DEBUGGING AND REPAIR OF DESCRIPTION LOGIC ONTOLOGIES

by
Kodylan Moodley

Submitted in fulfilment of the academic
requirements for the degree of
Master of Science in the
School of Computer Science,
University of KwaZulu-Natal
Durban

December 2010

As the candidate's supervisor I have/have not approved this thesis/dissertation for submission.

Signed:_____ Name:_____ Date:_____

Preface

The experimental work described in this dissertation was carried out in the School of Computer Science, University of KwaZulu-Natal, Durban, from January 2009 to December 2010, under the supervision of Professor Thomas Meyer, Dr Deshendran Moodley and Dr Ivan Varzinczak.

These studies represent original work by the author and have not otherwise been submitted in any form for any degree or diploma to any tertiary institution. Where use has been made of the work of others it is duly acknowledged in the text.

Declaration 1 - Plagiarism

I, _____, declare that:

1. The research reported in this dissertation, except where otherwise indicated, is my original research.
2. This dissertation has not been submitted for any degree or examination at any other university.
3. This dissertation does not contain other person's data, pictures, graphs or other information, unless specifically acknowledged as being sourced from other persons.
4. This dissertation does not contain other persons' writing, unless specifically acknowledged as being sourced from other researchers. Where other written sources have been quoted, then:
   (a) Their words have been re-written but the general information attributed to them has been referenced
   (b) Where their exact words have been used, then their writing has been placed in italics and inside quotation marks, and referenced.
5. This dissertation does not contain text, graphics or tables copied and pasted from the Internet, unless specifically acknowledged, and the source being detailed in the dissertation and in the References section.

Signed: _____

Declaration 2 - Publications

Publication 1

Meyer, T. Moodley, K. and Varzinczak, I. (2010). First Steps in the Computation of Root Justifications. In *Proceedings of the ECAI-10 Workshop on Automated Reasoning about Context and Ontology Evolution (ARCOE)*. `http://www.arcoe.org`

Contributions of Authors:

T. Meyer contributed to the theoretical aspects and writing of the document. K. Moodley contributed to the theoretical aspects, implementation and experimental work and writing of the document. I. Varzinczak contributed to the theoretical aspects and writing of the document.

Signed: _____

Acknowledgements

Firstly, I would like to thank my supervisor Prof. Thomas Meyer and co-supervisors Dr. Deshendran Moodley and Dr. Ivan Varzinczak for their support and invaluable advice during this work. I would also like to thank the members of the Knowledge Representation and Reasoning (KRR) group at CSIR Meraka for their help and encouragement.

I am very grateful to Matthew Horridge of the University of Manchester who piqued my interest in Ontology Debugging and Repair with his presentation on "Laconic and Precise Justifications" and provided important advice with the practical component of my research.

Many thanks to the Council for Scientific and Industrial Research (CSIR) for awarding me the funding to conduct the research and last, but not least, a special appreciation goes to my family for providing me with their love and support throughout my studies.

List of Figures

2.1 Concept hierarchy for Pizza Ontology 16

3.1 Preservation of entailment and justifications in a module 30
3.2 Example HS-tree 34
3.3 Example Justification Tree 37
3.4 Justification Tree for Example 3.5 43
3.5 Error-dependency graph for Example 3.7 46
3.6 Flow diagram for root and derived unsatisfiable concept approach 47
3.7 Error-dependency graphs: updating dependencies 48

4.1 All justifications for the set of unwanted axioms γ_1, γ_2 and γ_3 57
4.2 Root Justification Tree for the set of unwanted axioms $\{\gamma_1, \gamma_2, \gamma_3\}$ 58

5.1 Protégé 4 ontology editor. 62
5.2 Functioning of the main aspects of OWL API 64
5.3 Flow diagram for *OntoRepair* 65
5.4 *OntoRepair* interface. 66
5.5 *OntoRepair* interface, computing root justifications. 67
5.6 *OntoRepair* interface, identifying repairs. 67

A.1 Tutorial: Download Protégé 75
A.2 Tutorial: Install Protégé 75
A.3 Tutorial: Download *OntoRepair* 76
A.4 Tutorial: Protégé welcome screen 77
A.5 Tutorial: Protégé Tabs Menu 77
A.6 Tutorial: *OntoRepair* Interface 78
A.7 Tutorial: *OntoRepair* Regular Justifications 79
A.8 Tutorial: *OntoRepair* Root Justifications 79
A.9 Tutorial: *OntoRepair* Repairs 80
A.10 Tutorial: *OntoRepair* Apply Repair 81

List of Tables

2.1 Summary of $\mathcal{ALC}$ syntax and semantics. 10
2.2 A list of common DL feature symbols 12

3.1 Justifications for Example 3.7 46

5.1 Correlation between OWL and DL syntax. 63
5.2 Results for Test Case 1. 69
5.3 Results for Test Case 2. 69
5.4 Results for Test Case 3. 69

Abstract

In logic-based Knowledge Representation and Reasoning (KRR), ontologies are used to represent knowledge about a particular domain of interest in a precise way. The building blocks of ontologies include concepts, relations and objects. Those can be combined to form logical sentences which explicitly describe the domain. With this explicit knowledge one can perform reasoning to derive knowledge that is *implicit* in the ontology. Description Logics (DLs) are a group of knowledge representation languages with such capabilities that are suitable to represent ontologies. The process of building ontologies has been greatly simplified with the advent of graphical ontology editors such as SWOOP, Protégé and OntoStudio. The result of this is that there are a growing number of ontology engineers attempting to build and develop ontologies. It is frequently the case that errors are introduced while constructing the ontology resulting in undesirable pieces of implicit knowledge that follows from the ontology. As such there is a need to extend current ontology editors with tool support to aid these ontology engineers in correctly designing and debugging their ontologies. Errors such as *unsatisfiable concepts* and *inconsistent ontologies* frequently occur during ontology construction. Ontology Debugging and Repair is concerned with helping the ontology developer to eliminate these errors from the ontology. Much emphasis, in current tools, has been placed on giving *explanations* as to *why* these errors occur in the ontology. Less emphasis has been placed on using this information to suggest efficient ways to *eliminate* the errors. Furthermore, these tools focus mainly on the errors of unsatisfiable concepts and inconsistent ontologies. In this dissertation we fill an important gap in the area by contributing an alternative approach to ontology debugging and repair for the more general error of a list of *unwanted sentences*. Errors such as unsatisfiable concepts and inconsistent ontologies can be represented as unwanted sentences in the ontology. Our approach not only considers the *explanation* of the unwanted sentences but also the identification of repair strategies to *eliminate* these unwanted sentences from the ontology.

Contents

Preface ii

Declaration 1 iii

Declaration 2 iv

Acknowledgements v

List of Figures vi

List of Tables vii

Abstract viii

1 Introduction 2
- 1.1 Ontologies . . . 2
- 1.2 Description Logics . . . 2
- 1.3 Semantic Web & OWL . . . 3
- 1.4 Ontology Debugging and Repair . . . 4
- 1.5 Organization of the Dissertation . . . 6

2 Description Logics 7
- 2.1 Background . . . 7
- 2.2 The Description Logic $\mathcal{ALC}$ . . . 8
  - 2.2.1 Syntax and Semantics . . . 8
  - 2.2.2 DL Expressivity and Reasoning Complexity . . . 11
- 2.3 Standard Reasoning Services . . . 14
  - 2.3.1 Satisfiability . . . 14
  - 2.3.2 Subsumption . . . 15
  - 2.3.3 Classification . . . 16
  - 2.3.4 Instance Checking . . . 16
  - 2.3.5 Consistency Checking . . . 17
- 2.4 Non-Standard Reasoning Services . . . 18
  - 2.4.1 Explanation . . . 18
  - 2.4.2 Repair . . . 19

3 Ontology Debugging and Repair 20
- 3.1 DL Ontology Errors . . . 20
- 3.2 Explanations for Errors . . . 21
- 3.3 Computing Justifications: Black-box Approach . . . 23
  - 3.3.1 Computing a Single Justification . . . 24

3.3.2 Computing All Justifications . . . 32
3.4 Computing Justifications: Glass-box Approach . . . 37
3.5 Computing Justifications: Hybrid Approach . . . 38
3.6 Fine-grained Justifications . . . 39
3.7 Error Resolution . . . 40
3.7.1 Resolving a Single Error . . . 41
3.7.2 Resolving a List of Errors . . . 45
3.8 Open Issues and Limitations . . . 47

4 Ontology Repair Using Root Justifications 50
4.1 Generalization of Errors . . . 50
4.1.1 Unsatisfiable Concepts . . . 51
4.1.2 Unintended Inferences . . . 51
4.1.3 Inconsistent Ontologies . . . 52
4.1.4 Unwanted Axioms . . . 52
4.2 Root Justifications . . . 53
4.2.1 Computing Root Justifications . . . 54
4.2.2 Repairs From Root Justifications . . . 58
4.3 Comments . . . 59

5 Implementation and Evaluation 61
5.1 Protégé . . . 61
5.1.1 Protégé User Interface . . . 62
5.1.2 The OWL API . . . 63
5.1.3 Protégé Plugins . . . 64
5.2 Implementation of OntoRepair Plugin . . . 64
5.2.1 Problem/Purpose . . . 65
5.2.2 Interface . . . 65
5.3 Evaluation of OntoRepair Plugin . . . 68
5.3.1 Test cases . . . 68
5.3.2 Results . . . 68
5.3.3 Analysis and Conclusion . . . 69

6 Conclusion 71
6.1 Summary of Contribution . . . 71
6.2 Open Issues and Future Work . . . 72

A Appendix 74
A.1 Mini Tutorial: OntoRepair . . . 74
A.1.1 Installing Protégé 4 . . . 74
A.1.2 Installing OntoRepair . . . 75
A.1.3 Getting Started with OntoRepair . . . 76

Bibliography 87

Chapter 1

Introduction

1.1 Ontologies

An *ontology* (also referred to as a *terminology*, *knowledge base*) is an entity used to represent some domain (field of knowledge). To be more specific, an ontology precisely depicts some representation of the domain. Usually the building blocks of an ontology include categories (concepts), relations (roles) and objects (individuals). If we were to consider the domain of a university, we could use concept names such as Lecturer, Student and Module, and role names such as teaches and enrolledFor, and combine these constructs to form sentences which describe the domain, like "A Lecturer is someone who teaches a Module" or "A Student is someone who enrolledFor a Module". An ontology can be viewed as a set of such sentences or as a taxonomy in which the concepts are classified as more general (*super-concepts*) or more specific (*sub-concepts*) in relation to one another. For example the sentence "Every Student is a Person" means that the Person concept is more general than the Student concept. There are many different formalisms for representing ontologies. These include: Semantic Networks [Woods, 1975; Sowa, 1991], Conceptual Graphs [Sowa, 1976], Unified Modelling Language (UML) [Rumbaugh et al., 1997], frame-based systems [Bobrow and Winograd, 1977; Brachman and Schmolze, 1985; Fikes and Kehler, 1985] and Description Logics (DLs) [Baader et al., 2003].

1.2 Description Logics

The purpose of an ontology, as mentioned, is to provide a precise description of the domain. In this description one can classify the individuals in our domain into concepts and we can also represent relationships that may occur between these individuals. DLs, which are a group of logic-based knowledge representation formalisms, are suitable languages for representing ontologies in this way. They have an advantage over representations such as Semantic Networks and frame-based systems in that they have precise logic-based semantics. This eliminates ambiguity in the meaning of sentences in the ontology. Conceptual Graphs and UML formats may also be used to represent ontologies but DLs provide a richer language for this purpose and are thus preferred in most cases.

The basic building blocks of DLs are *atomic concepts*, *atomic roles* and *individuals*. An atomic concept is a unary predicate which defines a collection of objects from the domain.

An atomic role is a binary predicate indicating objects from the domain which are related in a certain way. Individuals denote the actual objects in the domain. DLs also have concept and role *constructors* to define more complex concepts and roles.

DLs allow us to represent the domain using logical sentences. Collectively this set of sentences is known as a knowledge base (KB). DLs allow us to perform automated *reasoning* over the KB to gather information from it that is not explicitly shown (in the form of implicit sentences). The KB is sometimes viewed in two separate parts: the *Terminological Box* (TBox) storing so-called *intensional* information about the ontology, and the *Assertional Box* (ABox) storing so-called *extensional* information. The intensional information in the TBox defines the different concepts in the domain and, with the use of role names, it also defines how these concepts are related in the domain. The ABox stores assertions about individuals in the domain, that is, given a concept, which individuals belong in its extension.

DLs vary according to their expressive power. Less expressive DLs have restrictions on the kinds of sentences one can represent in the language, and in turn this reduces the complexity of reasoning [Levesque and Brachman, 1985]. More expressive DLs allow for the representation of more detailed information but in doing so reasoning becomes much more complex. The choice of which DL to use will differ according to the application. For example, in more performance driven applications, tractability may be a required characteristic for the DL used. A more detailed introduction to DLs is given in Chapter 2.

1.3 Semantic Web & OWL

Tim Berners-Lee, the inventor of the World Wide Web, had a two-part vision for its development. The first part was to establish a "common information space" which could be used by people for communication and information sharing purposes. The World Wide Web, as we know it today, provides an excellent framework which achieves this. However with the sheer amount of information (which is increasing every day) available on the web today, it is becoming more difficult for users to isolate information that is relevant for their purposes. The second part of Berners-Lee's vision was to be able to use computers to analyze and process the information on the Web. The reason for this was to enable us to collaborate better and understand our context. The idea was conceived that somehow we should enrich the information across the World Wide Web with meaning (semantics). This would allow machines (computers) to attach meaning to information and thus enable better communication between computers, between computers and humans and therefore between humans. Thus the Semantic Web was born.

The Semantic Web is an extension of the World Wide Web based on the Resource Description Framework (RDF) [Lassila and Swick, 1998] which encodes meaning (semantics) into web content. The motivation for this is that computers cannot inherently understand semantics [Berners-Lee et al., 2001]. Sentences like "Professor Anderson is a Lecturer", "Professor Anderson teaches Physics" and "K. Wright is enrolled for Physics" can be understood by humans but not by computers. To facilitate the processing of meaning in the Semantic Web, web pages are enriched with semantic information. The Semantic Web uses ontologies (introduced in Section 1.1) to represent this semantic information. With the use

of ontologies and the Semantic Web, computers are able to create a shared understanding of web content so that tasks such as querying of information is more powerful and returns much more meaningful and relevant results.

In keeping with Tim Berners-Lee's vision for the Semantic Web, the OWL (Web Ontology Language) family was developed as a series of languages for representing Semantic Web ontologies. OWL languages are based foundationally on description logics (introduced in Section 1.2) although they borrow syntactically from the popular RDF/XML formats and have certain language varieties which differ considerably from DLs by allowing more expressive constructs.

Initially when developing Semantic Web technologies, the focus was on RDF schema but because of its limited expressive power a more suitable language was required. The World Wide Web Consortium (W3C) then initiated development on what would become the *OWL 1* set of languages. There are three "species" of OWL 1: the less expressive OWL Lite, more so OWL DL and the most expressive OWL Full. OWL 1 was identified to have some minor shortcomings in the expressivity, syntax and semantics and it was decided that a new standard should be developed to resolve some of these issues [Cuenca-Grau et al., 2008]. This resulted in OWL 2: the latest revision of the OWL standard which is, at the time of writing, the World Wide Web Consortium (W3C) recommendation for the representation of Semantic Web ontologies. OWL 2 has two main 'flavours' namely OWL 2 DL and OWL 2 Full.

There are three OWL 2 *profile languages* tailored for specific ontological applications. These are essentially subset languages of OWL 2 Full which simplifies the reasoning for their respective applications.

OWL 2 EL is based on the description logic $\mathcal{EL}$++ [Baader, 2003]. $\mathcal{EL}$++ is an extension of the $\mathcal{EL}$ [Baader, 2003] language which is the foundation for many large-scale ontologies such as the biomedical ontology SNOMED CT[1]. SNOMED CT, which has thousands of concepts and roles, does not require the highly expressive features available in certain DLs which would make reasoning potentially very complex. This is where $\mathcal{EL}$ becomes an appropriate choice [Baader et al., 2005a] for representing the ontology. Because of its limited expressivity, reasoning is tractable with $\mathcal{EL}$ which makes the huge task of reasoning over an ontology like SNOMED that much easier.

OWL 2 QL (Query Language) is another OWL 2 Profile formulated for applications which require a lot of querying of instance data (individuals). Relational Database systems also use OWL 2 QL for ontology-based data access (OBDA) [Calvanese et al., 2007; Poggi et al., 2008]. OWL 2 QL makes it possible in OBDA to translate ontological queries into standard relational query languages such as SQL. The OWL 2 RL (Rule Language) is designed for specifying reasoning systems which implement reasoning tasks as a set of ontology rules.

