knowledge base preparation and insertion time, and validation time. In this section, we present our findings along with analysis of the results.

- **Load, Translation, and Preparation Time:** Figure 6.2 below shows the running time for RDF load, RDF translation, and knowledge base preparation and insertion.

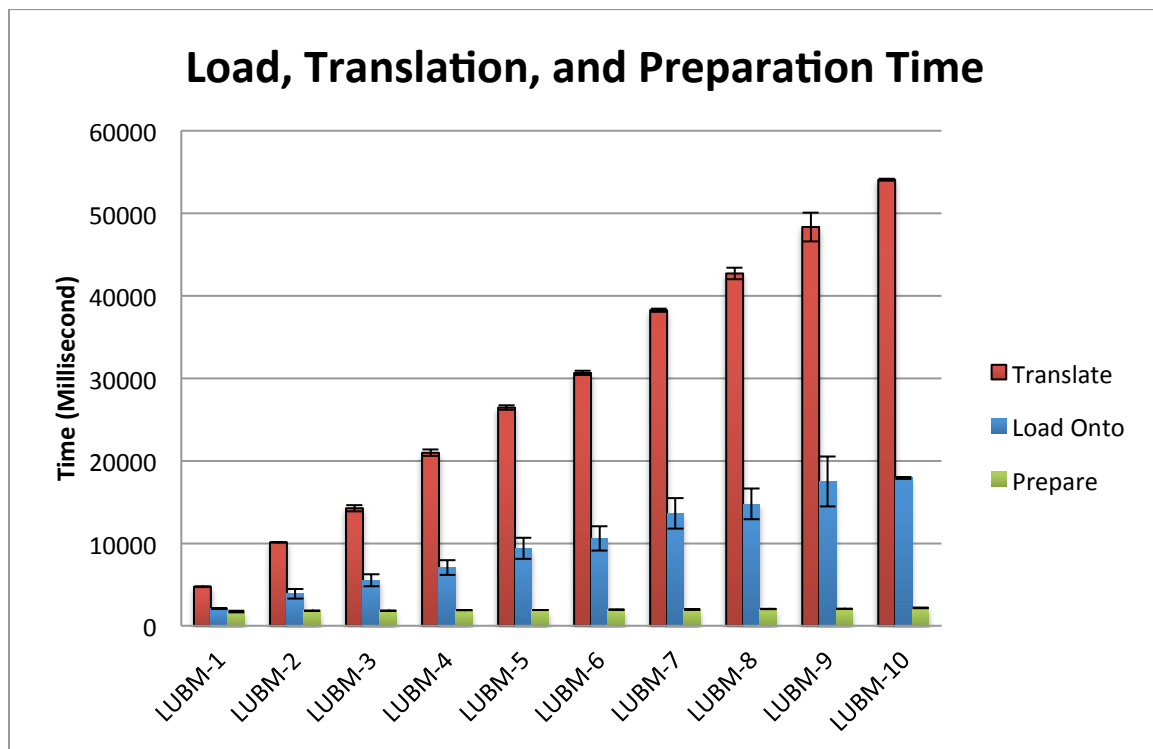


Figure 6.2 RDF load, RDF instance translation, and knowledgebase preparation and insertion time.

Both RDF load time and RDF translation time grow linearly as the size of the datasets grows, while knowledgebase preparation and insertion time remains flat. It is because inserting Java facts into Drools knowledgebase is so fast that its time becomes insignificant to the constant knowledgebase initialization time. It is clear that translating from instance data in the RDF model into Java object facts takes more time than the other two steps combined. This reflects the fact that the translation part is where most of the calculations come from. But note that, although translating 1.2 million RDF statements into Java object facts takes about 55 seconds, once these objects are created and stored, Jena2Drools will never need to translate those 1.2 million statements again.

- **Validation Time:** Next set of experiments look at the validation time only (i.e. the time that the system take to reason against rules and facts) and the entire system running time (i.e. from loading the input all the way till outputting results). We then compare these results with the ones in DL-based system from [Tao12] for integrity constraint 1, 4 and 5, as shown in figure 6.3, 6.4, and 6.5, respectively. This DL-based system represents the most recent development in the integrity constraint research of Semantic Web. In these tests, both systems are required to find out all the violations to the constraint specified. There are several interesting observations in these tests. When only validation time is considered, Jena2Drools outperformed DL-based system in all three cases. In addition, the growth rate for validation time in Jena2Drools is also lower than the other system. The most drastic contrasts come from IC4 and IC5. For example, in the case of IC4, validation time for DL-based system grew from about 2000 milliseconds (LUBM-1) to more than 15000 milliseconds (LUBM-2),

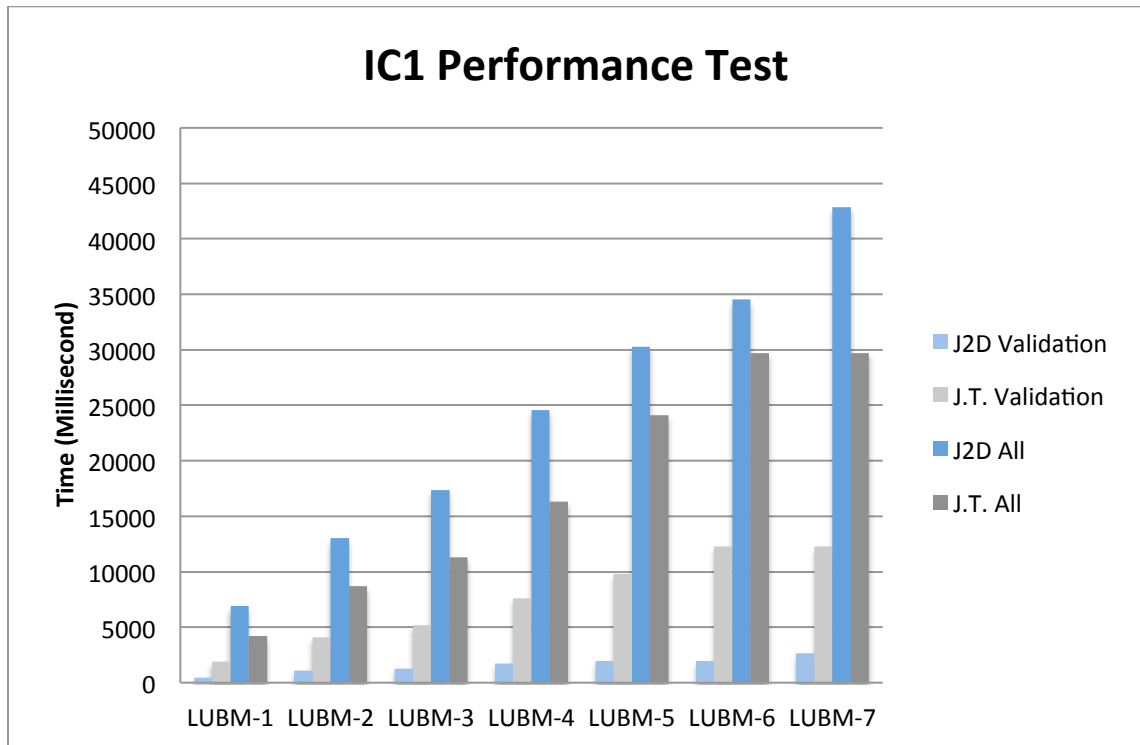


Figure 6.3 Comparing validation time and total running time for integrity constraint 1.