1.4 Ontology Debugging and Repair

The process of building ontologies has been greatly simplified with the advent of graphical ontology editors such as SWOOP [Kalyanpur et al., 2005a], Protégé [Knublauch et al.,

[1] `http://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html`

2004] and OntoStudio[2]. The result of this is that there are a growing number of *ontology engineers* – in conjunction with experts in the domain – attempting to build and develop ontologies. It is frequently the case that *errors* are introduced while constructing the ontology resulting in implicit knowledge from the ontology which is not desired. As such there is a need to extend current ontology editors with tool support to aid these ontology engineers in correctly designing and debugging their ontologies.

Errors such as *unsatisfiable concepts* and *inconsistent ontologies* frequently occur during ontology construction. Ontology Debugging and Repair is concerned with helping the ontology developer to eliminate these errors from the ontology. Much emphasis, in current tools, has been placed on giving explanations as to *why* these errors occur in the ontology. Less emphasis has been placed on using this information to suggest efficient ways to eliminate the errors. In addition, the existing approaches tend to focus only on certain types of errors such as those of unsatisfiable concepts and ontology inconsistency.

Motivation

The motivation of this dissertation is that firstly, we would like to generalize the existing principles for debugging and repairing DL ontologies to deal with all types of unwanted consequences (errors) which arise in the ontology. Secondly we would like to develop an approach to ontology repair which aids the ontology engineer in *eliminating* all the unwanted consequences from the ontology and not just explaining *why* they follow from the ontology.

Contribution

The main contribution of this dissertation is an alternative approach to debugging and repair for DL ontologies. This approach generalizes all the different ("heterogeneous") errors which occur in DL ontologies to *one* type of error which we call an *unwanted axiom*. Therefore, the problem which we try to resolve is how to eliminate a *list* of such unwanted axioms from the ontology. Our approach generates a list of alternative repair strategies. Each of these may be applied to the ontology to eliminate the entire list of unwanted axioms from the ontology.

Scope

In this dissertation we focus only on *semantic defects* in DL ontologies. Specifically, in our ontology repair solution, we focus on eliminating a list of unwanted axioms from the ontology. Each of these unwanted axioms is derived from a semantic defect in the ontology. That is, we focus on resolving unwanted axioms that are derived from unsatisfiable concepts, inconsistent ontologies and unintended inferences and not other types of error. In addition, we only consider TBox unwanted axioms in the ontology and not any unwanted *assertions* in the ABox of the ontology. The list of repair strategies which are generated by our solution serves as a list of alternatives from which the ontology engineer and domain expert may choose. That is, we do not aid the ontology engineer and domain expert in selecting from these repairs.

[2]http://www.ontoprise.de/en/home/products/ontostudio

1.5 Organization of the Dissertation

The rest of this dissertation is organized as follows:

- Chapter 2 gives an introduction to DLs including syntax and semantics of the basic DL $\mathcal{ALC}$. This chapter also lays the foundation for the terminology which will be used throughout the dissertation.
- Chapter 3 introduces the problem of Ontology Debugging and Repair. This chapter examines the different types of errors that occur in the ontology during development and also looks at some of the established approaches to identifying, explaining and repairing these errors.
- Chapter 4 constitutes the main contribution of the dissertation. In this chapter we generalize the problem of repairing different types of semantic defects in the ontology to the problem of eliminating a list of unwanted sentences (axioms) from the ontology. We also present an alternative approach for resolving this problem in the ontology.
- Chapter 5 presents the other contribution of this dissertation: the implementation and evaluation of a Protégé plugin (*OntoRepair*) which demonstrates the approach to ontology debugging and repair that is presented in Chapter 4. The plugin collects a list of user-specified unwanted axioms in the ontology and computes a list of repair strategies. Each of these repair strategies are such that if it is applied to the ontology, then the entire list of unwanted axioms will be eliminated from the ontology.
- Chapter 6 gives a summary of the contribution and findings of the dissertation and concludes with some open issues and future work.
- Finally, the Appendix gives a short tutorial on how to get started with using Protégé and the *OntoRepair* plugin.

Chapter 2

Description Logics

This chapter gives an introduction to Description Logics (DLs) and some of their applications. Firstly, we give some background and history to DLs. Thereafter, we give an overview of some their constructs by introducing the basic description logic $\mathcal{ALC}$ together with its formal syntax and semantics. DL characteristics such as expressivity and reasoning complexity are also briefly discussed by mentioning some popular extensions and fragments of $\mathcal{ALC}$. The chapter concludes with a summary of the common reasoning tasks or inference problems that are relevant for DLs.

2.1 Background

Description Logics [Baader et al., 2003] are a set of logic-based knowledge representation languages that are most commonly used to represent information about some domain of interest in a formal and structured manner. DLs are designed to be decidable subsets of First-Order Logic (FOL) [Kleene, 1968] and, as a consequence, they have a precise model-theoretic semantics. This formal semantics eliminates ambiguity in the meaning of terms and sentences represented using DLs.

The basic building blocks of DLs are *atomic concepts*, *atomic roles* and *individuals*. An atomic concept (also called a *concept name*) is a unary predicate which intuitively represents a collection of objects from the domain. For example, a concept name Student could represent the set of all students in a particular university. An atomic role is a binary predicate indicating ordered object pairs which are related via the role. For example, the atomic role marriedTo could represent the set of all married couples in a particular domain. Therefore the sentence *john* marriedTo *susan* formally indicates that the objects in the domain which are referred to respectively by the names *john* and *susan* are related to each other via the role which is represented by the name marriedTo. Individuals (also called *instances*) refer to the actual objects in the domain. The objects which the names *john* and *susan* refer to are examples of individuals. In addition to the basic building blocks of DLs, there are also concept and role *constructors* which are used to define *complex* concepts and roles. We shall discuss these in the sequel.

An important application of DLs is in the formal specification of ontologies. DLs have a precise model-theoretic semantics and a capacity to allow for reasoning about the ontology. The former means that each DL sentence in the ontology specification has a precise meaning

and there is no confusion or ambiguity about what the sentence represents. This is the main advantage of DLs over other formalisms for representing ontologies such as Semantic Networks [Woods, 1975; Sowa, 1991] and Frame-based systems [Bobrow and Winograd, 1977; Brachman and Schmolze, 1985; Fikes and Kehler, 1985]. DLs allow one to represent the ontology in terms of logical sentences. These logical sentences are known as *axioms* or more specifically, *DL axioms*.

Reasoning is the task of deriving implicit knowledge from the explicitly stated facts in the ontology. Automated reasoning can be performed over DL ontologies by software inference engines called *DL reasoners*. The reasoners identify implicit consequences in the ontology through application of inference rules. For example if "*tweety is a bird*" and "*birds fly*" then one can infer that "*tweety flies*". DL reasoning is discussed later in this chapter. Another important aspect of DLs is their *expressivity*. The constructors included in a particular DL determines the expressivity of that DL. Furthermore, the expressivity of a DL has an effect on the complexity of reasoning for this DL. The issues of DL expressivity and DL complexity are discussed briefly in Section 2.2.2.

The rest of this chapter is structured as follows. Section 2.2 introduces the syntax and semantics of the basic Description Logic $\mathcal{ALC}$ [Schmidt-Schauß and Smolka, 1991] and discusses some popular variants of the $\mathcal{ALC}$ language. Issues concerning the naming convention of DLs, DL expressivity and reasoning complexity are also briefly discussed in this section. Finally, Sections 2.3 and 2.4 conclude the chapter by introducing some of the popular reasoning services provided by DLs.

2.2 The Description Logic $\mathcal{ALC}$

$\mathcal{ALC}$ (Attributive concept description Language with Complements) [Schmidt-Schauß and Smolka, 1991] is an important basic DL which may be extended or restricted to form others. $\mathcal{ALC}$ includes all the basic constructors one would need for most common applications and thus is often used as an example in introductory literature for DLs. The following section presents the syntax and semantics of the DL $\mathcal{ALC}$. Please note that the definitions and theorems presented in the remainder of this chapter are adapted from the DL Handbook [Baader et al., 2003].

2.2.1 Syntax and Semantics

$\mathcal{ALC}$ includes the following DL constructors: the $\top$ (Top) and $\bot$ (Bottom) special concepts, $\sqcap$ (concept conjunction), $\sqcup$ (concept disjunction), and $\neg$ (concept negation). In addition, the complex concept constructors $\exists$ (existential restriction) and $\forall$ (value restriction) are also available. The syntax and semantics for these are given below.

$\mathcal{ALC}$ syntax is defined as follows. $\mathcal{ALC}$ concept syntax is defined first, followed by the syntax for $\mathcal{ALC}$ sentences (axioms). Let N_C be a set of concept names. Examples of concept names are Bird and Male. Let N_R be a set of role names. Examples of role names include hasChild and marriedTo. The set of $\mathcal{ALC}$ *concept descriptions* is the smallest set such that:

- $\top$, $\bot$, and every concept name $A \in N_C$ is an $\mathcal{ALC}$ concept,
- If C and D are $\mathcal{ALC}$ concepts and $R \in N_R$, then $C \sqcup D$, $C \sqcap D$, $\neg C$, $\exists R.C$, and $\forall R.C$ are $\mathcal{ALC}$ concepts.

Some examples of $\mathcal{ALC}$ complex concepts are: $\neg(\mathsf{Bird} \sqcap \mathsf{FlyingAnimal})$ (representing the set of entities in the domain which are neither birds nor flying animals) , $\exists\mathsf{hasChild}.\top$ (representing the set of entities in the domain which have at least one child) and $\forall\mathsf{hasChild}.\mathsf{Male}$ (representing the set of entities in the domain which have only male children). Any concept following the role name in a concept description is called a *filler* concept for the role name. For example, the filler concept for the role $\mathsf{hasChild}$ in the concept description $\forall\mathsf{hasChild}.\mathsf{Male}$ is Male. In the particular case when the filler concept is $\top$, the concept description can be abbreviated to omit $\top$ because the description has the same meaning without $\top$. Therefore, $\exists\mathsf{hasChild}$ has the same meaning as $\exists\mathsf{hasChild}.\top$. In DLs, sentences are usually classified as *axioms* and *assertions*. The syntax of $\mathcal{ALC}$ sentences is now defined.

Given two $\mathcal{ALC}$ concepts C and D, a role name R and individual names a and b:

- $C \sqsubseteq D$ and $C \equiv D$ are axioms, where $\sqsubseteq$ is the *subsumption* symbol and $\equiv$ is the *equivalence* symbol.
- $C(a)$ and $R(a, b)$ are $\mathcal{ALC}$ assertions.

Some examples of $\mathcal{ALC}$ axioms are: $\mathsf{Penguin} \sqsubseteq \mathsf{Bird}$, $\mathsf{Parent} \sqsubseteq \exists\mathsf{hasChild}.\top$ and $\mathsf{Man} \equiv \mathsf{Person} \sqcap \mathsf{Male}$. Examples of $\mathcal{ALC}$ assertions are: $\mathsf{Student}(peter)$ and $\mathsf{marriedTo}(john, susan)$.

The meaning (semantics) for $\mathcal{ALC}$ concepts and $\mathcal{ALC}$ sentences is now given. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation with the non-empty set $\Delta^{\mathcal{I}}$ denoting the *domain* of $\mathcal{I}$ (A set of objects or individuals), and function $\cdot^{\mathcal{I}}$ which maps every $\mathcal{ALC}$ concept name to a subset of $\Delta^{\mathcal{I}}$, every role name to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and individual names a and b to elements $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and $b^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ respectively. The interpretation function $\cdot^{\mathcal{I}}$ is extended to complex concepts in the following way. For every $\mathcal{ALC}$ concept C and D and every role name R:

Definition 2.1 ($\mathcal{ALC}$ complex concept semantics)
Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation. Then:

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$, $\bot^{\mathcal{I}} = \emptyset$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
- $(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \text{There is a } b \in \Delta^{\mathcal{I}} \text{ s.t. } (a, b) \in R^{\mathcal{I}} \text{ and } b \in C^{\mathcal{I}}\}$
- $(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \text{For all } b \in \Delta^{\mathcal{I}}, (a, b) \in R^{\mathcal{I}} \text{ implies } b \in C^{\mathcal{I}}\}$

For example, the complex concept $\exists\mathsf{hasChild}.\top$ represents the set of individuals in the domain such that each of these individuals is related to at least one individual in the domain via the role represented by $\mathsf{hasChild}$. The concept description $\exists\mathsf{hasChild}.\top$ can therefore be represented in natural language as "the set of all entities which have at least one child".

As another example, we have the complex concept $\forall$hasChild.Male, which represents all the individuals such that, if they have one or more children, then these children are *only* males. Note that, an individual which is not related via the role represented by hasChild, to any individual in the domain, also appears (by vacuity) in the concept referred to by $\forall$hasChild.Male. This is confirmed in the semantics of $\forall R.C$ given above. The meaning for $\mathcal{ALC}$ axioms is now given.

Definition 2.2 (Satisfaction of $\mathcal{ALC}$ sentences in an interpretation)
Given $\mathcal{ALC}$ concepts C and D, a role name R, and individual names a, b. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation. Then:

- $\mathcal{I}$ *satisfies* $C \sqsubseteq D$ if and only if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
- $\mathcal{I}$ *satisfies* $C \equiv D$ if and only if $C^{\mathcal{I}} = D^{\mathcal{I}}$
- $\mathcal{I}$ *satisfies* $C(a)$ if and only if $a^{\mathcal{I}} \in C^{\mathcal{I}}$
- $\mathcal{I}$ *satisfies* $R(a, b)$ if and only if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$

For example, the axiom Parent $\sqsubseteq \exists$hasChild.$\top$ (*subsumption axiom*) means that "if some individual in our domain is a Parent, then this individual has at least one child". The axiom Man $\equiv$ Person $\sqcap$ Male (*equivalence axiom*) means that "an individual of the domain is a Man if and only if it is a Person *and* a male".

For the assertions, $C(a)$ is a *concept assertion* which means that "the individual referred to by a belongs to the set referred to by C". For example, if *peter* is an individual name and Student is a concept name, then the concept assertion Student(*peter*) means that Peter is a student ("Peter is an instance of Student"). $R(a, b)$ is called a *role assertion* which means that the individual referred to by a is related to the individual referred to by b via the role represented by R. For example, if marriedTo refers to a role and *john* and *susan* refer to individuals John and Susan, the role assertion marriedTo(*john*, *susan*) means that John is married to Susan. Table 2.1 below gives a summary of $\mathcal{ALC}$ syntax and semantics.

Name	**Syntax**	**Semantics**
Top concept	$\top$	$\Delta^{\mathcal{I}}$ (all individuals)
Bottom concept	$\bot$	$\emptyset$ (no individuals)
Conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Concept negation	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
Full existential restriction	$\exists R.C$	$\{a \in \Delta^{\mathcal{I}} \mid$ There is a $b \in \Delta^{\mathcal{I}}$ s.t. $(a, b) \in R^{\mathcal{I}}$ and $b \in C^{\mathcal{I}}\}$
Value restriction	$\forall R.C$	$\{a \in \Delta^{\mathcal{I}} \mid$ For all $b \in \Delta^{\mathcal{I}}$, $(a, b) \in R^{\mathcal{I}}$ implies $b \in C^{\mathcal{I}}\}$
Subsumption	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
Equivalence	$C \equiv D$	$C^{\mathcal{I}} = D^{\mathcal{I}}$
Concept instance	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
Role instance	$R(a, b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$

Table 2.1: Summary of $\mathcal{ALC}$ syntax and semantics.

The sentences which constitute a DL ontology are usually separated into the set of sentences which represent information about the relationships between concepts (called axioms) and the set of sentences which represent assertions about individuals in the domain (called

assertions). This separation is essentially made because there are good reasons to make a distinction between axioms and assertions and there is a difference in terms of the purposes that they serve in the ontology representation.

Definition 2.3 (Ontology, TBox and ABox) A TBox $\mathcal{T}$ is a finite and possibly empty set of axioms. An ABox $\mathcal{A}$ is a finite and possibly empty set of assertions. If $\mathcal{T}$ is a TBox and $\mathcal{A}$ is an ABox, then $\mathcal{O} = \mathcal{T} \cup \mathcal{A}$ is an ontology.

In theory, an ontology need not contain both a TBox and ABox. It is down to the specific application which the ontology is used for which determines if it will contain both a TBox and an ABox, purely a TBox or purely an ABox.