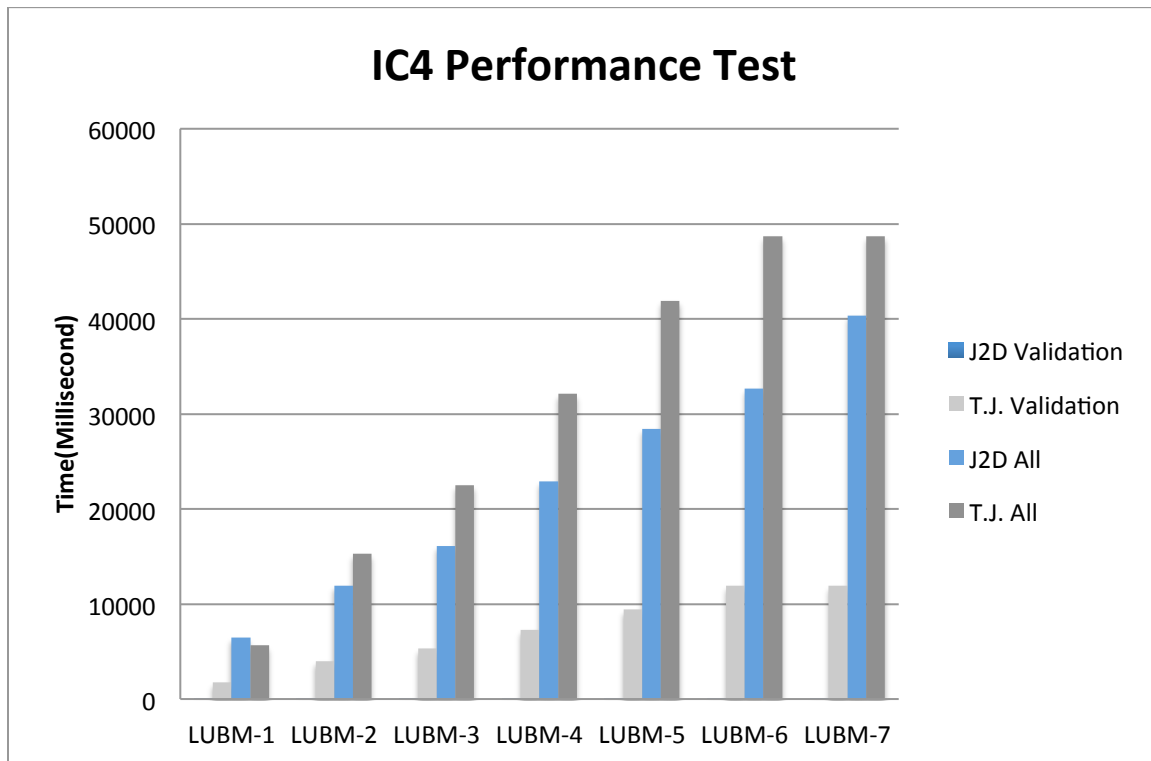


Figure 6.4 Comparing validation time and total running time for integrity constraint 4.

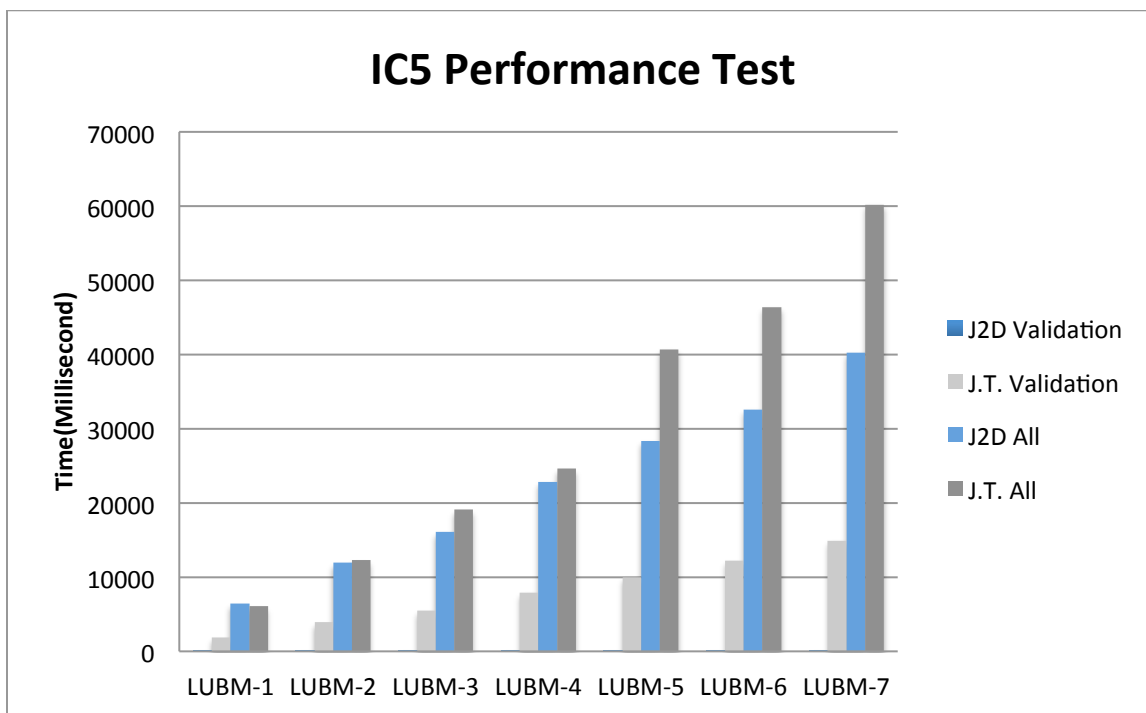


Figure 6.5 Comparing validation time and total running time for integrity constraint 5.

while validation time for Jena2Drools only grew from about 30 milliseconds to about 100 milliseconds. As introduced in previous chapter, Drools is a very mature business rule system in which many query answering and rule executing tasks have been extremely optimized for Java model. For total running time, Jena2Drools outperformed DL-based system in IC4 and IC5, but not IC1. Note that not only the size of datasets is important to validation time, but also the complexity of queries/rules (thus the type of queries/rules that a system is good at) and the total number of violations in the datasets can significantly affect validation time. In IC1, the number of violations is much larger than the number of violations in IC4 and IC5, thus resulting in a higher validation time and total running time in this case for Jena2Drools. In both systems, the actual validation time accounts for less than one third of the total running time. It is important to note that in the DL-based system, in addition to validation time, the to-

tal running time also contains an extra service called justification of axiom entailment (i.e. the minimal sets of axioms in the knowledgebase that are sufficient to produce an entailment in OWL DL). In summary, when the RDF instance data load is infrequent but updates are frequent (thus the need to validate the RDF model after updates), Jena2Drools can be the preferable system.

- **Single Violation versus All Violations:** Jena2Drools comes with a feature to turn on or off finding all violations. When all-violation feature is off, Jena2Drools will only look for and return for one violation instance and then stop. This feature becomes handy when ontology engineer only wants to perform a quick check to see whether the Semantic Web data is in compliance with constraints, or only want to find and fix one violation at a time.

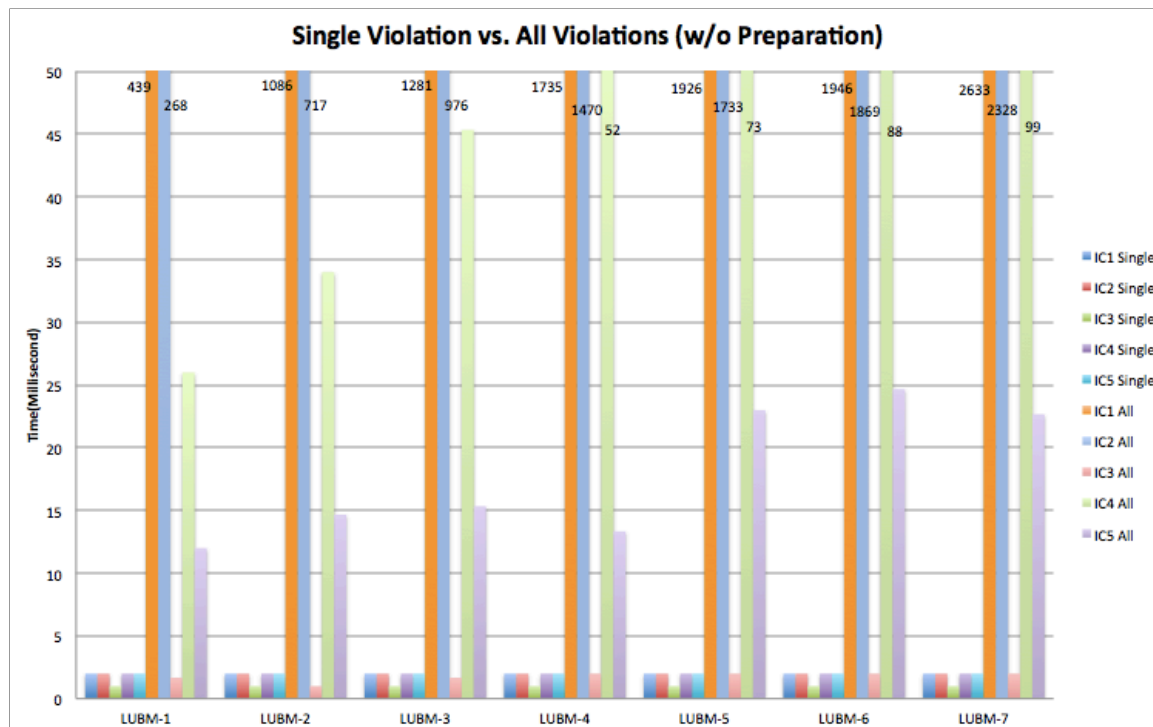


Figure 6.6 Validation time comparison between finding a single violation and finding all violations, when load, translation, and preparation time is not taken into account.

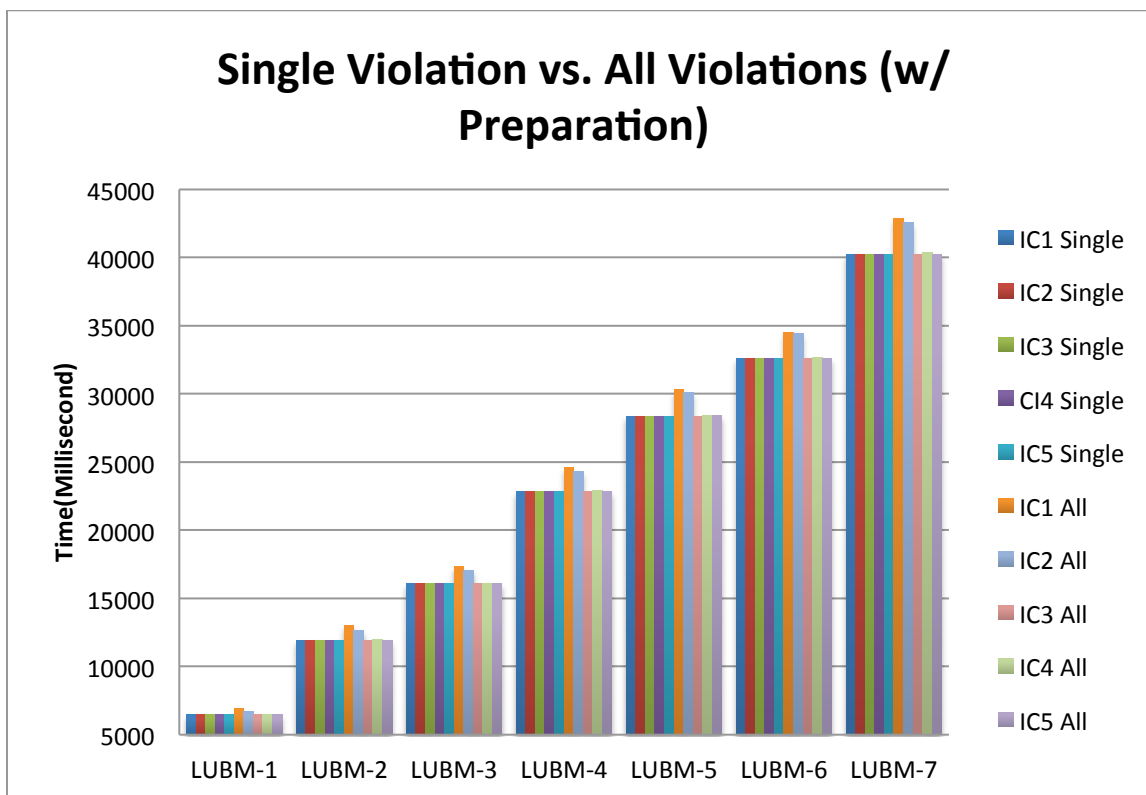


Figure 6.7 Validation time comparison between finding a single violation and finding all violations, when load, translation, and preparation time is taken into account.

Figure 6.6 compares the validation time only between finding only a single violation and finding all the violation, for all five modeled integrity constraints, while Figure 6.7 compares the total running time. In figure 6.6, it is expected to see the time to find all violation largely depends on the size of datasets. On the other hand, the time to find a single violation is insensitive to the size of datasets. It is reasonable because these datasets are generated incrementally. The first violation to be found in LUBM-1 will also present and be the first to be found in LUBM-7. Figure 6.6 also agrees with the argument that the number of violations (thus finding all of them) is a critical factor for validation time. When looking at the total running time, whether or not to turn on the all-violation feature becomes insignificant to the running time. This is because

when compared with the combination of the load, translation and preparation time, validation time, whether the time to find single validation or all validations, is of one even several orders of magnitude less.

CHAPTER 7 MAINTAINING INTEGRITY CONSTRAINTS IN MULTIPLE DISTRIBUTED OWLS

Due to the nature of knowledge bases, ontologies tend to be very large, distributed, and interconnected. Thus, maintaining constraints and enforcing data consistency for a group of ontologies become very challenging. In addition, frequent updates on ontologies necessitate an automatic approach to checking for potential constraint violations before any change takes place.

In this section, we focus on solving the integrity constraint issue for process 3 in OWL lifecycle shown in figure 1.1. Based on the same Rule-based DL language with set extension formalism we have introduced in chapter 4, we can easily extend it to express global constraints that span multiple ontologies. We conducted a pioneer study and presented a framework for checking global constraints and ensuring integrity on data that span multiple ontologies. As an update is issued to a single site, global constraints that can be potentially violated are broken down into sub-constraints that only involve a small subset of ontologies. The checking of sub-constraints runs effectively in parallel and returns results about each subset. The collection of these results determines the violation of global constraints.

7.1 Motivations

As an initial investigation as well as an evidence to show ontologies are interdependent and dynamic, we explored a group of well-established and well-known ontologies from the biomedical field. As a result, we drew the following conclusions.

7.1.1 *Biomedical ontologies are heavily interconnected*

We have investigated 81 bio-ontologies from The Open Biological and Biomedical Ontologies and the Ontology Lookup Service. For each of the ontology, we have examined and

parsed its data file in order to obtain interdependency information. As shown in figure 6.1, each directed edge indicates a reference from the origin to the destination. There is also a decimal number associated with each individual edge, representing the percentage of the total number of terms in that origin ontology reference to the destination ontology. The result shows 218 directed edges in the graph. This means in average, each biomedical ontology references to/depends on three other bio-ontologies. For the simplicity of the graph, we took off self-referenced edges, as well as edges pointing to ontologies other than the 81 ontologies we have examined. Taking these factors into account, the interdependencies between biomedical ontologies are much more complicated than what is shown in figure 6.1. Therefore, it is safe to say that the entire biomedical ontology networks are heavily interconnected. This result is not surprising at all because one of the purposes of ontology is to promote information exchange and reuse.