If an interpretation $\mathcal{I}$ satisfies a sentence, then it is a *model* for that sentence. Furthermore, $\mathcal{I}$ is a model for a TBox $\mathcal{T}$ (resp. ABox $\mathcal{A}$) if and only if it is a model for each sentence in $\mathcal{T}$ (resp. $\mathcal{A}$). $\mathcal{I}$ is a model for an ontology $\mathcal{O}$ (which is the union of the TBox and ABox) if and only if it is a model for the TBox and ABox of $\mathcal{O}$. The notion of *entailment* is now defined.

Definition 2.4 (Entailment)
Given a TBox $\mathcal{T}$ (resp. ABox $\mathcal{A}$, resp. ontology $\mathcal{O}$) and a sentence α, $\mathcal{T}$ (resp. $\mathcal{A}$, resp. $\mathcal{O}$) *entails* α, written $\mathcal{T} \models \alpha$ (resp. $\mathcal{A} \models \alpha$, resp. $\mathcal{O} \models \alpha$) if and only if every model of $\mathcal{T}$ (resp. $\mathcal{A}$, resp. $\mathcal{O}$) satisfies α.

TBox axioms of the form $C \sqsubseteq D$ are known as *subsumption* or *inclusion* axioms. When C and D are *concept descriptions* (i.e., complex concepts built from concept names), then $C \sqsubseteq D$ is a *general concept inclusion* (GCI) axiom. Axioms of the form $C \equiv D$ are called *equivalence* axioms. If the left hand side (LHS) of a subsumption axiom or equivalence axiom is a *concept name* then the axiom is a *concept definition axiom* (CDA) for this concept name. A special case of equivalence and inclusion axioms, known as *disjointness* axioms, are of the form $C \sqcap D \sqsubseteq \bot$ or $C \sqcap D \equiv \bot$. In both these cases we say that "C is *disjoint* with/from D" or "C and D are *disjoint*", which semantically reflects the fact that C and D have no elements in common. From hereon, thoughout this dissertation, we refer to general sentences (axioms or assertions) in the ontology as axioms.

Some applications of DLs require more expressivity (more constructors) than $\mathcal{ALC}$ offers. On the other hand, for other applications, some $\mathcal{ALC}$ constructors are not required. The next section discusses issues related to expressivity and complexity of reasoning with DLs by introducing some popular variants of the $\mathcal{ALC}$ DL.

2.2.2 DL Expressivity and Reasoning Complexity

$\mathcal{ALC}$ includes some basic features such as full concept negation, concept conjunction, concept disjunction, existential quantification and value restriction quantification. Despite the inclusion of all these constructors in the $\mathcal{ALC}$ DL, it is generally not considered as an overly expressive DL. For example, it does not include features such as *role hierarchies* (to express that a role is included in another one e.g. $\mathsf{hasParent} \sqsubseteq \mathsf{hasAncestor}$ [Horrocks et al., 2000; Horrocks and Sattler, 1999] and *cardinality restrictions* (to express the concept of being in at most/at least n relationships via a role) [Baader et al., 1996]. Therefore, for

some applications one may require additional features for more accurate representation of the domain. The feature of *role transitivity* [Horrocks and Sattler, 1999], for example, may be desired in certain medical domains. Role transitivity can be defined as follows:

A role R is *transitive* if, for every interpretation $\mathcal{I}$, $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ and $(b^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$ implies that $(a^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$, where a, b and c are individual names.

Part-whole relations [Winston et al., 1987; Varzi, 1996; Artale et al., 1996; Keet, 2008] are examples of roles which exhibit this transitivity. These relations are used in various fields such as that of anatomy for categorizing the parts of the human body. For example if *aorta* isPartOf *heart* and *heart* isPartOf *chestCavity* then if the role represented by isPartOf is *transitive*, it follows that *aorta* isPartOf *chestCavity*. The symbol for the role transitivity feature in DLs is $\mathcal{S}$. If one applies this convention, then the DL $\mathcal{ALC}$ with the additional feature of role transitivity would be called $\mathcal{ALCS}$. However, for brevity and since $\mathcal{ALC}$ is widely considered as a "base" DL for building more expressive DLs, $\mathcal{ALC}$, together with the transitive roles feature, is known by the abbreviation $\mathcal{S}$. This is the starting base for a class of other expressive DLs such as $\mathcal{SHOIN}$which forms the basis for OWL-DL [Horrocks, 2005] (as of writing, superceded by OWL 2 [Cuenca-Grau et al., 2008]) as well as the $\mathcal{SHOIN}$ extension, $\mathcal{SROIQ}$ [Horrocks et al., 2006].

As mentioned, DLs are named according to the features they include. Usually each feature is represented by a unique letter (symbol) in the name of the DL. The more features a DL includes, the more expressive it is and vice versa. For $\mathcal{SHOIN}$, the $\mathcal{H}$ represents the feature of *role hierarchies* or *role inclusion*. For example one can state that $R \sqsubseteq S$ where R and S refer to atomic roles. The $\mathcal{O}$ represents usage of *nominals* [Schaerf, 1994]. Nominals allow one to represent further information about the actual elements in the domain. The symbol $\mathcal{I}$ represents the feature for expressing *inverse roles* [Horrocks and Sattler, 1999]. $\mathcal{N}$ represents *unqualified number restrictions* [Tobies, 2000]. $\mathcal{R}$ and $\mathcal{Q}$, which are included in $\mathcal{SROIQ}$, represent *complex role inclusions* [Horrocks and Sattler, 2004; Horrocks et al., 2000] and *qualified number restrictions* [Tobies, 2000; Baader et al., 2003], respectively. The features mentioned here are formally defined in Table 2.2.

	Feature	**Syntax**	**Semantics**
$\mathcal{C}$	Complex concept negation	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$ (C is complex)
$\mathcal{E}$	Full existential quantification	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid$ There is a $b \in \Delta^{\mathcal{I}}$ s.t. $(a,b) \in R^{\mathcal{I}}$ and $b \in C^{\mathcal{I}}\}$
$\mathcal{F}$	Functional roles	(funct R)	(funct $R)^{\mathcal{I}}$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ and there is no $c^{\mathcal{I}} \neq b^{\mathcal{I}}$ s.t. $(a^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$
$\mathcal{S}$	$\mathcal{ALC}$ with transitive roles	(trans R)	(trans $R)^{\mathcal{I}}$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ and $(b^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$ implies $(a^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$
$\mathcal{H}$	Role hierarchies	$R \sqsubseteq S$	$(R \sqsubseteq S)^{\mathcal{I}} = R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
$\mathcal{R}$	Role composition Complex role hierarchies	$R \circ S$ $R \circ S \sqsubseteq R$	$(R \circ S)^{\mathcal{I}} = \{(a,c) \mid$ There is $b \in \Delta^{\mathcal{I}}$ s.t. $(a,b) \in R^{\mathcal{I}}$ and $(b,c) \in S^{\mathcal{I}}\}$ $(R \circ S \sqsubseteq R)^{\mathcal{I}} = (R \circ S)^{\mathcal{I}} \subseteq R^{\mathcal{I}}$
$\mathcal{O}$	Nominals	$\{a_1, \ldots, a_n\}$	$(\{a_1, \ldots, a_n\})^{\mathcal{I}} = \{a_1^{\mathcal{I}}, \ldots, a_n^{\mathcal{I}}\}$
$\mathcal{I}$	Inverse roles	R^-	$(R^-)^{\mathcal{I}} = \{(b,a) \mid (a,b) \in R^{\mathcal{I}}\}$
$\mathcal{N}$	Number restrictions (# means cardinality of a set)	$\leq nR$ $\geq nR$	$(\leq nR)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\} \leq n\}$ $(\geq nR)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\} \geq n\}$
$\mathcal{Q}$	Qualified number restrictions (# means cardinality of a set)	$\leq nR.C$ $\geq nR.C$	$(\leq nR.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#\{b \in C^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\} \leq n\}$ $(\geq nR.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#\{b \in C^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\} \geq n\}$

Table 2.2: A list of common DL feature symbols

In addition to very expressive DLs such as $\mathcal{SHOIN}$ and $\mathcal{SROIQ}$, there are also classes of DLs which are designed for applications which require tractability (polynomial decidability) for reasoning tasks and that do not require the full expressive power of $\mathcal{ALC}$. These

inexpressive DLs include a smaller number of constructs which limits the kind of information one can represent but ensures lower complexity of reasoning. It is important to note that even though the expressivity of these DLs is limited, it is sufficient for numerous applications. For example, in the biomedical field, there are various large-scale ontologies being developed. These ontologies represent information and terminology from the biomedical domain for use in medical information systems and contain on the order of thousands of axioms.

SNOMED CT[1] (Systematized Nomenclature of Medicine, Clinical Terms) is a medical terminology which is an example of such an ontology. SNOMED is represented using a DL from the inexpressive $\mathcal{EL}$ family of DLs [Baader, 2003]. This DL is perfectly suitable for representing the kind of information that SNOMED CT requires and at the same time provides efficient reasoning properties. Roughly speaking, $\mathcal{EL}$ can be seen as a Horn [Horn, 1951] fragment of DLs. Its complexity is polynomial.

It is to be noted that SNOMED CT is an extremely large ontology having in the region of half a million terms (concept and role names)! Reasoning over such large-scale ontologies is an extremely complex task because of the sheer amount of information. The use of inexpressive DLs (such as $\mathcal{EL}$) for representing these ontologies simplifies the task considerably. Although $\mathcal{EL}$ provides few constructs (existential quantification, concept conjunction and the $\top$ concept), it has nevertheless proved to be extremely versatile in numerous applications where more expressive features are not required.

DLs were designed to be used in Knowledge Representation Systems where reasoning can be used to derive implicit information from an ontology. Important TBox reasoning services, provided by DLs, include checking if a concept description could contain any individuals (*Satisfiability testing for concepts*) and determining if one concept is a sub-concept or super-concept of another (*Subsumption testing*). Important ABox reasoning services include checking if a particular individual belongs to a certain concept (*Instance checking*) and checking if ABox statements are consistent with TBox statements (*Consistency checking*).

The computational complexity of these reasoning tasks vary with the expressivity of the DL [Levesque and Brachman, 1987]. $\mathcal{ALC}$ is an example of a DL which has a good balance between expressivity and complexity of reasoning. Concept satisfiability and ABox reasoning are applicable reasoning tasks for $\mathcal{ALC}$ because $\mathcal{ALC}$ includes both concept negation and ABox axioms. The computational complexity for concept satisfiability in $\mathcal{ALC}$ is PSPACE-complete [Schmidt-Schauß and Smolka, 1991]. Some popular extensions of $\mathcal{ALC}$ retain PSPACE-complete reasoning complexity (with respect to concept satisfiability). These extensions include DLs such as $\mathcal{SI}$ [Horrocks et al., 2000; Baader et al., 2008], $\mathcal{SIN}$ [Horrocks et al., 1998] and $\mathcal{ALCQO}$ [Baader et al., 2005b]. Complexity of reasoning increases considerably when role hierarchies are included in the DL. Therefore in DLs such as $\mathcal{SHIQ}$, $\mathcal{SHIO}$ and $\mathcal{SHOQ}$, the reasoning complexity increases to exponential time [Tobies, 2001; Hladik, 2004] with respect to concept satisfiability.

Less expressive DLs such as those in the $\mathcal{EL}$ family enjoy polynomial time reasoning complexity [Baader, 2003] because of the fewer constructors which are included in these logics. Even though $\mathcal{EL}$ does not include certain common constructors, most notably value restrictions ($\forall$), the expressivity is sufficient for a variety of applications. SNOMED CT, for example, is represented entirely in $\mathcal{EL}$. Large portions of Galen [Rector and Horrocks,

[1] `http://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html`

1997] and the Gene Ontology [Consortium, 2000] can also be represented using $\mathcal{EL}$. In the next section, we give an overview of the basic reasoning services which DLs provide, some of which were mentioned during this section.

2.3 Standard Reasoning Services

This section gives a brief overview of the common standard reasoning tasks for DLs. Many of these services are implemented in various DL reasoners (mentioned earlier). It is important to note that some of these reasoning tasks are applicable only to certain DLs. For example, the task of checking *concept satisfiability* (Section 2.3.1) is only applicable to DLs which include concept negation. Other important inference services which are discussed in this section include *subsumption testing* (checking whether a certain concept is more general than another), ontology *classification* (a special case of subsumption testing) and *consistency checking* (checking if an ontology has a model). It is worth mentioning that some of these reasoning tasks can be represented as special cases of other tasks. This principle of reducing one reasoning task to another is illustrated and discussed where applicable in this section.

2.3.1 Satisfiability

Satisfiability testing refers to the process of determining the possibility of (the extension of) a concept having any individuals (instances). This means that a concept C is *unsatisfiable* with respect to an ontology $\mathcal{O}$ if there is no model in which the interpretation of C is non-empty with respect to $\mathcal{O}$. More formally:

Definition 2.5 (Unsatisfiable concept)
Given an ontology $\mathcal{O}$, a concept C is *unsatisfiable* with respect to $\mathcal{O}$ if and only if there is no model $\mathcal{I}$ of $\mathcal{O}$ for which $C^{\mathcal{I}} \neq \emptyset$. For brevity, if a concept A is unsatisfiable with respect to an ontology $\mathcal{O}$, we say that $\mathcal{O}$ is *A-unsatisfiable* [Meyer et al., 2010] or equivalently we may refer to the *A-unsatisfiability* of $\mathcal{O}$.

In general, unsatisfiable concept *names* are seen as erroneous consequences of ontologies. In the context of debugging an ontology, one of the common tasks is to resolve the unsatisfiable concept names in the ontology. Note that there are many unsatisfiable concept *descriptions* (e.g., $A \sqcap \neg A$) in the ontology as well, but from a debugging perspective, the focus is primarily on repairing unsatisfiable concept *names*. Therefore, when we talk about an *unsatisfiable concept* in this dissertation, we refer to an unsatisfiable concept *name* A or C (unless otherwise stated).

Definition 2.5 has a useful consequence. That is, C is unsatisfiable with respect to an ontology $\mathcal{O}$ if and only if: $\mathcal{O} \models C \sqsubseteq \bot$ (if and only if $\mathcal{O} \models C \equiv \bot$).

This consequence illustrates that the unsatisfiability of a concept can be represented as an entailment test and is useful for certain approaches to ontology repair. This is elaborated on in Chapter 4. Unsatisfiable concept names usually arise because of inconsistent definitions

for entities in the ontology. Consider the following example. (Throughout the example ontologies in this dissertation, we may refer to axioms by their provided indices).

Example 2.1 Consider the following ontology:

$$\mathcal{O} = \left\{ \begin{array}{l} 1.\ A \sqsubseteq B, \\ 2.\ A \sqcap B \sqsubseteq \bot, \\ 3.\ B \sqsubseteq C \end{array} \right\}$$

It turns out that A is unsatisfiable with respect to $\mathcal{O}$. This can be verified as follows: for Axiom 1 to be satisfied in an interpretation $\mathcal{I}$, it means that $A^{\mathcal{I}} \subseteq B^{\mathcal{I}}$. For Axiom 2 to be satisfied in $\mathcal{I}$, it means that $A^{\mathcal{I}} \cap B^{\mathcal{I}} \subseteq \emptyset$. It is clear from these two statements that the only interpretations $\mathcal{I}$ which satisfy these two axioms are ones in which $A^{\mathcal{I}} = \emptyset$. Therefore, the concept referred to by A is unsatisfiable. □

Unsatisfiable roles are another class of possible defects in ontologies, although they are not as prevalent as unsatisfiable concepts. Role satisfiability will not be explicitly characterized and discussed. However it can be demonstrated that the problem is reducible to concept satisfiability for ontology repair purposes [Kalyanpur, 2006]. This is illustrated by the fact that checking if a role R is satisfiable, is equivalent to checking if the *concept* $\geq 1.R$ or $\exists R.\top$ is satisfiable. Therefore the existing techniques for checking *concept* satisfiability are applicable to checking *role* satisfiability as well.

2.3.2 Subsumption

In DL systems, the "is-a" role [Brachman, 1983] has special significance. The reason for this is that the meaning of the is-a relation is so applicable in various application domains. In the context of a university, for example, a Lecturer can be represented as a type of Employee. In academia, the Artificial Intelligence Discipline can be seen as a type of Computer Science Discipline. The is-a relation is used in DLs to represent this kind of relationship. The specification and definition of roles in an ontology is usually left up to the ontology engineer but because the is-a role is so inherent in many application domains, it is encoded into the syntax and semantics of DLs as *subsumption* (as we have seen in Section 2.2.1).