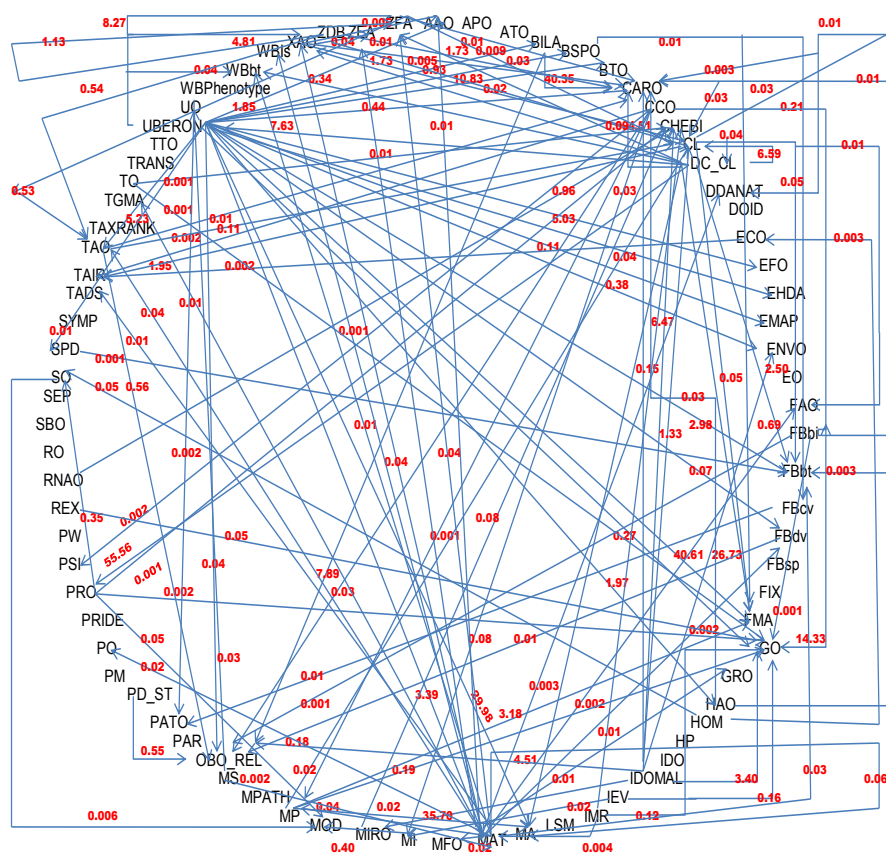


Figure 7.1 Interdependency between Biomedical Ontologies

7.1.2 Biomedical ontologies are updated frequently

We further scrutinized 45 biomedical ontologies whose update logs are immediately available. We were interested in information about update frequency and update scale. Figure 6.2 shows the averaged number of updates in a year for each bio-ontology. Although some ontologies in the set do not change very often, the majority of these ontologies updates at least 6 times a year, making the average update frequency 21.31 times/year for the entire set. In figure 6.3, we measured the averaged number of lines that are involved in each update for every bio-ontology in the set. The result gives a 98484.55 lines/update as the highest 0 lines/update as the lowest, and 7010.57 lines/update as the average for the whole set of biomedical ontologies. In short, biomedical ontologies are updated rapidly over time, with each update involving great amount of changes.

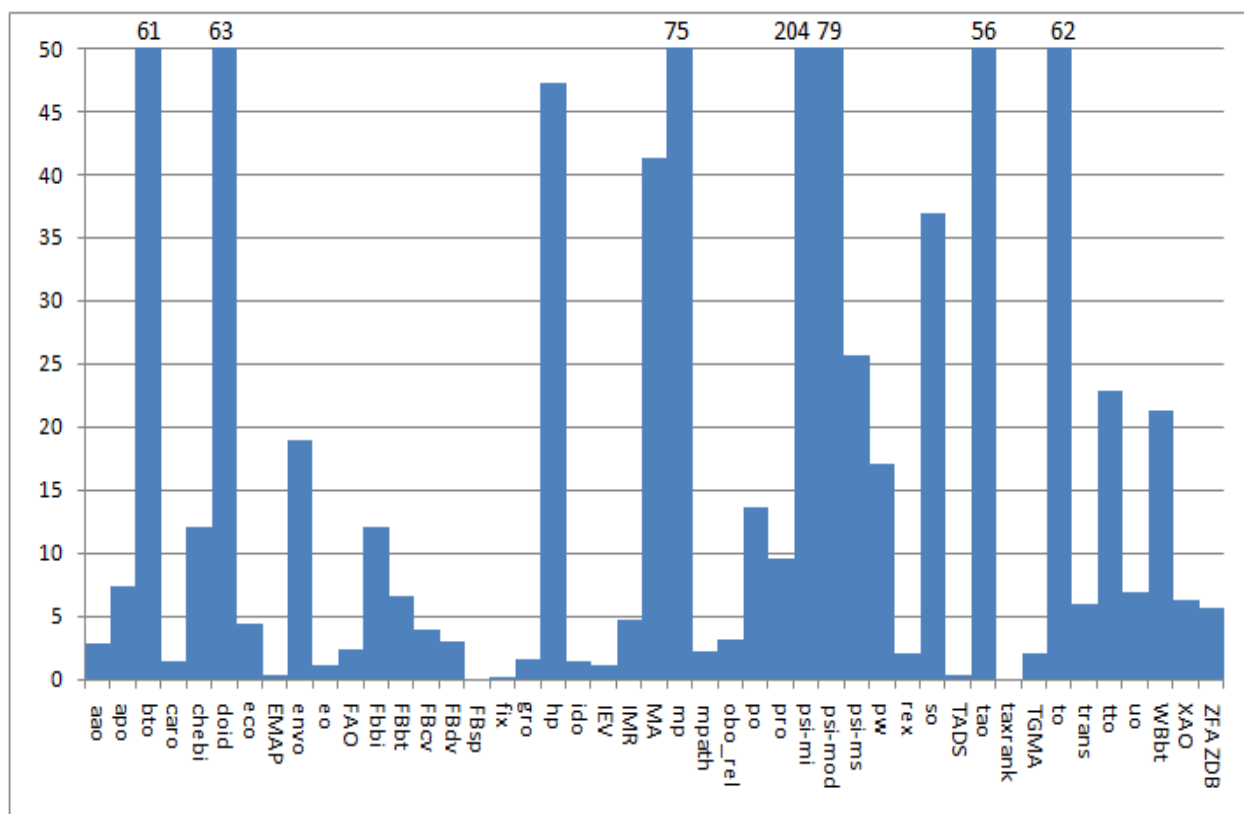


Figure 7.2 Biomedical Ontologies update frequency (updates/year)

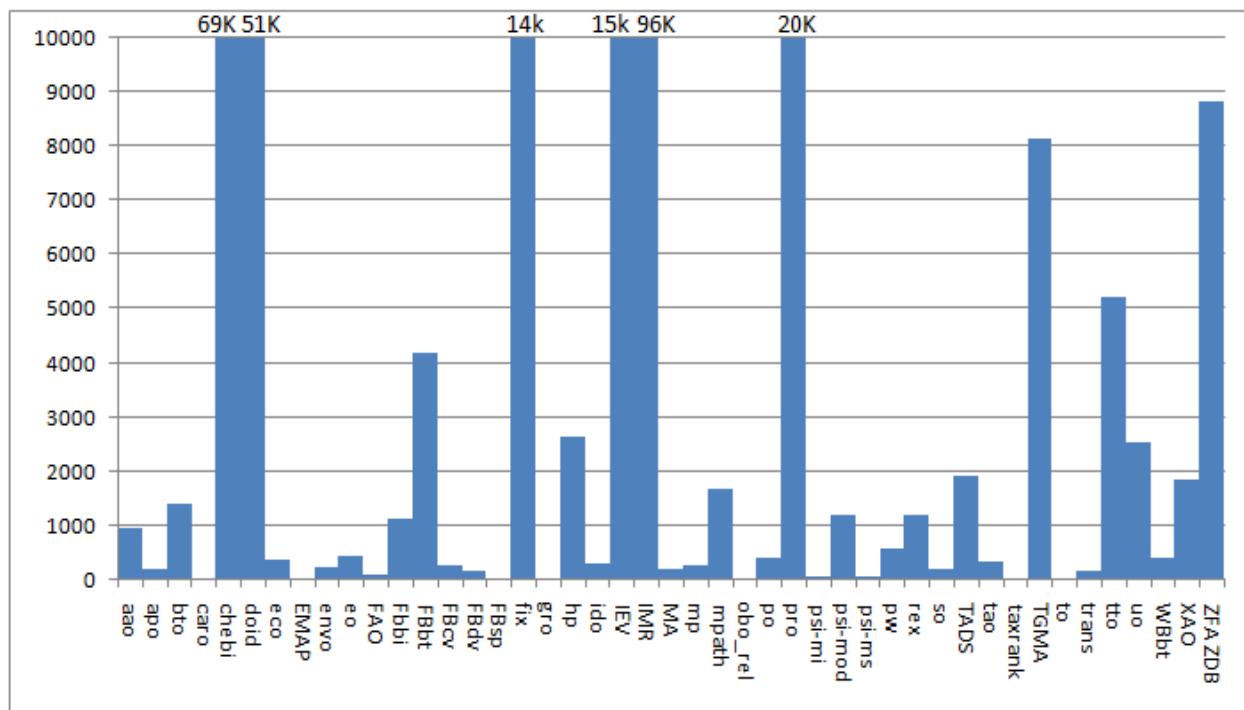


Figure 7.3 Biomedical Ontologies update scale (lines/update)

7.2 A framework for distributed ontologies

Throughout the rest of section, we will illustrate our framework using the following small example.