It can be shown that concept subsumption (as well as concept equivalence and disjointness) can be reduced to concept satisfiability in DLs permitting full negation [Horrocks and Patel-Schneider, 2004]. This reduction is illustrated by the following validities [Baader et al., 2003]:

(i) C *subsumed by* D if and only if $(C \sqcap \neg D) \sqsubseteq \bot$
(ii) C *equivalent to* D if and only if $(C \sqcap \neg D) \sqsubseteq \bot$ and $(D \sqcap \neg C) \sqsubseteq \bot$
(iii) C *disjoint with* D if and only if $(C \sqcap D) \sqsubseteq \bot$

Statement (i) means that the subsumption axiom $C \sqsubseteq D$ follows from a particular TBox if and only if the complex concept $C \sqcap \neg D$ is unsatisfiable (necessarily empty) in that TBox. Therefore, in this case, the problem of subsumption can be reduced to the problem of satisfiability as long as the DL includes (full) concept negation. The obvious intuition here is that if every individual in the interpretation of C is included in the interpretation of

D, then there cannot be an individual which is in the interpretation of C but *not* in the interpretation of D. The same applies for (ii) but in both directions since $C \equiv D$ captures the meaning of both axioms $C \sqsubseteq D$ and $D \sqsubseteq C$.

The subsumption problem for concepts is defined in a similar way for *roles* [Horrocks et al., 2000] although this is not discussed in further detail here.

2.3.3 Classification

A special case of the problem of subsumption testing discussed in the previous section, is the problem of *classifying* an ontology. The problem is known by various other names in the literature such as: computing the *subsumption hierarchy* of the ontology, computing the *concept hierarchy* of the ontology or computing the *taxonomy* of the ontology. What the process essentially entails is determining the sub-concepts and super-concepts for all concept *names* in the ontology. The ontology engineer is (generally) not interested in the subsumption hierarchy of complex concepts because there are an infinite number of these in the ontology and the engineer is more interested in the relationships between the concept *names* which are asserted by him/her in the ontology. Being able to classify an ontology is a primary capability of most DL reasoners. The complexity of classification depends on the complexity of the particular DL. Therefore much emphasis in the literature is placed on optimizing the process, especially in reasoners designed for more expressive DLs [Sirin et al., 2007; Tsarkov and Horrocks, 2006; Shearer et al., 2008]. Figure 2.1 depicts the subsumption hierarchy for the Pizza ontology [Rector et al., 2004] in the Protégé[2] ontology editor.

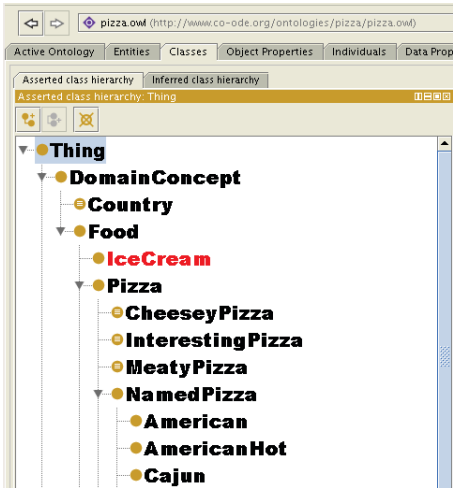

Figure 2.1: Portion of the concept hierarchy for the Pizza ontology in Protégé 4.

2.3.4 Instance Checking

Given an individual name a and a (possibly complex) concept C in an ontology, instance checking determines if the individual a belongs to C. A formal definition:

[2] `http://protege.stanford.edu`

Definition 2.6 (Instance)
An individual a is an *instance* of some concept C with respect to ontology $\mathcal{O}$ if and only if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all interpretations $\mathcal{I}$ of $\mathcal{O}$ if and only if $\mathcal{O} \models C(a)$ (also written $\mathcal{O} \models a : C$).

The problem of instance checking has been proven to be reducible to the problem of consistency checking [Baader and Sattler, 2001]. This can be observed in the following proposition: There exists a model $\mathcal{I}$ for $C(a)$ in ontology $\mathcal{O}$ if there is no model $\mathcal{I}$ for $\mathcal{O} \cup \{\neg C(a)\}$. This translates to the fact that an individual a is an instance of some concept C with respect to the ontology $\mathcal{O}$, if adding the negated assertion (a is an instance of the *complement* of C) to $\mathcal{O}$, makes $\mathcal{O}$ inconsistent. The notion of ontology consistency checking is introduced in the next section.

2.3.5 Consistency Checking

Consistency checking is the task of determining if an ontology has a model (if there is an interpretation which satisfies all the sentences in the ontology). If an ontology does not have a model, it is *inconsistent*. An inconsistent ontology is an erroneous situation because in such an ontology, "everything is false", i.e., $\top \sqsubseteq \bot$ holds. In ontology development, there are various reasons why an inconsistent ontology may occur. All the reasons boil down to contradicting statements specified in the ontology but the most common culprits are incorrect ABox assertions. For example, if the ontology engineer specifies that a is an instance of C in the ABox, i.e., $C(a)$, and if C turns out to be unsatisfiable, i.e., $\mathcal{O} \models C \equiv \bot$ then a logical contradiction arises, namely "a is an element of $\emptyset$".

DL versions of Tableau algorithms [Baader and Sattler, 2001] for determining if an ontology is consistent try to construct a model of the ontology. That is, the algorithm tries to construct a finite interpretation $\mathcal{I}$ that satisfies each sentence in the ontology. If it does not find such an interpretation then the ontology is inconsistent. If such an interpretation is found then the ontology is consistent.

Tableau algorithms for various other reasoning tasks are closely related to the one for consistency checking. This is the reason why certain reasoning problems can be expressed in terms of others. For example, the tableau algorithm for satisfiability in $\mathcal{ALC}$ works by trying to construct a finite interpretation in which the concept under consideration, is non-empty. It does this by applying a series of transformations to a starting ABox which contains a single statement $C(a)$ where C is the concept being checked for satisfiability and a is an individual. If C is empty in all interpretations then it is clear that the assertion $C(a)$ will generate an inconsistent ontology. Therefore the algorithm checks the satisfiability of the concept by trying to generate an inconsistent ontology. It is clear then, that the two tasks of checking ontology consistency and concept satisfiability are related. It is fairly straightfoward to see that the problem of instance checking can be related in a similar way.

The interested reader is referred to the references for details on the tableau algorithms [Baader and Sattler, 2001] and the alternative methods to the tableau approach (structural algorithms [Nebel, 1990]) for the reasoning tasks described. The most common standard DL reasoning services were introduced and discussed in this section. It was mentioned that many of these services can be expressed as other related reasoning problems. Certain tasks such as consistency checking and instance checking are only applicable for DLs which

include the use of ABox assertions. Satisfiability testing is only applicable in DLs which are able to express unsatisfiable concept names. In the next section we introduce some of the relevant non-standard reasoning services for DLs.

2.4 Non-Standard Reasoning Services

The standard reasoning services are primarily concerned with identifying different kinds of consequences from the ontology. Non-standard reasoning services are another category of inference services which have various applications in ontology development and maintenance. The most widely used non-standard reasoning services for ontology debugging and repair are those of *explanation* [Kalyanpur et al., 2007; Horridge et al., 2008] and *repair* [Kalyanpur, 2006; Kalyanpur et al., 2006b].

Explanation services are concerned with discovering *why* certain (possibly unwanted) consequences follow from the ontology, while repair is involved with formulating methods to modify the ontology in an intelligent way, so as to eliminate the *unwanted* consequences. A brief overview of these two services is given in this section. The focus is only on these specific services because they form part of existing methods for ontology debugging and repair, which is the subject of this dissertation. There are various other non-standard DL reasoning services available such as finding the *least common subsumer* [Baader et al., 2003], *concept approximation* [Baader et al., 2000; Brandt et al., 2002] and *modularization* [Herzig and Varzinczak, 2006; Cuenca-Grau et al., 2006]. However the details of these services are beyond the scope of this dissertation.

2.4.1 Explanation

Explanation is a service within the broader area of ontology debugging and repair which focuses on explaining *why* selected consequences follow from the ontology. A basic kind of "explanation" for some entailment $\mathcal{O} \models \alpha$, where α is an axiom (recalling our convention from Section 2.2.1, of referring to a general sentence in the ontology as an axiom), is a set $\mathcal{O}' \subseteq \mathcal{O}$ such that $\mathcal{O}' \models \alpha$. $\mathcal{O}'$ is seen as a "reason" for α following from $\mathcal{O}$.

Explanations provide information to ontology engineers which helps them understand why certain entailments hold. Therefore if the ontology engineers discover certain *undesired* entailments, they are able to use explanation to identify which axioms in the ontology cause the entailment. This information is very useful when developing a method to eliminate the undesired entailment.

A special kind of explanation, which is central to ontology debugging and repair, is called a *justification* (also known as a Minimal Unsatisfiability Preserving Sub-TBox (MUPS) [Schlobach and Cornet, 2003] or Minimal Axiom Set (MinA) [Baader et al., 2007]). A justification is a *minimal* explanation for some ontology consequence. More formally:

Definition 2.7 (Justification for entailment)
Let $\mathcal{O}$ be an ontology and α an axiom such that $\mathcal{O} \models \alpha$. A set of axioms $J \subseteq \mathcal{O}$ is a *justification* for $\mathcal{O} \models \alpha$ if and only if $J \models \alpha$ and there is no $J' \subset J$ such that $J' \models \alpha$.

This principle of a justification is illustrated with the following example:

Example 2.2 Consider the following ontology:

$$\mathcal{O} = \left\{ \begin{array}{l} 1.\ A \sqsubseteq B \sqcap \exists R.C, \\ 2.\ D \sqsubseteq G \sqcap B, \\ 3.\ C \sqsubseteq F \end{array} \right\}$$

One can see that $\mathcal{O} \models A \sqsubseteq \exists R.F$. It is clear that this entailment follows from Axioms 1 and 3. Therefore $\{1, 3\} \subseteq \mathcal{O}$ is an explanation for $\mathcal{O} \models A \sqsubseteq \exists R.F$. It is also clear that $A \sqsubseteq \exists R.F$ does not logically follow just from $\{1\}$. Similarly it does not logically follow just from $\{3\}$. These two sets are the only two proper (non-empty) subsets of the explanation $\{1, 3\}$. Therefore, by Definition 2.7, one can conclude that $\{1, 3\}$ is a justification for $\mathcal{O} \models A \sqsubseteq \exists R.F$. □

The next section introduces the subject of *repair* which deals with resolving an erroneous entailment once the reason for its existence is discovered.

2.4.2 Repair

Repair is the name given to the process of analyzing generated explanations for unwanted consequences and using this information to identify appropriate modifications to the ontology which will eliminate these consequences. In Example 2.2, it is clear that Axioms 1 and 3 are responsible for the entailment $\mathcal{O} \models A \sqsubseteq \exists R.F$. If one considers this entailment as an *unwanted* consequence, one can eliminate it by removing Axiom 1 or Axiom 3 (or both) from $\mathcal{O}$. Although the example is trivial and not realistic in terms of the number of axioms in the ontology, the same principles hold in a larger context.

An important aspect of repair is reducing, as far as possible, the amount of modifications to the ontology to eliminate the entailment. Therefore, in Example 2.2, removing Axiom 1 or Axiom 3 (rather than both) is a better repair strategy for $\mathcal{O} \models A \sqsubseteq \exists R.F$. The notion of repair is characterized further in the next chapter.

This chapter has introduced the family of knowledge representation formalisms known as Description Logics. The basic DL $\mathcal{ALC}$ was presented and some common extensions to this language were mentioned and discussed. Issues about DL complexity, DL expressivity and DL reasoning tasks were also briefly discussed. The next chapter outlines the problem of *ontology debugging and repair* and explains in detail some of the existing approaches which address the issue.

Chapter 3

Ontology Debugging and Repair

The process of identifying errors in the design of an ontology, understanding why they arise, and devising strategies to fix them is generally known as *ontology debugging and repair*. There are various approaches to addressing these issues as described in the literature. The most widely used strategies are discussed in this chapter. The first part of the chapter presents the existing approaches for *explanation* for semantic defects (unsatisfiable concept names and inconsistent ontologies) in DL ontologies. In the second part of the chapter, the attention is turned to identifying strategies to *repair* (eliminate) these defects.

3.1 DL Ontology Errors

Errors in ontologies represented using OWL languages generally fall into three categories. These are *syntactic*, *semantic* and *style* defects [Kalyanpur, 2006]. Because OWL is based on DLs, these same terms are used here to describe error categories for DL ontologies.

Syntactic defects in DL ontologies imply the inclusion of illegal syntax used in the ontology which falls outside the expressivity of the particular DL chosen for the application. For example in the Description Logic $\mathcal{EL}$, the constructs included in the language are the $\top$ concept, the conjunction ($\sqcap$) concept constructor and the existential ($\exists$) quantifier. The concept description $\neg C$ is therefore not a valid $\mathcal{EL}$ concept description because concept negation is not included in the $\mathcal{EL}$ language.

Semantic defects are the most extensively studied defects and are the target for the bulk of research on ontology debugging and repair. These defects are logical errors arising in the ontology description due to modelling errors. The main examples of these defects are unsatisfiable concepts and inconsistent ontologies. Another interesting semantic defect is the *unintended inference* or *unwanted axiom*. An unwanted axiom is one that follows from the ontology but is not intended to do so by the ontology engineer and the domain expert. The meaning of such an axiom is inconsistent with what is known about the domain of interest. Unwanted axioms are categorized as *style* defects in OWL ontologies [Kalyanpur, 2006]. The definition of a semantic defect (with respect to a DL ontology) which is used in this dissertation is: any logical consequence of the ontology which is not desired by the ontology engineer and domain expert. It is clear from this definition that an unwanted axiom is better classified as a semantic defect and not a style defect.

Style defects do not necessarily imply logical inconsistencies or invalid syntactical descriptions in the ontology. These defects are simply symptoms of untidy ontology modelling. They do not impact to a large extent on the meaning of terms in the ontology. Examples of style defects include: *unreferenced entities*, *cyclic definitions* and *conflicting ranges* in terms [McGuinness et al., 2000].

Existing ontology debugging and repair approaches focus primarily on semantic defects such as unsatisfiable concepts and inconsistent ontologies. The remainder of this chapter presents some of the existing approaches to correcting specifically these types of errors. *Syntactic* and *style* defects are disregarded for the remainder of this dissertation. The usual first step of debugging is to ascertain why the error holds in the ontology. This means finding the information in the ontology which is responsible for these errors. This task is dealt with by *explanation*. The field of explanation is concerned with finding the assertions in the ontology which cause the unsatisfiability or inconsistency.

Therefore, the need for explanation in ontology debugging and repair is motivated and discussed. After this, the special explanation known as a *justification*, is characterized. Justifications are the most widely used explanation because of their conciseness. The existing methods for computing justifications, which feature in many state-of-the-art ontology debugging tools, are presented in detail.

Lastly, in Section 3.7, the usefulness of justifications for identifying *repairs* (ontology modifications for eliminating the error) is then discussed, along with some existing methods for identifying repairs. A few approaches for resolving *multiple* errors are also presented. Finally, in Section 3.8, some of the limitations of existing explanation and repair methods are outlined and discussed.

3.2 Explanations for Errors

During ontology repair, a natural step after detecting errors in the ontology is to understand *why* they have occurred. This understanding helps the ontology engineer to make appropriate plans to *correct* the errors. In large ontologies with many axioms and terms (concept, role and individual names), the task of locating the causes of the errors is very tedious and hence automated tools which perform this task for the ontology engineer are very useful.

The idea of explanation is to do just that, locate and explain the *causes* of the errors in the ontology. Explanation methods can be used to find the causes of any logical consequence of the ontology hence the same methods may be used to understand why the *erroneous* consequences of the ontology occur. This information, in turn, helps the ontology engineer to develop strategies to eliminate these consequences.

The intuition behind explanation methods is that any consequence of the ontology is supported (caused) by some subset of axioms from the ontology. This set of axioms which is responsible for the error in the ontology is called an *explanation*. An explanation is therefore any subset of the ontology from which the error logically follows.

The identification of explanations for errors is now a widely used ontology debugging service.

Many existing ontology editors/browsers such as Protégé[1], OWLSight[2] and SWOOP[3] include or have support for explanation services. The majority of these tools generate special types of explanations called *justifications* [Schlobach and Cornet, 2003; Baader and Hollunder, 1995; Kalyanpur et al., 2007; Suntisrivaraporn et al., 2008].