- **Patient Ontology (PO):** A doctor's office maintains an ontology that captures various types of *patients* and their properties such as *healthPlan*, etc.
- **Treatment Ontology (TO):** A hospital keeps an ontology that records *treatments* and their properties such as *doctor*, *treatedDisease* etc.
- **Claim Ontology (CO):** A health insurance company stores an ontology that describes patients' *claims* and their properties like *patient* who files the claim, the *treatment* he/she files for, and the amount of money for the claim, etc.

The following texts along with figure 6.4 describe the schema and contents of these three ontologies. For the sake of simplicity, we designate some information such as *disease* as literals.

In practice, unique URIs are usually used as identifiers to avoid ambiguity. Note that it is acceptable for an ontology to have missing information because of OWA.

Table 7.1 Contents of Patient Ontology, Treatment Ontology, and Claim Ontology.

<u>PO</u>	<u>TO</u>	<u>CO</u>
TBox: <i>pediatricPatient</i> \sqsubseteq <i>patient</i>	TBox: <i>drugTreatment</i> \sqsubseteq <i>treatment</i>	TBox:
ABox: <i>pediatricPatient</i> (Tommy), <i>patient</i> (Tom), <i>hasPlan</i> (Tom, A)	ABox: <i>treatment</i> (trt1) <i>hasDoctor</i> (trt1, Bill), <i>treatDis-</i> <i>ease</i> (trt1,smallpox),	ABox: <i>hasPatient</i> (claim1, Tom), <i>hasTreatment</i> (claim1, trt1)

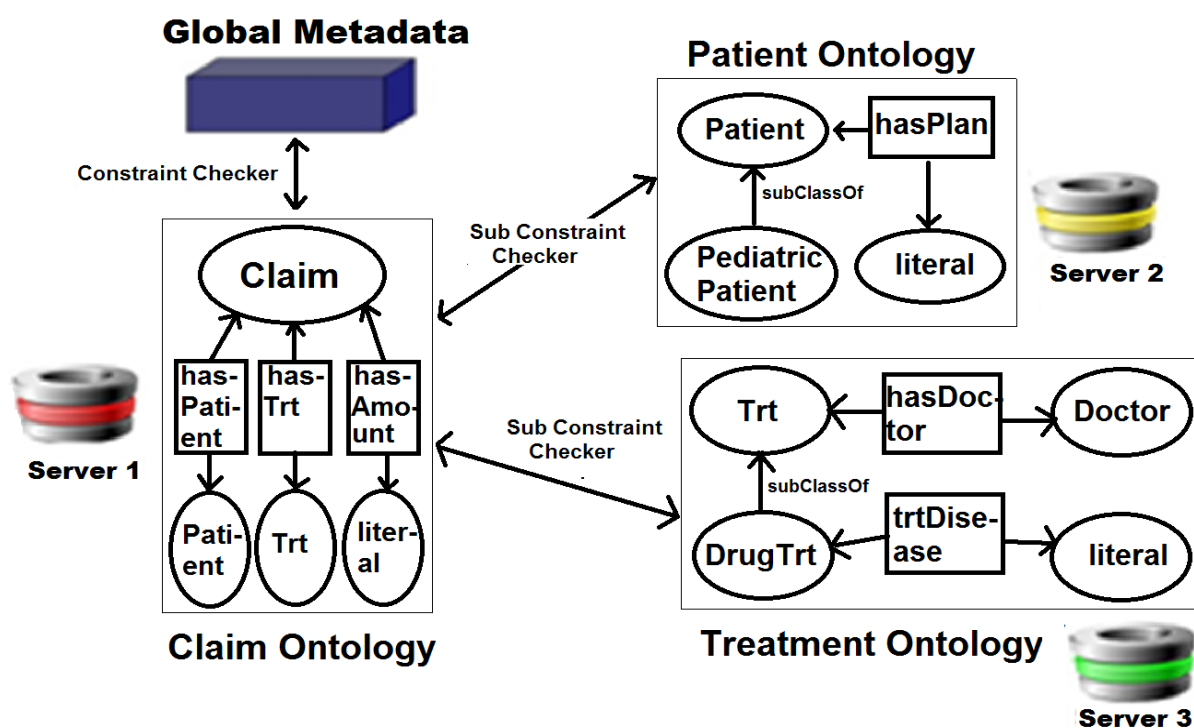


Figure 7.4 Ontology Schema and Constraint Checking Architecture

Global Constraints: (1) Semantic constraint. Any patient with plan 'A' cannot claim more than 5000 for smallpox. (2) OWL-style constraint. A useful OWL-style constraint called Specific Individual Type ([TDB+08]) constraint requires that the explicitly declared individual of a concept or relationship (property) in the instance data must be the most specific one. As an example, we are trying to enforce this constraint on the *hasPatient* property in Claim Ontology.

Since $hasPatient(claim1, Tom)$ is explicitly declared, we cannot insert $hasPatient(claim1, X)$ where X is Tom 's alias in the super or sub class of concept $patient$, otherwise there must be a claim-patient pair that is not the most specific one. In other words, a person is not allowed to file one claim using two identities: $patient$ and $pediatricPatient$.

These two global constraints can be formulated as rules in Logic Programming as follows:

C1:

$:- K hasAmount(x, y), K hasPatient(x, z), K hasTrt(x, w), K trtDisease(w, 'smallpox'), K hasPlan(z, 'A'), y > 5000.$

C2:

$:- K hasPatient(x, y), K hasPatient(x, z), K Patient(y), K pediatricPatient(z), y = z.$

First we illustrate the system architecture using figure 5.4. Assume we are trying to insert (1) $hasAmount(claim1, 7000)$ (2) $hasPatient(claim1, Tommy)$ into Claim Ontology in sequence. We further assume Tom and $Tommy$ actually refer to the same entity (so that Global Constraint 2 will be violated). This is possible in OWL because OWL uses Non-Unique Name Assumption. The constraint checker that resides on Server 1 is responsible for communicating with the Global Metadatabase whenever an update occurs to the Claim Ontology.

The Global Metadatabase is a central repository of site information that describes where each ontology source resides. It also contains domain information about metadata descriptions for instances of all ontology sources as well as the global constraints C_1, \dots, C_n . Each biomedical ontology registers itself to the Global Metadatabase with its hierarchical concept structures and its term compositions as well.

The constraint checker will first consult Global Metadata to collect all global constraints that could be possibly violated. All these global constraints will be examined one at a time. In our case, there are two global constraints and they can be potentially violated. Each global constraint is decomposed into several sub constraints for parallel validations, with each one of the sub constraints targeting at a small subset of all the ontologies involved in the global constraint. Take our first global constraint C_1 as an example. Upon inserting $hasAmount(claim1, 7000)$, the values of corresponding amount, patient, treatment will be available locally at server 1. Subsequently, one sub constraint checker will be dispatched to check whether **that** *treatment* is for disease smallpox in the Treatment Ontology, and return true if it is the case. Similarly, another sub constraint checker is dispatched to Patient Ontology in parallel to see if **that** *patient* has health plan 'A'. In the meanwhile, **that** *amount* value will be checked locally in Claim Ontology to see whether it exceeds 5000. Finally, based on the results brought back by these sub constraint checkers, global constraint violation can be determined by checking if every predicate in C_1 has been met. If they are met, then an empty rule head is derivable, thus the global constraint C_1 is violated. Therefore we have to reject the insertion of $hasAmount(claim1, 7000)$ and flag an error message to user. In this way, we prevent the communications to bring the entire Patient Ontology and Treatment Ontology to Server 1. Reader is encouraged to follow similar analysis for C_2 .