The term *justification* is also referred to by the name *Minimal Unsatisfiability Preserving Sub-TBox* (MUPS [Schlobach and Cornet, 2003]) and *Minimal Axiom Set* (MinA [Baader et al., 2007]). Intuitively, a justification for some unsatisfiable concept in the ontology is a minimal subset (with respect to set inclusion) of the ontology from which it follows that the concept is unsatisfiable [Horridge et al., 2009a]. A definition for justification for a *general* entailment has been given in Section 2.4.1.

A specialized definition for a justification for an *unsatisfiable concept* is given below. Note that in the rest of this chapter, the focus will be on presenting the existing methods for generating justifications for and repairing of *unsatisfiable concepts* and *general entailments* in the ontology. However, the notion of justifications and repairs can be easily extended for *inconsistent ontologies* as well, as we will briefly see later in the chapter.

Definition 3.1 (Justification for unsatisfiable concept)
Let $\mathcal{O}$ be an ontology and C a concept name in $\mathcal{O}$ such that $\mathcal{O}$ is C-unsatisfiable. A set of axioms $J \subseteq \mathcal{O}$ is a *justification* for the C-unsatisfiability of $\mathcal{O}$ if and only if J is also C-unsatisfiable and there is no $J' \subset J$ such that J' is C-unsatisfiable.

An example is given to illustrate the above definition. Recall that the ontology in Example 2.1 is A-unsatisfiable. The only justification, J, for the A-unsatisfiability of $\mathcal{O}$ is $\{A \sqsubseteq B,\ A \sqcap B \sqsubseteq \bot\}$. For J to be a justification, J has to be A-unsatisfiable and no proper subset of J should be A-unsatisfiable. It is clear that $J \subseteq \mathcal{O}$ is A-unsatisfiable and that all the non-empty proper subsets of J, namely, $\{A \sqsubseteq B\}$ and $\{A \sqcap B \sqsubseteq \bot\}$ are *not* A-unsatisfiable. □

In general, because justifications are minimal subsets of the ontology from which the unsatisfiability follows, they are a more concise kind of explanation. This makes them better suitable for use in generating concise *repairs*. The relationship between justifications and repairs will be discussed later on in this chapter. Suffice it to say for now that a *repair* for some error in an ontology is a set of changes that may be applied to the ontology to correct the error. The first notions regarding *computation* of justifications (also known as *axiom pinpointing*) in DL-based Knowledge Representation, were introduced by Baader and Hollunder [1995] and Schlobach and Cornet [2003]. In general, methods for computing justifications are divided into *Black-box* [Kalyanpur et al., 2007; Horridge et al., 2009a; Wang et al., 2005] and *Glass-box* [Kalyanpur et al., 2005b; Meyer et al., 2006; Schlobach and Cornet, 2003; Lam et al., 2008] approaches. However, there are also approaches which use a combination of these. These are known as *Hybrid* [Kremen and Kouba, 2009] approaches. This categorization of methods depends on how the reasoner is used in the computation of the justifications.

Black-box methods use the reasoner solely for checking if an entailment holds (or if a

[1] `http://protege.stanford.edu`
[2] `http://pellet.owldl.com/ontology-browser`
[3] `http://www.mindswap.org/2004/SWOOP`

concept is satisfiable) in an ontology or not. The internals of the reasoner (how it performs this check) is not seen as important. The reasoner is therefore treated as a "black-box" entailment/satisfiability checker whose sole function is to say 'yes' or 'no' to queries. Black-box methods, therefore, can be used regardless of the choice of reasoner. They are known as *reasoner independent* methods. Glass-box methods, on the other hand, involve a modification of the procedures inside the reasoner which is usually non-trivial. This requires an awareness of the *inner-workings* of the reasoner, hence the "glass-box" name. Glass-box methods are bound to a specific reasoner (they are reasoner dependent).

DL reasoners are usually designed for a specific subset of DLs and therefore their reasoning algorithms are tailored specifically for these DLs. One example of such a pairing is the CEL reasoner[4] developed for the $\mathcal{EL}$ family of DLs. For example, if a Glass-box method for finding justifications uses a modified version of CEL's reasoning algorithms, then this method is bound to that reasoning implementation and cannot be used with a different reasoner.

The following three sections describe some existing Black-box, Glass-box and Hybrid methods for computing justifications. More detail is presented for the Black-box approaches while a much more general presentation of existing Glass-box and Hybrid approaches is given. The reader of this dissertation is referred to the references provided for detailed descriptions of Glass-box and Hybrid approaches. The reason for focusing on *Black-box* methods is that the main contribution of this dissertation (discussed in Chapter 4) is to provide an alternative method for ontology debugging and repair and this method uses a Black-box approach with regards to computing justifications.

3.3 Computing Justifications: Black-box Approach

Black-box methods for computing justifications are so named because the reasoner is used solely as a "black-box" for checking if an entailment holds in an ontology or not. There is no concern about the internals of the reasoner. Since the service of checking entailment (or checking satisfiability) is a standard capability of most existing DL reasoners, the only requirement for Black-box methods is a sound and complete reasoner for the particular description logic being used.

The existing Black-box algorithms are centered around removing axioms from the ontology that are irrelevant to the unsatisfiability of the concept under consideration. This is done using sophisticated *expand/contract* or *expand/shrink* [Kalyanpur et al., 2007] techniques until the remaining axioms in the ontology constitute a justification for the entailment. The upside to this approach is that it is applicable regardless of the reasoner being used (reasoner independent). A down-side to this approach is that each time axioms are removed from the ontology, the reasoner has to be called to check satisfiability. This check is, in general, computationally expensive and many such checks are typically required to compute a justification. As such, Black-box methods have various optimization techniques to increase the general performance of computing justifications.

The rest of this section shows how justifications are computed using the Black-box approach

[4] `http://lat.inf.tu-dresden.de/systems/cel`

including the optimization techniques mentioned above. The section is split into two subsections. The first discusses computing a *single* justification for an unsatisfiable concept. The second presents the methods for computing *all* justifications for an unsatisfiable concept.

3.3.1 Computing a Single Justification

This section begins with a presentation of one of the simpler (albeit unoptimized) algorithms for determining a single justification. It is commonly known as the *naïve pruning algorithm* [Suntisrivaraporn et al., 2008]. The algorithm's simplicity makes it a good place to start when formulating a better performance Black-box method for computing a justification.

Naive Pruning Algorithm

Given an ontology $\mathcal{O}$ which is C-unsatisfiable, Algorithm 1 extracts a set of sentences from $\mathcal{O}$ with the property that this set is a justification for $\mathcal{O}$ being C-unsatisfiable.

Algorithm 1: *slowContract* (Single justification)

Input: Ontology $\mathcal{O}$ and concept C such that $\mathcal{O}$ is C-unsatisfiable
Output: Justification J for $\mathcal{O}$ being C-unsatisfiable

```
1 J := O;
2 foreach α ∈ J do
3 |   if J\{α} is C-unsatisfiable then
4 |   |   J := J\{α};
5 |   end
6 end
7 return J;
```

Algorithm 1 works by moving in a stepwise fashion through each axiom α in the ontology. At each step it removes the current axiom from the ontology and then invokes the reasoner to check if C is still unsatisfiable in the ontology. If the ontology is still C-unsatisfiable, then the removed axiom is not *critical* to the entailment and can be discarded. If the ontology is no longer C-unsatisfiable then it means that the removed axiom is critical to the C-unsatisfiability of $\mathcal{O}$ and is therefore added back into the ontology. After one pass through the ontology the algorithm returns a justification for C-unsatisfiability of $\mathcal{O}$. This method (removing axioms from the ontology until we arrive at a justification) is sometimes known as *contraction*[5].

A brief study of Algorithm 1 shows its correctness. It is obvious that any ontology $\mathcal{O}$ which is C-unsatisfiable has some subset ($J \subseteq \mathcal{O}$) which is also C-unsatisfiable. Examining Lines 1 and 2 of Algorithm 1 one can see that the algorithm considers each and every axiom in the ontology when searching for the justification. Lines 3 and 4 ensure that the axioms which have no effect on the satisfiability of C (when removed from the ontology), are discarded. These two properties prove that J as returned by Algorithm 1 is a justification for $\mathcal{O}$ being C-unsatisfiable.

[5]The term contraction refers to the reduction of the ontology in size.

Note that the condition on Line 3 of Algorithm 1 asks the reasoner to check if C is unsatisfiable with respect to the current J. This check is equivalent to asking the reasoner if the entailment $\mathcal{J} \models C \sqsubseteq \bot$ holds. This can be done because subsumption testing can (in DLs which include concept negation such as $\mathcal{ALC}$) be reduced to satisfiability checking and vice versa. This is mentioned in Chapter 2. As an example, if one wants to test if $\mathcal{O} \models A \sqsubseteq B$ then it is equivalent to checking if $\mathcal{O}$ is $(A \sqcap \neg B)$-unsatisfiable (as stated in Section 2.3.2). The following example illustrates how Algorithm 1 works to compute a single justification for a general entailment.

Example 3.1 Consider the following ontology:

$$\mathcal{O} = \left\{ \begin{array}{l} 1.\ G \sqsubseteq H, \\ 2.\ A \sqsubseteq D \sqcap \forall R.E, \\ 3.\ A \sqsubseteq B \sqcap C, \\ 4.\ C \equiv \forall R.E, \\ 5.\ F \sqsubseteq \exists S.G \end{array} \right\}$$

It is fairly straightfoward to see that $\mathcal{O} \models A \sqsubseteq C$. The most obvious reason for this consequence is Axiom 3. The axiom $A \sqsubseteq B \sqcap C$ implies that $A \sqsubseteq B$ and $A \sqsubseteq C$. But if one examines $\mathcal{O}$ more closely one can see that Axiom 3 is not the only reason for $\mathcal{O} \models A \sqsubseteq C$. In fact it can be shown that $\mathcal{O} \backslash \{A \sqsubseteq B \sqcap C\} \models A \sqsubseteq C$. From Axiom 2 and Axiom 4 it follows that $A \sqsubseteq D \sqcap C$, and through a similar argument for Axiom 3, one can arrive at the consequence $A \sqsubseteq C$. Thus, there are two justifications for the entailment $\mathcal{O} \models A \sqsubseteq C$, one justification being $\{A \sqsubseteq B \sqcap C\}$ and the other $\{A \sqsubseteq D \sqcap \forall R.E, C \equiv \forall R.E\}$. □

A demonstration of Algorithm 1 will now be given to compute one of the established justifications in Example 3.1. With input ontology $\mathcal{O}$ and $\alpha = A \sqsubseteq C$, Algorithm 1 works in the following way:

Firstly the output justification set J is assigned the set of axioms from $\mathcal{O}$ (Line 1). Because $\mathcal{O} \models \alpha$, it is known that $J \models \alpha$. Lines 2 to 4 remove the axioms that do not form part of the justification. First, Axiom 1 is removed from J and the reasoner is asked whether $J \backslash \{1\} \models A \sqsubseteq C$ (Line 3). In this example it does and so Axiom 1 is discarded permanently from J (Line 4).

J is now the set of axioms $\{2, 3, 4, 5\}$. Next, Axiom 2 is removed from J. $J \backslash \{2\} \models A \sqsubseteq C$ holds because of Axiom 3. Axiom 2 is therefore permanently removed from J. J is now reduced to $\{3, 4, 5\}$. After removing Axiom 3 it follows that $J \backslash \{3\} \not\models A \sqsubseteq C$. Axiom 3 is therefore added back to the ontology and J remains $\{3, 4, 5\}$.

The next axiom (Axiom 4) is removed. The reasoner should tell us that $J \backslash \{4\} \models A \sqsubseteq C$. This is due to the presence of Axiom 3 in J. Axiom 4 is therefore discarded and J becomes $\{3, 5\}$. The last axiom (Axiom 5) is then processed similarly. All the axioms in $\mathcal{O}$ have thus been processed and the loop in Line 2 terminates. The set $J = \{3\}$ is returned by the algorithm. This set is indeed a justification for $\mathcal{O} \models \alpha$ as shown earlier. □

Sliding Window Technique

Although Algorithm 1 is correct it is also computationally expensive because it requires the same number of entailment checks as there are axioms in the ontology. An entailment check is a complex task and thus one should try to reduce the number of these checks as far as possible when computing justifications to achieve acceptable performance. One optimization that could reduce the number of entailment checks is to remove a set of axioms (more than one) from the ontology before performing the entailment check. Algorithm 2 uses this key optimization when computing a justification:

Algorithm 2: *fastContract* (Single justification - Sliding window)

Input: Ontology $\mathcal{O}$, axiom α such that $\mathcal{O} \models \alpha$ and window size $k \geq 1$
Output: Justification J for $\mathcal{O} \models \alpha$

```
1  J := O;
2  W := ∅;
3  while k ≥ 1 do
4      W := getNextWindow(J, k);
5      while |W| ≠ 0 do
6          if J\W ⊨ α then
7              J := J\W;
8          W := getNextWindow(J, k);
9      k := ⌊k/2⌋;
10 return J;
```

Algorithm 2 uses an optimization which removes *sets* (a "window") of axioms at a time from the ontology instead of a single axiom. This set is chosen by the *getNextWindow* procedure and the number of axioms in this set is determined by the input window size k. The benefit of the above mentioned optimization is that if the removal of the set of axioms has no effect on the entailment under consideration, then the axioms can be permanently discarded from the ontology. And because a *set* of extraneous axioms is removed at once, instead of just one, the algorithm is that much closer to pinpointing the justification. Most importantly, $k - 1$ entailment checks have been saved, where k is the number of axioms removed. Thus the overall number of entailment checks required to compute the justification is reduced. The *sliding window technique* [Kalyanpur, 2006] is an example of a method which employs this kind of axiom pinpointing strategy.

The question arises: what value is appropriate to use as a starting window size? In theory, k can be any positive integer. However, in practical terms there have been experiments conducted which indicate that, in the context of ontologies with a large amount of axioms, the *greater* of two numbers, $\frac{|\mathcal{O}|}{10}$ or 10 is chosen as the starting window size where $|\mathcal{O}|$ is the number of axioms in the ontology. These values have been shown empirically to provide optimal performance when applied to real-world ontologies [Kalyanpur et al., 2007].

Algorithm 2 uses a version of the sliding window technique described above, and is demonstrated here with the aid of an example.

Example 3.2 Consider an ontology $\mathcal{O} = \bigcup_{i=1}^{5} w_i$, where each w_i is as follows:

$$w_1 = \left\{ \begin{array}{l} 1.\ A \sqsubseteq C, \\ 2.\ A \sqsubseteq \exists R.E, \\ 3.\ E \sqsubseteq H \end{array} \right\}, w_2 = \left\{ \begin{array}{l} 4.\ D \equiv F \sqcup G, \\ 5.\ F \sqsubseteq C, \\ 6.\ G \sqsubseteq C \sqcap \exists R.H \end{array} \right\}, w_3 = \left\{ \begin{array}{l} 7.\ H \equiv E \sqcup L, \\ 8.\ L \sqsubseteq M, \\ 9.\ M \equiv N \sqcap \exists R.H \end{array} \right\},$$

$$w_4 = \left\{ \begin{array}{l} 10.\ N \sqsubseteq P, \\ 11.\ B \equiv D \sqcap \exists R.E, \\ 12.\ C \sqsubseteq D \end{array} \right\}, w_5 = \left\{ \begin{array}{l} 13.\ P \equiv Q \sqcap U, \\ 14.\ Q \sqsubseteq H, \\ 15.\ U \sqsubseteq L \sqcup P \end{array} \right\}$$

Axiom $A \sqsubseteq B$ follows from $\mathcal{O}$, therefore the entailment $\mathcal{O} \models A \sqsubseteq B$ holds. This consequence can be deduced from Axioms 1, 2, 11 and 12 in the following way: from Axioms 1 and 12 it follows that A is included in D ($A \sqsubseteq D$). From Axiom 2 it is known that A is also included in the complex concept $\exists R.E$ (i.e., $A \sqsubseteq \exists R.E$). Therefore it follows that $A \sqsubseteq D \sqcap \exists R.E$. Finally Axiom 11 tells us that B is equivalent to $D \sqcap \exists R.E$ and thus it is found that $A \sqsubseteq B$. In conclusion, it follows that the set $\{1, 2, 11, 12\}$ is a justification for $\mathcal{O} \models A \sqsubseteq B$. This set also happens to be the *only* justification for $\mathcal{O} \models A \sqsubseteq B$.