Figure 6.5 gives a detailed and decomposed structure of the Constraint Checking methodology. Each module in the constraint checker is explained as follows:

- Update Parser: Parses a user specified update, and return involved ontology objects.
- Metadata Extractor: Extract the set of global constraints that could be potentially violated by the update statement.

- Constraint Planner: runs an effective algorithm to generate sub constraints that will be dispatched to remote ontology sites.
- Constraint Optimizer: reorganizes the order of sub constraints in order to achieve higher efficiency.
- Constraint Executor: Execute sub constraints in parallel, and made decisions about the global constraint upon receiving the results of sub constraints.

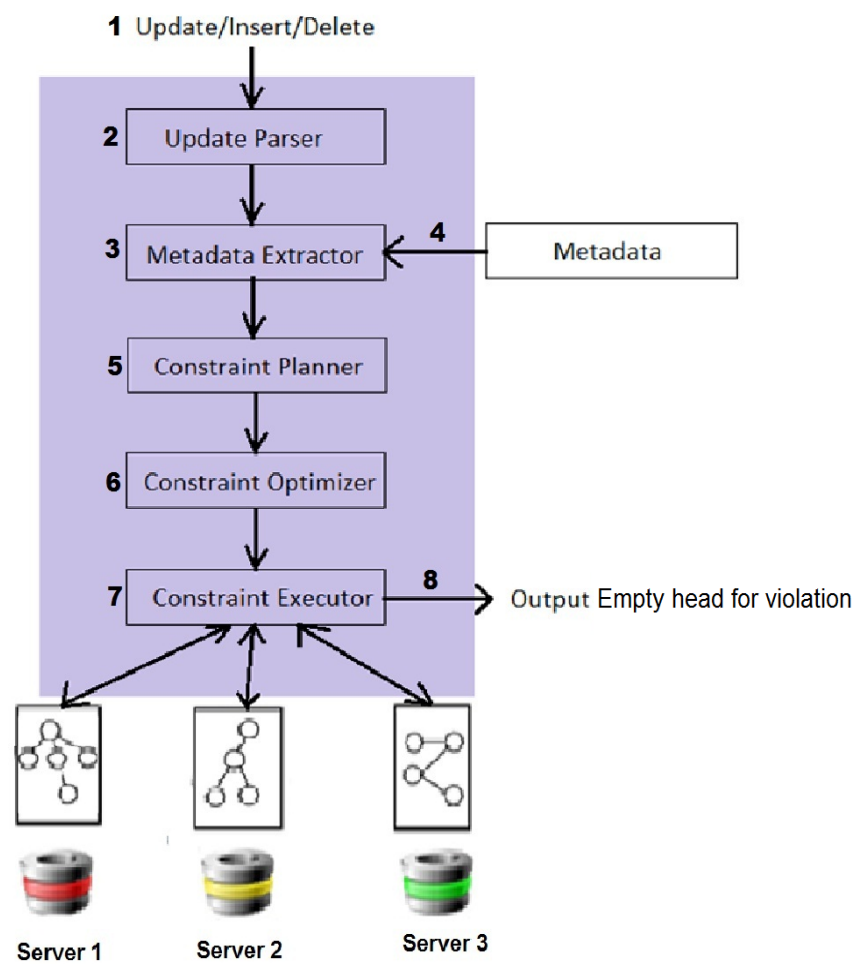


Figure 7.5 Internal Architecture of Constraint Checker

The entire process of global constraint checking can be further decomposed into the following eight steps. Consider the three ontologies and two global constraints we have introduced at the beginning of this section. The Specific Individual Type Constraint can be formalized by

adopting the notations used in ([TDB+08]). For the sake of argument, suppose we have the following generalized class hierarchy in the Patient Ontology shown in figure 6.6. An arrow from P_m to P_n indicates a *subClassOf* relationship between these two classes. In other words, P_m is subsumed by P_n . The Specific Individual Type Constraint posed on *hasPatient* property in Claim Ontology can be formally described as the following:

for every pair (P_m, P_n) where P_m is a direct or indirect subclass of P_n , we put one rule for this pair:

$\text{:- } K \text{ hasPatient}(x, y), K \text{ hasPatient}(x, z), K P_m(y), K P_n(z), y=z.$

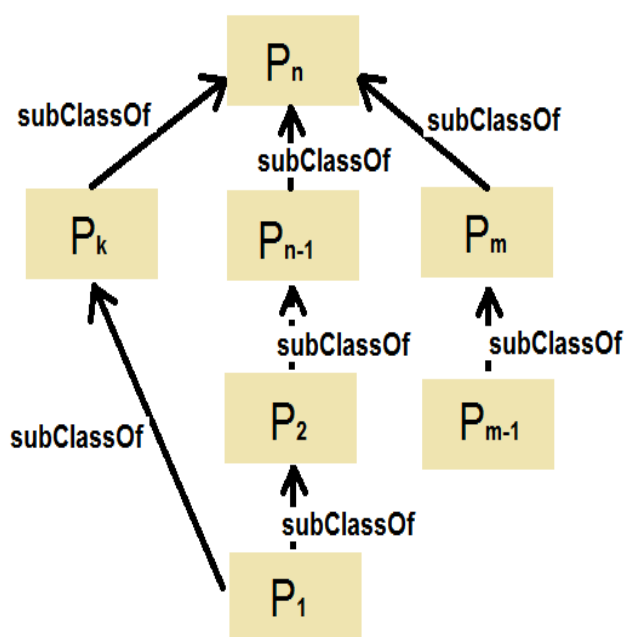


Figure 7.6 A Generalized Class Hierarchy for Patient Ontology

Each of the sub-goal in a rule can be easily translated to a query in standard OWL query language SPARQL. To finally determine whether an update violates this constraint, the Constraint Executor will conjunct a set of SPARQL queries. Each of the ask query represents a sub constraint and returns a value to the Constraint Executor.

From here we will provide a step-by-step procedure to show how each global constraint is automatically checked upon updates.

Step 1

The user of the Claim Ontology tries to issue a change to his local ontology. This change can be an insertion, a deletion, or an update statement. In most biomedical ontologies, updates are handled by a deletion followed by an insertion. For our small example, user issues the following insertion statements to Server 1:

- 1) insert *hasAmount(claim1, 7000)*
- 2) insert *hasPatient(claim1, Tommy)*

Due to the fact that there is no equivalent SQL insertion statement in the OWL query language SPARQL, this insertion can be executed by directly using the available API. It is easy to try and find out insertion (1) will not affect global constraint C_2 and insertion (2) will not affect global constraint C_1 . Thus, for demonstration purpose, we group these two insertions together in our stepwise discussion as if they are a single insertion. After the insertion(s), we will have the following Claim Ontology;

Table 7.2 The updated content of Claim Ontology

<p>CO</p> <p>TBox:</p> <p>ABox:</p> <p><i>hasPatient(claim1, Tom),</i></p> <p><i>hasTreatment(claim1, trt1),</i></p> <p><i>hasAmount(claim1, 7000),</i></p> <p><i>hasPatient(claim1, Tommy)</i></p>
--

Our goal is to check if this updated Claim Ontology still satisfies the global constraints.

Step 2 (Update Parser)

In this step, the Update Parser identifies which part of ontology is affected by the update statement issued by the user. In the case of insertion, new terms are added to the ontology. The output of this step will be an ontology object list that contains updated values of each item.

Ontology Object List = {Claim Ontology (patient = 'Tommy', claim = 'claim1', amount = '7000')}

Step 3 (Metadatabase Extractor)

By taking the output from the Update Parser, the site that is being affected (Server 1) will send the ontology object list to the Global Metadatabase. The Metadatabase Extractor will take as input the ontology object list, consult the domain information stored in the metadatabase, and return a Constraint-Source Table (CST) in the following format:

$$\text{CST}(C_i) = \langle C_i, \text{list}(S_j) \rangle$$

where C_i is the Global Constraint Identifier, and $\text{list}(S_j)$ is the list of ontology sources that are being affected by the constraint C_i . The CST captures which global constraints could be potentially violated and which ontologies are involved for each of the global constraint. For our small example, the server 1 will receive the following CST shown in table 6.3.

Table 7.3 Constraint-source table

C_i	List(S_i)
C_1 (Semantic Constraint)	(S_1, S_2, S_3)
C_2 (Specific Type Constraint)	(S_1, S_2)

Step 4

The constraint planner residing on Server 1 receives the CST from Metadatabase Extractor.

Step 5 (Constraint Planner)

The Constraint Planner, based on Constraint Planning Algorithm is the core part of the entire framework. It takes the CST from previous step as the input, and produces the Constraint Planning Table (CPT) in the following format:

$$\text{CPT}(C_i) = \langle C_i, \text{list}(C_{ik}, S_j), \text{and-list}(C_i) \rangle$$

where C_i is the Global Constraint Identifier, C_{ik} is the sub constraint that is decomposed from C_i and targeted at the ontology source S_j . The and-list is a list of SPARQL queries forming a conjunction of sub constraints whose value will be used to determine whether this global constraint C_i is being violated or not. For our running example, the following CPT in table 6.4 will be generated.

Table 7.4 Constraint planning table

C_i	$\text{list}(C_{ik}, S_j)$	$\text{and-list}(C_i)$
C_1 (Semantic Constraint)	$(C_{11}, S_1)(C_{12}, S_1)(C_{13}, S_1),$ $(C_{14}, S_2), (C_{15}, S_3), (C_{16}, S_1)$	$(C_{11}, C_{12},$ $C_{13}, C_{14}, C_{15}, C_{16})$
C_2 (Specific Individual Type Constraint)	$(C_{21}, S_1)(C_{22}, S_1)(C_{23}, S_2),$ $(C_{24}, S_2), (C_{25}, S_2)$	$(C_{21}, C_{22},$ $C_{23}, C_{24}, C_{25})$

It is obvious that each sub constraint corresponds to a sub-goal in the constraint rule:

C1:

$:- K \text{ hasAmount}(x, y), K \text{ hasPatient}(x, z), K \text{ hasTrt}(x, w), K \text{ trtDisease}(w, \text{'smallpox'}), K$
 $\text{hasPlan}(z, \text{'A'}, y > 5000).$

Algorithm: Constraint Planning Algorithm

For each constraint c in the list of global constraints \mathbf{C}

For each site s from CST that is affected by constraint c in the update

If site s is not where the update happens

Then generate sub-constraints in the form of SPARQL queries from all the predicates (available in OOL) that reference to site s using appropriate conditions. Include arithmetic queries when necessary.

Elseif site s is where the update happens

Then **If** there are variables whose values are from s

Then generate sub-constraints similar to the above case

If there are variables whose values are from remote sites

Then generate queries to retrieve values from those remote sites first, then use those values to generate sub-constraints using similar method above

End for

End for

C2:

$\text{: - } \mathbf{K} \text{ hasPatient}(x, y), \mathbf{K} \text{ hasPatient}(x, z), \mathbf{K} \text{ Patient}(y), \mathbf{K} \text{ pediatricPatient}(z), y=z.$

$C_{11} = \text{SELECT } ?x, ?y$

WHERE

{ ?x rdf:hasAmount ?y }

```
C12 = SELECT ?x, ?z
      WHERE
      { ?x rdf:hasPatient ?z }
```

```
C13 = SELECT ?x, ?w
      WHERE
      { ?x rdf:hasPatient ?w }
```

With the values from Ontology Object List, the above SPARQL queries run locally, as C₁₁, C₁₂, C₁₃ are targeted to S₁ (Claim Ontology) from Constraint Planner Table. The values of those variables (identifiers that begin with a ?) will be available for parallel execution of the rest sub constraints. Under current state, $?x = claim1$; $?y = 7000$; $?w = treatment1$; $?z = Tom$ ($?z = Tommy$ can also be obtained if we combine two insertions, but to zoom in the fine-grained steps involved here, we consider them as separate updates in step 5).

```
C14 = ASK
      { treatment1 rdf:trtDisease smallpox. }
```

```
C15 = ASK
      { Tom rdf:hasPlan A. }
```

```
C16 = ASK
      { 7000 rdf:greatThan 5000. }
```

Since all sub goals are satisfied, an empty head will be obtained from constraint rule C₁, indicating a violation. The idea is to run C₁₄, C₁₅ and C₁₆ in parallel, and also to avoid bringing the Patient Ontology and Treatment Ontology to server 1 and to achieve higher efficiency.

For global constraint C₂, we will obtain:

```
C21 = SELECT ?x, ?y
```

WHERE

{ ?x rdf:hasPatient ?y }

C₂₂ = SELECT ?x, ?z

WHERE

{ ?claim rdf:hasPatient ?z }

Note that the result of these two queries will give four combinations for (?y, ?z) pair: *(Tom, Tom)*, *(Tommy, Tommy)*, *(Tommy, Tom)*, *(Tom, Tommy)*. All four combinations will be checked later on, but the first three will fail at the sub goal

patient(y), pediatricPatient(z)

in constraint rule C₂ (as a result of failing the following ASK queries). Thus empty head cannot be obtained from the first three combinations. For simplicity, we only consider the last combination for C₂₃, C₂₄, C₂₅.

C₂₃ = ASK

{ *Tom* rdf:type *patient*. }

C₂₄ = ASK

{ *Tommy* rdf:type *pediatricPatient*. }

C₂₅ = ASK

{ *Tom* rdf:sameIndividualAs *Tommy*. }

As we assumed earlier that *Tom* and *Tommy* reference to the same entity, all sub goals are met. Even though only one of four combinations gives us an empty head, one empty head is enough to indicate an inconsistency.

Step 6 (Constraint Optimizer)

This step is necessary when there are two or more global constraints are potentially violated by a single update. The Optimizer will reorganize the order of the constraint checking for global constraints in order to achieve higher efficiency. For example, instead of having the Constraint Planning Table in table 6.5, we rearrange it into the following CPT table and let C_2 to be checked first:

Table 7.5 Optimized Constraint Planning table

C_i	list(C_{ik}, S_j)	and-list(C_i)
C_2 (Specific Individual Type Constraint)	(C_{21}, S_1)(C_{22}, S_1) (C_{23}, S_2), (C_{24}, S_2), (C_{25}, S_2)	(C_{21} , C_{22} , C_{23} , C_{24} , C_{25})
C_1 (Semantic Constraint)	(C_{11}, S_1)(C_{12}, S_1) (C_{13}, S_1), (C_{14}, S_2), (C_{15}, S_3), (C_{16}, S_1)	(C_{11} , C_{12} , C_{13} , C_{14} , C_{15} , C_{16})

In this case, if either one of the global constraints is violated, the update will be rejected. Therefore, it takes less time if the Constraint Optimizer reorganizes the sequence of global constraint checking and let the Constraint Executor from Step 7 test Specific Individual Type Constraint first. This is because it only involves accessing two ontologies S_1 and S_2 , while the other constraint involves accessing three ontologies S_1 , S_2 and S_3 . If C_2 turns out to be invalid, there is no need to test for C_1 anymore. In addition, the Constraint Optimizer can further reduce the time by employing short-circuit evaluation. Since the final result can be treated as a Boolean value obtained from a conjunction of set of sub constraints, any sub constraints that are evaluated to be false will not result in an empty head, meaning no violation will happen. Furthermore, we can evaluate the local (S_1) sub constraint first in order to avoid unnecessary access to remote sites.

Step 7 (Constraint Executor)

In this step, the Constraint Executor will take input as the optimizer CPT and send sub constraint checkers to remote site to obtain information that is necessary for determining the final result. In case of C_1 , C_{14} and C_{15} will be sent to S_3 and S_2 respectively, in parallel, while C_{16} is running locally at the same time. Next, the constraint checker at S_1 will use these values to evaluate the value of a set of conjunctions of the constraint rule. Since the all sub goals are met, the insertion of *hasAmount(claim1, 7000)* will cause a violation to the global constraint C_1 .