Algorithm 2 will now be demonstrated to be able to compute this justification. The input ontology is $\mathcal{O} = w_1 \cup \ldots \cup w_5$, the input axiom $A \sqsubseteq B$ and a starting window size of three is chosen. The value three for the starting window size is chosen because it is more appropriate for the unrealistically small ontology in the example. The starting window size of three also indicates the reason for the example ontology $\mathcal{O}$ being broken up into the "windows" w_1 – w_5, each having exactly three axioms. The output justification J is assigned the set of all axioms in the original ontology $\mathcal{O}$. The sliding window analogy now becomes apparent as the focus is now on a certain portion (window) of the ontology. A window size of three means that at each iteration a set of at most three axioms is considered.

In the example ontology, the window w_1 consisting of Axioms 1, 2 and 3, is first considered. This set is temporarily removed from J. The entailment check $J \backslash w_1 \models A \sqsubseteq B$ is performed. If this holds then J can be reassigned as being the axioms in $w_2 \cup ... \cup w_5$, permanently discarding w_1. Otherwise, if this entailment does not hold, w_1 is added back to J. In the current example it is the case that $J \backslash w_1 \not\models A \sqsubseteq B$. w_1 is therefore added back to J. The window of focus then "slides" to w_2 (the next three axioms), which is processed in a similar fashion. In this way the procedure continues until the window in focus is empty (reached the end of the ontology). After this, J potentially has much fewer axioms than the start. After applying this initial procedure to the example, J is reduced to $\{1, 2, 3, 10, 11, 12\}$.

The next step is to shrink the window size by some factor so there is a finer-grained view of the new J. Then the same sliding window technique is applied again on J with the *new* window size. This is repeated until the window size is equal to one. It is easy to see that in this particular case Algorithm 2 performs in exactly the same way as Algorithm 1. It is important to note that, to ensure computation of a justification, there has to be one pass of Algorithm 2 executed with a window size of one. Therefore the factor to reduce the window size by, between passes, must be chosen such that the window size is reduced to one at some stage.

In the example, the factor which is chosen is half ($\frac{1}{2}$) [Kalyanpur et al., 2007]. Again,

this particular factor is chosen because it is shown to provide reasonable performance in practice. Returning to the example, the new window size becomes $\lfloor \frac{3}{2} \rfloor = 1$. In this case, one axiom is removed at a time and the entailment is monitored after each removal, exactly like Algorithm 1. Algorithm 2 terminates once the window size is halved again and becomes $\lfloor \frac{1}{2} \rfloor = 0$. This is because (Line 3) requires $k \geq 1$. Notice that Axioms 3 and 10 are discarded during this last step and the output ontology $J = \{1, 2, 11, 12\}$ is a justification for $\mathcal{O} \models A \sqsubseteq B$ as shown earlier. □

Example 3.2 shows the performance advantage that may be gained when using Algorithm 2 over Algorithm 1. Eleven entailment checks are required when using Algorithm 2 to compute a single justification in this example, whereas Algorithm 1 would require fifteen entailment checks. The four entailment checks saved in this example may not sound significant but in practice, when applied to much larger ontologies, the gains become more substantial.

There are other variations of the sliding window technique. For example, the binary chop (divide and conquer) method [Baader and Suntisrivaraporn, 2008]. This method partitions the ontology into two halves and checks if the entailment is preserved (holds) in one of them. If it does then the other half can be discarded. For details, the reader is pointed to the reference provided. Sliding window and binary chop fall into the general category of axiom pinpointing optimizations called *expansion or contraction techniques*. These techniques essentially expand (add required axioms to) and shrink (remove irrelevant axioms from) the ontology in a controlled manner until a justification remains. Another useful optimization which may be used, as a preprocessing step, to prune irrelevant axioms from the ontology is the technique of *module extraction*.

Module Extraction

One of the driving forces behind the development of module extraction [Konev et al., 2008; Suntisrivaraporn, 2008; Cuenca-Grau et al., 2007; Doran et al., 2007] techniques is the need for ontology reuse [Jimenez-Ruiz et al., 2008]. While developing an ontology the user may want to incorporate terms in the ontology that have already been defined (to his/her satisfaction) in other ontologies. Instead of manually adding these terms and their defining constructs into the ontology, module extraction provides an automated method to extract the relevant axioms (module) from these "foreign" ontologies. This set of sentences can then be easily imported into the user's ontology. Module extraction approaches are generally divided into two categories, i.e., *syntactic* and *semantic* approaches [Konev et al., 2008].

To perform module extraction, one requires an ontology from which to extract the module and an *input signature*. The ontology from which the module is extracted is known as the "foreign" ontology. A *signature* with regards to module extraction is a list of *terms* (concept, role and individual names). The signature of an axiom α, for example, is the list of all terms in α which we denote by $\mathsf{Sig}(\alpha)$. Similarly, the signature of an ontology $\mathcal{O}$ is the list of all terms in $\mathcal{O}$ denoted by $\mathsf{Sig}(\mathcal{O})$. Therefore the specified input signature for module extraction is a list of terms whose meaning is to be fully captured in the module which is extracted from the foreign ontology.

This signature influences which axioms from the foreign ontology are to form part of the module. Furthermore, the axioms in the extracted module have to be sufficient to

fully define the terms in the input signature. For example, if the ontology engineers are developing an ontology describing the functioning of the human brain, they may discover that the SNOMED CT medical terminology defines this functioning as they would like. To capture this information in their ontology, a naïve approach would be to import the entire SNOMED terminology into their ontology. However, SNOMED contains many thousands of axioms (a lot of them not pertaining to the human brain) and this would make the user's ontology unnecessarily large.

Module extraction is a solution to this problem because it allows one to extract a *subset* of the terminology which is *required* for the specific application. In the example, the ontology engineer could use a module extraction technique with SNOMED being the foreign ontology and the input signature being $\{\mathsf{Brain}\}$. This would provide the engineer with a much smaller subset (module) of SNOMED which fully captures the meaning of the Brain term and all related terms. Ideally a module should be as small as possible while retaining this property of fully capturing the meaning of the input terms. There are several different types of modules in the literature. The definitions and methods for extracting these modules are not given here. The reader is referred to the provided references for details on these issues.

The question arises: how is module extraction useful in the context of axiom pinpointing? The answer can be found in notions such as conservative extensions [Cuenca-Grau et al., 2007] and reachability-based modules [Suntisrivaraporn, 2008]. These kinds of modules *preserve entailment* and *justifications*. This can be explained as follows. Suppose we are given an entailment $\mathcal{O} \models \alpha$ and the task is to compute a justification for this entailment. One can use module extraction to extract only those axioms from $\mathcal{O}$ which pertain to α. If the input signature is $\mathsf{Sig}(\alpha)$ and foreign ontology is $\mathcal{O}$, the module $\mathcal{M}$ for $\mathsf{Sig}(\alpha)$ with respect to $\mathcal{O}$ is such that $\mathcal{M} \models \alpha$ if and only if $\mathcal{O} \models \alpha$ [Suntisrivaraporn, 2008]. $\mathcal{M}$ preserves entailment of α and $\mathcal{M}$ is called an α-module [Suntisrivaraporn, 2008] for $\mathcal{O}$.

From the preservation of entailment property of a module we know that the module preserves at least one justification for the entailment (Otherwise the entailment would not hold in the context of the module). However certain modules also preserve *all* justifications [Suntisrivaraporn et al., 2008]. This means that, every justification J for the entailment is such that $J \subseteq \mathcal{M}$. Note that $\mathcal{M}$ is not necessarily a *justification* for the entailment but it serves as a relatively small subset of the original ontology in which to start searching for the justifications. Those axioms which do not pertain to the input signature are not included in the module and depending on the number of these axioms one can save heavily on the number of entailment checks required to find the justifications. Module extraction is thus very useful as a preprocessing step (and important optimization) for axiom pinpointing.

Figure 3.1 gives a visual representation of the properties of preservation of entailment and justifications in certain modules.

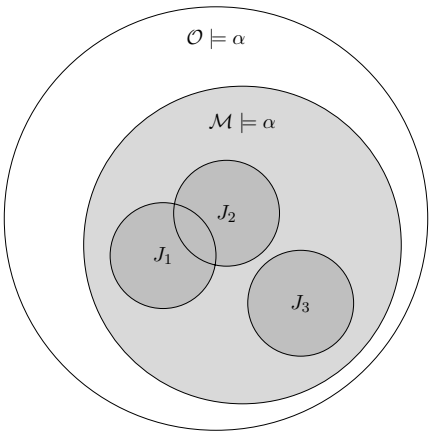

Figure 3.1: α-module $\mathcal{M}$ for $\mathcal{O}$ preserves entailment and justifications

J_1, J_2 and J_3 represent sets of axioms which are the justifications for the entailment $\mathcal{O} \models \alpha$. The larger unshaded circle represents the set of axioms which constitutes the ontology $\mathcal{O}$. The smaller shaded circle represents the set of axioms which constitutes the α-module, $\mathcal{M}$, for $\mathcal{O}$.

Expansion

In addition to module extraction, expansion [Horridge et al., 2009a] is another useful technique for optimizing Black-box algorithms for computing justifications. Consider the task of computing a justification for an entailment $\mathcal{O} \models \alpha$, where α is an axiom. Expansion works by creating a new ontology (with no axioms), $\mathcal{O}'$. After this, subsets of axioms are carefully selected from $\mathcal{O}$ to add to $\mathcal{O}'$ until $\mathcal{O}' \models \alpha$. From this description, expansion seems to extract a kind of *module* for the entailment similar to that of module extraction. But a key difference is that the "module" which the expansion procedure extracts does not necessarily preserve *all* justifications. However, it is clear that expansion arrives at a module which preserves the entailment and therefore at least *one* justification. Expansion is therefore suitable as an optimization for computing a single justification.

During the expansion procedure, the addition of a *set* of axioms, as opposed to a single axiom at a time, helps to save on the number of entailment tests. The criteria for selecting the axioms to add are based on the entailment in question. This usually means that the structure of the axioms from $\mathcal{O}$ are studied to determine which of these are related to the axiom in the entailment. For example if the entailment in question is $\mathcal{O} \models A \sqsubseteq B$ then one may start by adding the CDAs for A and B.

This particular expansion technique can be illustrated by means of the following example. Consider the ontology in Example 3.2. A consequence of this ontology is $A \sqsubseteq B$. To perform expansion, one begins by creating an empty ontology (ontology with no axioms), say, $\mathcal{O}'$. Now one can add the CDAs for A and B to $\mathcal{O}'$. In the example ontology, Axioms 1 and 2 are CDAs for A and Axiom 11 is a CDA for B. Now the question is asked, is it the case that $\mathcal{O}' \cup \{1, 2, 11\} \models A \sqsubseteq B$? The answer is 'no' in this case. How does one proceed from here? One option is to add the CDAs for the *remaining* concept names from Axioms

1, 2 and 11 to $\mathcal{O}'$. This continues until $\mathcal{O}' \models A \sqsubseteq B$.

The remaining concept name in Axiom 1 is C. The only CDA for C in the example ontology is Axiom 12. For Axiom 2, the remaining concept name is E, however, there are no CDAs for E in the example ontology. For Axiom 11, the remaining concept name is D and the only CDA for D in the example ontology is Axiom 4. These CDAs are added to $\mathcal{O}'$ and the resulting entailment $\mathcal{O}' \cup \{1, 2, 4, 11, 12\} \models A \sqsubseteq B$ now holds. Therefore, the resulting module from expansion is the ontology $\mathcal{O}' = \{1, 2, 4, 11, 12\}$.

The ontology in Example 3.2 has a total of fifteen axioms. After expansion is executed (at the cost of two entailment checks), the resulting ontology contains only five axioms. This latter ontology is much smaller and is thus a better starting ontology from which to pinpoint a justification. The justification for the consequence $A \sqsubseteq B$, in the example ontology, is $\{1, 2, 11, 12\}$. This justification, incidently, contains only one axiom fewer than the expansion module $\mathcal{O}' = \{1, 2, 4, 11, 12\}$.

Note that there are various other techniques (apart from the CDA technique described here) that one could use for selecting the axioms to add during expansion. In the next section, the module extraction and expansion optimizations discussed are put together to present a typical optimized Black-box algorithm for computing justifications. This algorithm serves as a guideline for what existing Black-box algorithms look like.

Optimized Black-box Algorithm

Algorithm 2 is an optimized version of Algorithm 1. In fact, executing Algorithm 2 with a starting window size of one, is equivalent to executing Algorithm 1. The major difference between the algorithms is that Algorithm 2 (potentially) prunes the ontology much faster. It starts with a coarse approach to pruning, removing bigger subsets of the ontology at a time and reducing the size of these subsets (by reducing the window size) until it reaches the finest-grained subset being a single axiom. Although Algorithm 2 is much more efficient than Algorithm 1, there are various other optimizations one could use to make axiom pinpointing faster. For example, prior to executing Algorithm 2, a common technique to trim away some irrelevant axioms from the ontology, is to use module extraction and expansion. The following algorithm combines these optimizations with Algorithm 2 to resemble a typical optimized Black-box approach for axiom pinpointing.

Algorithm 3: *fastContract+* (Single justification — Optimized)

Input: Ontology $\mathcal{O}$, axiom α such that $\mathcal{O} \models \alpha$ and window size $k \geq 1$
Output: Justification $\mathcal{J}$ for $\mathcal{O} \models \alpha$

```
1 J, M := ∅;
2 M := extractModule(Sig(α), O);
3 while J ⊭ α do
4     J := J ∪ selectSubset(M);
5 J := fastContract(J, α, k);
6 return J;
```

In Algorithm 3, module extraction and expansion are used as preprocessing steps to trim away axioms which are unnecessary to the entailment $\mathcal{O} \models \alpha$. First, an α-module for $\mathcal{O}$, preserving the justifications for $\mathcal{O} \models \alpha$, is extracted using any appropriate module

extraction technique. The module $\mathcal{M}$ is then passed to an expansion technique (Lines 3 and 4). This expansion technique uses a procedure *selectSubset* to incrementally add axioms which are relevant to the entailment, to $\mathcal{J}$ until $\mathcal{J} \models \alpha$. $\mathcal{J}$ is then passed to the *fastContract* procedure (Algorithm 2) to pinpoint a justification for the entailment.

Algorithm 3 provides a method for computing a single justification for a single entailment using the module extraction and expansion optimizations that have been discussed in the previous sections. Algorithm 3 (*fastContract+*), also forms the base procedure that is used in a method for computing *all* justifications for an entailment. This method is discussed in Section 3.3.2.

Inconsistent Ontologies

In an inconsistent ontology $\mathcal{O}$, there is no interpretation which satisfies any of the axioms in $\mathcal{O}$. Therefore *every* axiom follows from the ontology. The reason is that any ontology $\mathcal{O}$ entails some axiom α if and only if every model of $\mathcal{O}$ satisfies α and if $\mathcal{O}$ is inconsistent, then it means that $\mathcal{O}$ has *no* models and hence all models of $\mathcal{O}$ satisfy α and hence *any* other axiom. It is therefore not very meaningful to reason with an inconsistent ontology and therefore it is treated as an erroneous situation in the ontology.

Inconsistency in ontologies is usually caused by ABox assertions that are contradictory with TBox statements (e.g., $C(a)$ and $C \equiv \bot$) in the ontology or by some ABox assertions which conflict with each other (e.g., $R(a, b)$, $\forall R.C(a)$ and $\neg C(b)$).

It was mentioned in Section 3.2 that the notion of justification can be extended to inconsistent ontologies. Essentially, the same general strategy for axiom pinpointing for unsatisfiable concepts applies to axiom pinpointing for inconsistent ontologies. However, any additional optimizations to this strategy, which analyze the signature of the erroneous entailment in order to extract modules (such as module extraction and certain expansion techniques) are not applicable to inconsistent ontologies. This is because, if an ontology is inconsistent, the axiom $\top \sqsubseteq \bot$ is entailed by the ontology and the signature of this axiom (i.e., $\{\top, \bot\}$) is not conducive for extracting a useful module for computing justifications for the inconsistency.

Therefore to summarize how justifications can be computed for an inconsistent ontology, the expand/shrink strategy, used for a general entailment, may be used again to continually remove axioms from the ontology which are not relevant to the inconsistency until only a justification for the inconsistency remains.

3.3.2 Computing All Justifications

We have seen that entailments may have more than one justification. Each of these justifications is a set of sentences in the ontology that fully supports the given entailment. That is, each of these sets of sentences is a sufficient condition for the entailment to hold. Therefore, for repair purposes, if one wants to get rid of an erroneous entailment it is useful as a starting point to know *all* justifications for that entailment.