Step 8

The final result is send back to the user with an empty head meaning violation(s) or otherwise meaning no violations.

Now we conclude our approach to maintaining integrity constraints in the third process of OWL lifecycle. The proposed general framework employs mobile agents for concurrent validations for global constraints that involve multiple scattered ontologies. Our architecture is completely distributed. It focuses on the details of the constraint checker, which runs on each ontology. Our main contribution is the discovery of global constraint violation issue, and the design of a mechanism that is faster and of less network traffic. To the best of our knowledge, this is the first time to identify the constraint-maintaining problem among multiple distributed ontologies. The applicability of this proposed framework is not limited to the ontologies we used in our illustration, as we did not make any assumptions about the structure of ontologies. Our method can be easily extended to ontologies in other areas like wireless sensor networks and business models. In addition, the global constraint (Specific Individual Type constraint) that we have demonstrated can be replaced by other types of constraints examined in chapter 4.

CHAPTER 8 CONCLUSION AND FUTURE WORK

In this dissertation, we have presented our approaches to maintaining integrity constraints in the Semantic Web. Our design works closely around the entire lifecycle of an OWL document since its creation. We have identified three processes in which constraints can be potentially violated, namely the processes of OWL generation, maintenance, and interactions with other ontologies. Our work employed a divide-and-conquer approach to tackle these three sub-problems individually. During the process of OWL generation from relational database, the paraconsistent model is used to maintain integrity constraints during the relational database to OWL translation process. To preserve integrity constraints in a single OWL during its update, a new rule-based language with set extension is introduced as a platform to allow users to specify constraints, along with a demonstration of 18 commonly used constraints written in this language. In addition, we have designed a hybrid architecture that combines Jena framework and Drools rule system to handle inconsistencies in biomedical ontologies. Afterwards, an improved constraint maintenance system, called Jena2Drools 2.0, was implemented and evaluated to show its effectiveness and efficiency to handle general-purpose constraint checking. To further handle inconsistencies in interactions between multiple distributed ontologies, we constructed a framework to break down global constraints into several sub-constraints for efficient parallel validation. These methods are brought together as a coherent integral by their underlying uniformed constraint specification language: Rule-based DL language with set extension. The second and third process utilize this formalism at the logic level, to build the IC capability on top of OWL DL, while the paraconsistent model serves as the constraint enabling mechanism at the implementation level.

Current and future work will focus on the construction of a single system that encompasses all three processes. This system will be able to handle integrity constraint issues throughout the lifecycle of OWL files. The efforts presented in this dissertation are unified under the same rule-based framework for constraint checking, which essentially lay out the theoretical foundation for this single system. The solutions introduced in each chapter were yet tested alone and were meant for only a single process in OWL lifecycle. Thus a single constraint checking system that unifies all these efforts is desirable in the future.

REFERENCES

- [BHL01] T. Berners-Lee, J. Hendler, O. Lassila: The Semantic Web, in *Scientific American* 284(5) 34-43, 2001.
- [Biz03] C. Bizer: D2R MAP – A Database to RDF Mapping Language, in *WWW (Posters)*, 2003.
- [BM06] D. Brickley, L. Miller: FOAF Vocabulary Specification, <http://xmlns.com/foaf/0.1/>, 2006.
- [BN02] F. Baader, W. Nutt: Basic Description Logics, in the *Description Logic Handbook*, edited by F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider, Cambridge University Press, pages 47-100, 2002.
- [BS95] R. Bagai and R. Sunderraman. A Paraconsistent Relational Data Model, in *International Journal of Computer Mathematics*, 1995.
- [Car07] J. Cardoso: The semantic web vision: Where are we?, in *Intelligent Systems*, 22(5):84–88, 2007.
- [CGL+07] D. Calvanese, G. Giacomo, D. Lembo, M. Lenzerini, R. Rosati: EQL-Lite: Effective First-Order Query Processing in Description Logics. in *IJCAI*, 274-279, 2007.
- [Dan04] B. Dan: RDF Vocabulary Description Language 1.0: RDF Schema, <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>, 2004.
- [DFJ+04] L. Ding, T. Finin, A. Joshi, R. Pan, R. Cost, Y. Peng, P. Reddivari, V. Doshi, J. Sachs: Swoogle: a search and metadata engine for the semantic web, in *Proceedings of the thirteenth ACM international conference on Information and knowledge management (CIKM '04)*, 652-659, 2004.

- [DPD+05] L. Ding, K. Pranam, Z. Ding, S. Avancha, T. Finin, A. Josh :Using Ontologies in the Semantic Web: A Survey, *Ontologies in the Context of Information Systems*, October 2005.
- [EIL+04] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, H. Tompits: Combining Answer Set Programming with Description Logics for the Semantic Web, in *Proceedings of the 9th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, 141–151, 2004.
- [FHH+01] D. Fensel, F. Harmelen, I. Horrocks, D. McGuinness, P. Schneider: OIL: An ontology infrastructure for the semantic web, in *IEEE Intelligent Systems*, 16(2):38–45, 2001.
- [GJ2004] K. Graham, C. Jeremy: Resource Description Framework (RDF): Concepts and Abstract Syntax, <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>, 2004.
- [Gru93] T. Gruber, A translation approach to portable ontology specifications, *Knowledge Acquisition*, 199–220, 1993.
- [HS02] I. Horrocks, U. Sattler: Description Logics Basics, Applications, and More, in *Tutorial at ECAI*, 2002.
- [HSH02] I. Horrocks, P. Schneider , F. Harmelen: Reviewing the Design of DAML+OIL: An Ontology Language for the Semantic Web, in *Proceedings Eighteenth National Conference on Artificial intelligence*, 792–797, 2002.
- [Joh91] S. John: *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, San Mateo: Kaufmann, 1991.
- [Kun11] M. Kunder: The size of the world wide web, Worldwidewebsite.com, October 2011.
- [MG04] D. Mike, S. Guus: OWL Web Ontology Language Reference, <http://www.w3.org/TR/2004/REC-owl-ref-20040210/>, 2004.
- [MHR06] B. Motik, I. Horrocks, R. Rosati, U. Sattler : Can OWL and logic programming live together happily ever after? in *Proc. ISWC-2006, LNCS vol. 4273, Springer*, 501 - 514, 2006.

- [MHS07] I. Horrocks, B. Motik, U. Sattler: Bridging the gap between owl and relational databases. in Proc. of the Sixteenth International World Wide Web Conference, Pages: 807 - 816, 2007.
- [Mot07] B. Motik: A Faithful Integration of Description Logics with Logic Programming, in IJCAI, 477-482, 2007.
- [Ng05] G. Ng: Open vs closed world, rules vs queries: use cases from industry, in Proceedings of OWL: experiences and directions, 2005.
- [RN03] S. Russell, P. Norvig: Inference in First-Order Logic, Artificially intelligence: a modern approach (3rd edition) 322-357, Edited by Hirsch M. New Jersey: Prentice Hall; 2003.
- [SSV02] N. Stojanovic, L. Stojanovic, R. Volz: A reverse engineering approach for migrating data-intensive web sites to the Semantic Web, in IFIP 17th World Computer Congress, pages 141–154, 2002.
- [Tao10] J. Tao: Adding Integrity Constraints to the Semantic Web for Instance Data Evaluation, in Proceedings of the 9th International Semantic Web Conference, 2010.
- [Tao12] J. Tao: Integrity Constraints For The Semantic Web: An Owl 2 DL Extension, Ph.D. Dissertation,.
- [TDB+08] J. Tao, L. Ding, J. Bao, D. McGuinness: Characterizing and Detecting Integrity Issues in OWL Instance Data, in Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008 EU), 2008.
- [TZ86] S. Tsur and C. Zaniolo: LDL: a logic-based data-language, in Proceedings of 12th VLDB Conference, 33-41, 1986.

[WRJ+09] W. Tai, R. Brennan, J. Keeney, D. OSullivan: An Automatically Composable OWL Reasoner for Resource Constrained Devices, in Proceedings of 3rd IEEE International Conference on Semantic Computing, pp. 495-502, September, 2009.