This set of all justifications serves as the minimal (and at the same time complete) reason as to why the entailment holds in the ontology. Hence this information is also sufficient for use in devising a repair strategy to remove the entailment in question. This section discusses

the widely used approach for pinpointing all justifications for a given entailment [Kalyanpur, 2006] using a variant of Reiter's hitting set algorithm [Reiter, 1987]. This method assumes that one already has a procedure for computing a *single* justification (an example of such a procedure is given in Section 3.3.1 as Algorithm 3). We now explain how this method can be used to compute all justifications for some entailment (or unsatisfiable concept).

Hitting Set Algorithm

In his theory of diagnosis [Reiter, 1987], Reiter refers to a system consisting of a set of first-order sentences (*system description*) and a set of components $\mathcal{U}$. These components may be "plugged in" to the system description to elicit some desired behaviour from the overall system. However, certain subsets of $\mathcal{U}$, if added to the system description, may cause a fault in the system. Such component subsets are called *conflict sets*. Given a collection of conflict sets $\mathcal{C}$ for a certain system, Reiter's algorithm identifies all *minimal hitting sets* (smallest subsets of $\mathcal{U}$ which intersect every element in $\mathcal{C}$) for $\mathcal{C}$. These terms are recapped as follows:

Universal set: The universal set $\mathcal{U}$ is the set of all components available in a system.

Conflict set: A set $\mathcal{S} \subseteq \mathcal{U}$ that is responsible for a fault in the system.

Hitting set: A hitting set is some subset of the component set $\mathcal{U}$ that contains at least one element of each conflict set in the given collection of conflict sets. More formally, a set $\mathcal{H} \subseteq \bigcup_{\mathcal{S} \in \mathcal{C}} \mathcal{S}$ is a hitting set for $\mathcal{C}$ if and only if $\mathcal{H} \cap \mathcal{S} \neq \emptyset$ for every $\mathcal{S} \in \mathcal{C}$. $\mathcal{H}$ is a *minimal hitting set* for $\mathcal{C}$ if and only if there is no proper subset of $\mathcal{H}$ which is also a hitting set for $\mathcal{C}$.

HS-tree: Reiter's algorithm constructs a Hitting-set tree (HS-tree) for the collection of conflict sets $\mathcal{C}$. This tree is a set of nodes $\mathcal{V}$ and edges $\mathcal{E}$. Each node $v \in \mathcal{V}$ has a label $v.label \in \mathcal{C}$, i.e., $v.label$ is a conflict set and each edge $e \in \mathcal{E}$ has label $e.label \in \bigcup_{\mathcal{S} \in \mathcal{C}} \mathcal{S}$, i.e., $e.label$ is an element of some $\mathcal{S} \in \mathcal{C}$. A function $P(v)$, the path function, returns the set of edge labels on the path from the *root* node to a node v.

The construction of an HS-tree T is carried out in a breadth-first fashion, according to the following rules:

- Firstly, a root node v_{root} for T is generated. v_{root} is labelled with an arbitrary conflict set $\mathcal{S} \in \mathcal{C}$.
- If a node v is labelled by a set $\mathcal{S} \in \mathcal{C}$ then for each $\varphi \in \mathcal{S}$ a successor node v_φ is attached to v via an edge e_φ labelled with φ.
- Each successor node v_φ is then labelled with a set $\mathcal{S}' \in \mathcal{C}$ such that $\mathcal{S}' \cap P(v_\varphi) = \emptyset$. If no such $\mathcal{S}'$ exists, v_φ is labelled with a '$\checkmark$'. A node labelled by '$\checkmark$' indicates a *terminating* node (leaf) which has no successors.

Each set $P(v)$ such that v is labelled with '$\checkmark$' is a hitting set for $\mathcal{C}$. Therefore the collection $\mathcal{L}$ of all hitting sets for $\mathcal{C}$ (paths $P(v)$), found in T, contains all *minimal* hitting sets for $\mathcal{C}$ [Reiter, 1987, Theorem 4.8]. A minimal hitting set in the HS-tree corresponds to a set $P(v)$ for a terminating node v such that there is no other terminating node v' in the HS-tree where $P(v') \subset P(v)$. An example illustrating the construction of an HS-tree for the collection $\mathcal{C} = \{\{\text{a, b, d}\}, \{\text{b, d, e}\}, \{\text{c, e}\}, \{\text{f}\}\}$ is given in Figure 3.2.

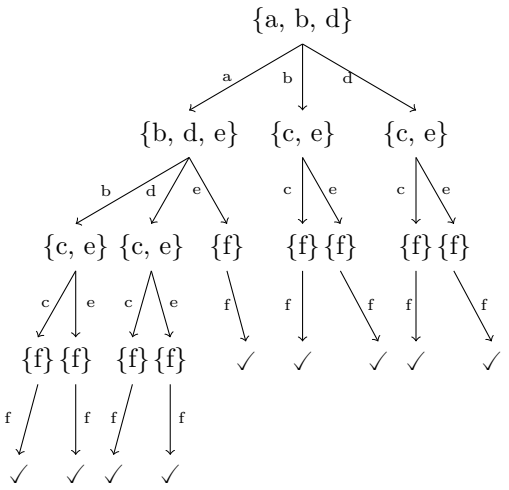

Figure 3.2: HS-tree for collection $\mathcal{C} = \{\{a, b, d\}, \{b, d, e\}, \{c, e\}, \{f\}\}$.

The minimal hitting sets for $\mathcal{C}$ depicted in the HS-tree of Figure 3.2 are {a, e, f}, {b, c, f}, {b, e, f}, {d, c, f} and {d, e, f}. The next section shows the applicability of the theory of Reiter's HS-tree to the problem of computing all justifications for an entailment.

Conflict Sets and Justifications

Reiter's algorithm for computing minimal hitting sets was briefly described above. How can this algorithm be used for computing all justifications for some entailment? We use a special *variant* of Reiter's algorithm to do this [Kalyanpur et al., 2007]. It works as follows. Firstly, let us assume that the universal set $\mathcal{U}$ in Reiter's algorithm corresponds to the set of axioms which constitutes an ontology $\mathcal{O}$ in the variant algorithm. Secondly, one can see the entailment of some axioms (or the unsatisfiability of a concept) as 'faults' in the system. These are caused by one or more sets of axioms in $\mathcal{U}$, i.e., the *justifications* for the entailment. These are therefore, the 'conflict sets' in the variant of Reiter's algorithm.

A crucial difference here is that at the start of Reiter's algorithm we are *given* all the conflict sets in the collection $\mathcal{C}$, whereas at the start of the variant algorithm we have no justifications. In Reiter's algorithm we have a function which selects a conflict set from $\mathcal{C}$ to label the particular node we are constructing. Therefore, in theory, if we have a corresponding function which is able to *generate* a conflict set (justification) in the variant algorithm, then we should be able to construct the HS-tree in its entirety.

Recall that in Section 3.3.1 a method for generating a single justification is given. This function could be used to generate the justifications which we require in the variant algorithm. The variant algorithm constructs a similar tree to the one in Reiter's algorithm, however the focus in Reiter's HS-tree is on finding minimal hitting sets (the paths to terminating nodes), whereas the focus in the variant algorithm is on finding the justifications (the nodes) for the entailment of some axiom or the unsatisfiability of some concept. Therefore the tree which is constructed in the variant algorithm is known as a *justification tree* and not an HS-tree.

Note: We know that Reiter's original algorithm focuses on finding the minimal hitting sets for the *conflict sets* and the variant algorithm focuses on finding the justifications ("conflict sets"). However, the variant algorithm, as a consequence of constructing the justification tree, also identifies the minimal hitting sets for the *justifications* computed in the tree. These minimal hitting sets correspond to the *repairs* for the unsatisfiability or entailment in question and is important for when we discuss repairs in the latter part of this chapter. We now proceed with the description of the variant algorithm.

Justification Tree

A justification tree is analogous to an HS-tree. A justification tree, $T_{\mathcal{J}}$, is a set of nodes $\mathcal{V}_{\mathcal{J}}$ and edges $\mathcal{E}_{\mathcal{J}}$. Each node $j \in \mathcal{V}_{\mathcal{J}}$ has a label $j.label$ which is a justification for the unsatisfiability of a concept, say C, with respect to the ontology $\mathcal{O}$. Each edge $e \in \mathcal{E}_{\mathcal{J}}$ has label $e.label \in \bigcup_{j.label \in T_{\mathcal{J}}} j.label$, i.e., $e.label$ is an axiom of some justification. The function $P(j)$, the path function, returns the set of edge labels on the path from the root node to node j.

To construct a justification tree $T_{\mathcal{J}}$ in a breadth-first fashion, the following rules are applied.

(i) The first step in Reiter's algorithm is to generate a root node, labelled with an arbitrarily chosen conflict set from $\mathcal{C}$. The corresponding first step in the variant algorithm is to generate a root node j_{root} for $T_{\mathcal{J}}$ which is labelled with a *justification* for C. This justification can be generated with respect to the ontology $\mathcal{O}$ using any method for computing a single justification (such as the *fastContract+* procedure in Section 3.3.1).

(ii) If a node j in the justification tree is labelled with a justification J, then for each axiom $\alpha \in J$, a successor node j_α is attached to j via an edge e_α which is labelled with α.

(iii) Each successor node j_α in the justification tree is labelled with a justification J' for C. J' is generated using the same method as in the first rule. However the difference now is that J' is computed with respect to the ontology $\mathcal{O}\backslash P(j_\alpha)$ and *not* $\mathcal{O}$. If however, it turns out that $\mathcal{O}\backslash P(j_\alpha)$ does not contain a justification for C it means that $\mathcal{O}\backslash P(j_\alpha)$ is not C-unsatisfiable and therefore we label j_α with '$\checkmark$' indicating a terminating node with no successors.

The key difference between Reiter's algorithm and the variant algorithm becomes evident in rule (iii). In the justification tree, this rule ensures that for a node j and its successor node j_α, $j.label \neq j_\alpha.label$. In other words, the same justification cannot appear as the label of a parent node as well as its direct children. In fact it is the case that for *any* two nodes which are on the same path in the justification tree, they cannot share the same label (justification). A similar property holds for an HS-tree in Reiter's algorithm. This can be verified from the rules for constructing an HS-tree given in Reiter's work [Reiter, 1987]. Here we prove that this property, as extended for a *justification* tree, holds as well.

Theorem 3.1 *Given two nodes j_1 and j_2 in a justification tree $T_{\mathcal{J}}$, if j_2 is a successor of j_1, that is if $P(j_1) \subset P(j_2)$, then $j_1.label \neq j_2.label$.*

Proof:
Let us assume that Theorem 3.1 is false, i.e., we assume that j_2 is a successor of j_1, i.e., $P(j_1) \subset P(j_2)$, but that $j_1.label = j_2.label$. From the rules of the justification tree construction we know that:

1. $j_1.label$ represents a justification for C with respect to $\mathcal{O}\backslash P(j_1)$ and $j_2.label$ represents a justification for C with respect to $\mathcal{O}\backslash P(j_2)$. (From rule (iii))

2. There exists an axiom $\alpha \in j_1.label$ such that $\alpha \in P(j_2)$. (From assumption $P(j_1) \subset P(j_2)$ and rule (ii))

From statement 2 it follows that:

3. $\alpha \notin \mathcal{O}\backslash P(j_2)$ and this implies that $\alpha \notin j_2.label$ because the justification represented by $j_2.label$ is a subset of $\mathcal{O}\backslash P(j_2)$ by Definition 3.1 of a justification.

Statements 2 and 3 lead to a contradiction with our assumption that if j_2 is a successor of j_1, then $j_1.label = j_2.label$. Therefore our assumption that Theorem 3.1 is false is incorrect. Hence, Theorem 3.1 is indeed true. □

The major difference between the construction of the HS-tree and the justification tree is that the operation/test performed to determine the label of a particular node is different. In an HS-tree one scans the collection $\mathcal{C}$ for a set $\mathcal{S}'$ such that $\mathcal{S}' \cap P(v_\alpha) = \emptyset$. In a justification tree, a justification is generated using any available axiom pinpointing method and the node is labelled with the justification. To ensure that one computes a unique justification for each node in a path, the ontology which is used to pinpoint the justification depends on the path to the node. That is, the ontology is denoted by $\mathcal{O}\backslash P(v_\alpha)$. We now give an example which demonstrates the construction of a justification tree.

Example 3.3 Suppose we have an ontology $\mathcal{O}$ with ten axioms denoted by $1, \ldots, 10$:

Suppose that $\mathcal{O} \models \alpha$ and we would like to compute all justifications for this entailment. Let us assume that the justifications for the entailment are $\{1, 2, 3, 4, 5\}$, $\{1, 3, 4, 6, 7\}$ and $\{1, 3, 4, 8\}$ (of course, before we begin constructing the justification tree we do not know these justifications). Figure 3.3 depicts the justification tree for the entailment and our analysis of its construction follows:

The first step is to generate the root node. The label for the root node should be a justification for the entailment. We use a procedure for finding a single justification (e.g., Algorithm 3) to find the first justification $\{1, 2, 3, 4, 5\}$ which we label the root node with. For each axiom β in the root node label we generate another node v such that the label of v is a justification with respect to ontology $\mathcal{O}\backslash\{\beta\}$. This ensures that the justification $\{1, 2, 3, 4, 5\}$ is not computed again because $\beta \in \{1, 2, 3, 4, 5\}$ is removed from the ontology. The algorithm continues in this fashion until all leaf nodes are labelled with '✓'. When the algorithm terminates, each unique node label represents a justification for the entailment. Furthermore the set of all node labels in the tree represents *all* justifications for the entailment in question [Kalyanpur, 2006, Theorem 4]. Therefore, in Figure 3.3, the justifications for the entailment are $\{1, 2, 3, 4, 5\}$, $\{1, 3, 4, 6, 7\}$ and $\{1, 3, 4, 8\}$. □

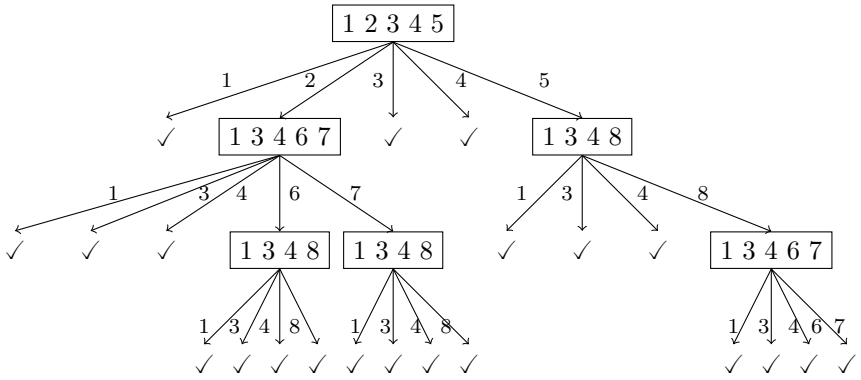

Figure 3.3: Justification Tree for $\mathcal{O} \models \alpha$ in Example 3.3.

Justification tree minimization

One of the advantages of using Reiter's algorithm as a framework for computing the justifications is that the optimizations for pruning the HS-tree in Reiter's algorithm can be applied to pruning the justification-tree. In Reiter's original algorithm, these optimizations "close" nodes in a particular path of the tree much earlier. Specifically this is done when it is discovered that a minimal hitting set cannot be found by extending this path any further. Using this optimization in a justification tree we can avoid computing some duplicate justifications and hence node labels in the tree. The optimizations for Reiter's original algorithm for constructing a HS-tree, that are *relevant and applicable* to the variant algorithm for constructing a justification tree, are discussed here. Readers interested in other optimizations for Reiter's original algorithm may consult his work [Reiter, 1987].

(i) If there is a node v in the justification tree and another node v' is about to be generated such that $P(v) = P(v')$ then v' is closed and labelled with 'X' (similar to '✓', both nodes cannot have any successors). The reason for this is that we are going to generate the same justifications in the sub-tree under v' as those in the sub-tree under v (because $\mathcal{O} \backslash P(v) = \mathcal{O} \backslash P(v')$) so we close the node ending the path here.

(ii) Similar to (i), another scenario in which we have a path $P(v)$ in the justification tree such that v is a leaf node ('✓'). If we are about to generate another node v' such that $P(v') \subset P(v)$ then we close v' with 'X' for the same reason as (i): the justifications which would have been generated under v' already appear in the sub-tree under v.

The optimizations discussed in (i) and (ii) are known as *early path termination* optimizations in Reiter's original algorithm. This concludes the discussion on Black-box axiom pinpointing methods. In the next section, the existing Glass-box approaches to computing justifications are briefly discussed.

3.4 Computing Justifications: Glass-box Approach

Existing Glass-box methods for computing justifications [Schlobach and Cornet, 2003; Kalyanpur et al., 2005b; Meyer et al., 2006; Lam et al., 2008] exploit the tableau algorithms for satisfiability used in DL reasoners to identify the root cause for the unsatisfiability

of the concept or unwanted consequence in the ontology. The "glass-box" name for this approach comes from the fact that it requires awareness of the inner-workings (algorithms) of the reasoner.

Recall from Chapter 2 that the tableau algorithm for determining concept satisfiability for a concept C, works by trying to construct an interpretation for C in which C is non-empty. It does this by repeatedly applying a set of transformation rules to some assertions about C in the ontology. This process builds what is known as a *completion graph* with each branch of the graph representing a different interpretation for C. Each transformation adds new assertions to the completion graph and these additional assertions may result in a contradiction or *clash*. An example of a clash can be found in the two assertions $C(x)$ and $\neg C(x)$. This is clearly a logical contradiction since an individual x cannot belong to a concept C and its complement $\neg C$. When a clash is detected in the completion graph or no more transformation rules apply in the branch, the algorithm stops constructing that particular branch and backtracks to construct another interpretation. This is because, the clash indicates that an interpretation for C in which C is non-empty cannot be found in that particular branch. The algorithm terminates when the transformation rules no longer apply in the completion graph. C is satisfiable if an interpretation for C was found in which C is non-empty. C is unsatisfiable if no such interpretation was found.

The Glass-box justification methods use the same algorithm discussed above. The difference is that C is already known to be unsatisfiable and the focus is on locating the root cause of the unsatisfiability. Therefore these algorithms keep track of the clashes in the completion graph. The clashes (contradicting assertions) indicate the causes of the unsatisfiability. After clashes are identified, a process of *tracing* [Kalyanpur, 2006] is performed to find the axioms in the ontology which are responsible for the clashes. These axioms ultimately form part of the justifications for the unsatisfiability of C.

A very brief discussion on the Glass-box approach for axiom pinpointing was given in this section. Since the focus in this dissertation is on Black-box approaches, no further details are given on the Glass-box methods and the interested reader should consult the references provided for details. The next section gives a similar brief overview on a third approach to axiom pinpointing which combines ideas from both Black-box and Glass-box approaches. This is called the Hybrid approach to axiom pinpointing. Hybrid methods are not as common as Black-box and Glass-box methods but a few of these approaches are nevertheless introduced in the next section.

3.5 Computing Justifications: Hybrid Approach

We have seen that Black-box axiom pinpointing methods use the reasoner solely as an "oracle" to determine if an entailment holds or not. Glass-box methods, on the other hand, require a modified version of the reasoner's tableau algorithm to determine the justifications for the entailment. Hybrid methods fit somewhere in between these two approaches.

One particular hybrid method uses an *incremental* algorithm called *singleMUPSInc* to compute a single justification for an unsatisfiable concept [Kremen and Kouba, 2009]. This approach captures the current state (partially expanded completion graphs and axiom assertions used for expanding the graph thus far) of the reasoner, hence the reasoner is

not used purely as a "black-box" to test entailment and at the same time the tableau algorithm for the reasoner is not modified in any way, therefore it is categorized as a Hybrid method. There is also a similar incremental algorithm for computing *all* the justifications or MUPSes for an unsatisfiable concept [de la Banda et al., 2003].

Finally, a partially Glass-box method exists for extracting relatively small subsets of the original ontology which preserves the entailment in question [Kalyanpur et al., 2005b]. The method is used as a preprocessing step to a sound and complete Black-box axiom pinpointing method to compute the actual justifications and hence it is classed as a *partially* Glass-box or Hybrid method.

3.6 Fine-grained Justifications

Recall that a justification for some entailment is a minimal subset of an ontology which is sufficient for the entailment to hold. However, if one assumes a finer level of granularity and we examine the axioms of a particular justification, we may find that only certain *parts* of the axioms in the justification contribute to the entailment and not the *whole* axioms. An example of this intuition follows:

Example 3.4 Consider the following ontology:

$$\mathcal{O} = \left\{ \begin{array}{l} 1.\ B \sqsubseteq D, \\ 2.\ A \sqsubseteq \neg D, \\ 3.\ A \sqsubseteq B \sqcap \exists R.C \end{array} \right\}$$

A consequence of $\mathcal{O}$ is the unsatisfiability of the concept A. It is clear that all the axioms in $\mathcal{O}$, i.e., {1, 2, 3}, form a justification for $\mathcal{O} \models A \sqsubseteq \bot$. However the conjunct $\exists R.C$ in Axiom 3 is not relevant to the entailment. Therefore, even if one *weakens* Axiom 3 to exclude the irrelevant part, the entailment $\mathcal{O} \models A \sqsubseteq \bot$ still holds. □

The following is a definition for what is meant when we say one axiom is weaker than another:

Definition 3.2 (Axiom weakness)
An axiom α' is *weaker* than an axiom α if and only if $\{\alpha\} \models \alpha'$ and $\{\alpha'\} \not\models \alpha$.

Therefore if one weakens Axiom 3 in Example 3.4 to exclude the conjunct $\exists R.C$ from the ontology, the justification for the A-unsatisfiability of $\mathcal{O}$ becomes $\{B \sqsubseteq D,\ A \sqsubseteq \neg D,\ A \sqsubseteq B\}$ which is a finer-grained and more concise justification. However, observe that in this scenario, by weakening Axiom 3, we are eliminating extra information from the ontology, i.e., the axiom $A \sqsubseteq \exists R.C$ (even though it does not contribute to the unsatisfiability). This behaviour is not desired.

The methods for computing fine-grained justifications [Kalyanpur et al., 2006a] get around this problem by *rewriting* each complex axiom in the ontology into multiple simpler axioms which together capture the meaning of the original complex axiom. In Example 3.4, the

axiom re-writing method would result in the ontology $\mathcal{O} = \{B \sqsubseteq D,\ A \sqsubseteq \neg D,\ A \sqsubseteq B,\ A \sqsubseteq \exists R.C\}$. It is clear that using any of the axiom pinpointing algorithms introduced in Section 3.3 to Section 3.5 on the new ontology will again yield the same justification $\{B \sqsubseteq D,\ A \sqsubseteq \neg D,\ A \sqsubseteq B\}$ for the A-unsatisfiability of $\mathcal{O}$. This kind of justification is called a *fine-grained* or *precise* justification for the unsatisfiability because it does not contain any superfluous information which is irrelevant to the entailment.

There are various motivations for the investigation into fine-grained justifications but the foremost reason is for repair purposes. A goal of ontology repair is to modify the ontology as little as possible while eliminating the undesired entailment. It is clear that using fine-grained justifications for repair would potentially yield smaller modifications to the ontology to resolve the error.

There are various other approaches to determine fine-grained justifications for entailments in an ontology [Horridge et al., 2008; Lam et al., 2008; Schlobach and Cornet, 2003]. In particular the work by Horridge et al. [2008] on *laconic* and precise justifications was the first to give a formal definition for what they mean by *parts* of axioms. They also present an algorithm with performance results for computing these kinds of justifications. There are also approaches for computing minimal fine-grained *repairs* of axioms in other (non-DL) formalisms [Varzinczak, 2008]. There is also some related work on *lemmatising* justifications [Horridge et al., 2009b], which augments justifications with information that makes them easier to understand by users. The interested reader is pointed to the references provided for more details.

The general classes of methods for computing justifications have been discussed in Sections 3.3 to 3.5. In the explanation phase of ontology repair the explanations can also be augmented with various types of information to make them more understandable to the ontology engineer. In existing approaches, the explanation phase forms the core of the ontology debugging and repair process. However, while explanations (and hence justifications) help with understanding why the erroneous consequences hold in the ontology, one still has to develop a strategy to correct the errors using this information. The next section presents some commonly accepted strategies for resolving errors using the information provided by explanations.

3.7 Error Resolution

This section examines the existing approaches to correcting ontology errors using their justifications. First, the simple case of a *single* error in the ontology is considered. The established notion of a *repair* [Kalyanpur, 2006] for a single ontology error is defined and the different methods for computing such repairs is then discussed. Following this, we discuss an existing method for computing more *appropriate* repairs for an entailment, taking into account the effect that each repair has on the ontology. Finally, the issue of resolving *multiple* errors in an ontology, is introduced. The main approaches for optimally resolving such a problem are presented.

3.7.1 Resolving a Single Error

The most common approach to resolving an ontology error is to use the information presented in its justifications to identify the repairs. The notion of repair as applied to a single entailment is defined as follows:

Definition 3.3 (Repair)
Let $\mathcal{O}$ be an ontology and α an axiom such that $\mathcal{O} \models \alpha$. A set of axioms $\mathcal{O}' \subseteq \mathcal{O}$ is a *repair* for α with respect to $\mathcal{O}$ if and only if $\mathcal{O} \backslash \mathcal{O}' \not\models \alpha$ and there exists no $\mathcal{O}'' \subset \mathcal{O}'$ such that $\mathcal{O} \backslash \mathcal{O}'' \not\models \alpha$.

It is clear from Definition 3.3 that a repair for an error in an ontology $\mathcal{O}$ is a minimal subset of $\mathcal{O}$ such that when it is removed from $\mathcal{O}$, causes the error to be eliminated. This means that the resulting ontology after the repair is applied, say $\mathcal{O}_s$, is a maximal subset of the original ontology in which the error does *not* hold. This ontology is called an *ontology solution*. We give a definition for this principle here:

Definition 3.4 (Ontology solution)
Let $\mathcal{O}$ be an ontology and α an axiom such that $\mathcal{O} \models \alpha$. If $\mathcal{O}'$ is a repair for $\mathcal{O} \models \alpha$ then $\mathcal{O}_s = \mathcal{O} \backslash \mathcal{O}'$ is an *ontology solution* for $\mathcal{O} \models \alpha$.

An important note with regards to Definition 3.3 of a repair is that it is on the "whole axiom" level. This means that an applied repair translates to the removal of entire axioms from the ontology. It does not include axiom modification/rewrites. However, this notion of repair is fully compatible with the *explanation* methods which perform these modifications and axiom rewrites (discussed in Section 3.6). This is because, in general, the approach used by most of these methods is to rewrite each complex axiom in the ontology into multiple simpler axioms which capture the same meaning as the complex axiom. Justifications are then computed using standard methods with respect to the simpler axioms in the ontology and as a result they may be larger (contain more axioms) but still resemble "regular" justifications.

Another important aspect of repairs is the desired property of *minimality*. This means that if we are given an erroneous entailment $\mathcal{O} \models \alpha$, we would like to ensure that the chosen repair for $\mathcal{O} \models \alpha$ has a minimal impact on the other consequences of the ontology. We elaborate on this later in this section.

Identifying a Single Repair

The most basic way to eliminate an unwanted entailment in the ontology is to choose one axiom from each of its justifications and remove these from the ontology. This is the case because we recall that each justification for an entailment is a (minimal) sufficient condition for the entailment to hold. Therefore, if we remove an axiom in each justification from the ontology, we are effectively nullifying each sufficient condition for the entailment to hold. Thus the entailment is eliminated.

In the approach described above (which we call the *naïve approach* to repair), the axioms are selected in a non-deterministic way from each justification and the final axiom set, say $\mathcal{O}'$, is removed from the ontology to eliminate the unwanted entailment. An important

characteristic of this approach is that it does not guarantee that $\mathcal{O}'$ is a repair for the entailment according to Definition 3.3. We illustrate this in the following example.

Example 3.5 Suppose that we have an ontology $\mathcal{O}$ which is C-unsatisfiable. Let us assume that the justifications for the C-unsatisfiability of $\mathcal{O}$ are depicted in the collection: $\{\{1, 2, 5\}, \{3, 7\}, \{1, 6, 7\}\}$. If we use the naïve approach for repair, we may choose Axiom 1 from the first justification, Axiom 3 from the second and Axiom 6 from the last. Observe that Axiom 1 appears in the first and last justifications so by removing Axiom 1 from the ontology we effectively "nullify" the first and last justifications so there is no need to select the Axiom 6 from the last justification as well. Therefore our so-called "repair" for C ($\{1$, 3, 6$\}$) is not actually a repair because we know that $\{1, 3\}$ will also eliminate the unwanted entailment and it is a subset of $\{1, 3, 6\}$. In our example, $\{1, 3\}$ and $\{5, 7\}$ are two valid repairs for the unsatisfiability of C. □

The above scenario illustrates the main drawback of the naïve approach which is that a repair according to Definition 3.3 is not guaranteed to be computed. However, even if by chance, a valid repair is chosen, there are other drawbacks to the naïve approach such as the fact that it only computes *one* repair for the unwanted entailment (while there may be more than one). The importance of knowing *all* the repairs for some unwanted entailment is illustrated with another example.

Example 3.6 Suppose we are given an ontology $\mathcal{O}$ and two axioms α and β which follow from $\mathcal{O}$. Let us assume that it is decided by the ontology engineer that α is an *undesired* consequence and β is an important *desired* consequence of $\mathcal{O}$. To remove α, we compute its justifications and thereafter using a naïve approach we happen to compute a single valid repair for α, say $\mathcal{O}'$. However, if we apply $\mathcal{O}'$ to the ontology we may find that β is removed along with α! Therefore, using this naïve approach, the ontology engineer would be oblivious to the fact that there may be a more appropriate repair $\mathcal{O}''$ for α which does *not* remove β as well. □

Therefore, each repair for some unwanted entailment may have a unique effect on the other entailments in the ontology and one has to take this into account when computing and selecting repairs to apply to the ontology.

In general, in the current tools for ontology development, the focus remains on presenting the *explanations* and not the *repairs* for errors to the ontology engineer. Therefore it is left up to the ontology engineer to figure out different repair strategies based on this information. The result is that the naïve approach to repair is often used as a "quick fix" to get rid of the unwanted entailment without regarding the consequences that this has on the ontology. It is clear then that methods are needed to present *all* the alternative repairs for the entailment to the ontology engineer so that he/she can make an informed decision on which repair to use. A few of these procedures are now discussed.

Identifying All Repairs

Once all justifications for an entailment are known, there are two suggested ways to identify all the repairs for the unwanted entailment. The first way assumes that the variant of the hitting set algorithm (also discussed in Section 3.3.2) was used to compute all the *justifications*. If this is the case, then the repairs are *automatically* identified during the

construction of the justification tree in the variant algorithm. This is a very useful property indeed.

We illustrate this property by noting that Reiter's original hitting set algorithm (discussed in Section 3.3.2) identifies all minimal hitting sets for the conflict sets in a collection. In the variant algorithm, the conflict sets correspond to justifications (sets of axioms) and we focus on identifying these justifications. However, because the justification tree is constructed in an analogous way to Reiter's HS-tree, the result is that all the minimal hitting sets for the *justifications* in the justification tree are also computed [Kalyanpur, 2006]. We also know that the set of all justifications in the *justification tree* corresponds to all justifications for the entailment under consideration.

The fact that all minimal hitting sets for the justifications in the justification tree correspond to all repairs for the entailment under consideration, follows from Definition 3.3 and the following properties [Kalyanpur, 2006, Theorem 3]:

- Given an ontology $\mathcal{O}$, an axiom α such that $\mathcal{O} \models \alpha$ and a set $\mathcal{H} \subset \mathcal{O}$, then $\mathcal{O} \backslash \mathcal{H} \not\models \alpha$ if and only if $\mathcal{H}$ is a hitting set for all justifications for $\mathcal{O} \models \alpha$.
- $\mathcal{H}$ is a minimal hitting set for all justifications for $\mathcal{O} \models \alpha$ if and only if there is no $\mathcal{H}' \subset \mathcal{H}$ such that $\mathcal{O} \backslash \mathcal{H}' \not\models \alpha$.

Figures 3.3 and 3.4 illustrate the above-mentioned property of the justification tree.

The second way for identifying all repairs considers the scenario where, the type of algorithm used to compute all the justifications, is not similar to the hitting set variant algorithm. Reiter's *original* hitting set algorithm may be used in this case to compute *all* repairs for the entailment. The original algorithm can be used because we now have all conflict sets (justifications) and hence the procedure for constructing the HS-tree is exactly the same. We give an example illustrating this.

Returning to Example 3.5, where the justifications for the unsatisfiability of the concept C are collected in $\mathcal{J} = \{\{1, 2, 5\}, \{3, 7\}, \{1, 6, 7\}\}$. Reiter's algorithm can be used to compute all minimal (with respect to set inclusion) sets r such that r intersects each justification in the list. These minimal hitting sets as they are known in Reiter's algorithm correspond to the *repairs* for the unsatisfiable concept C in this context. The HS-tree for the collection $\mathcal{J}$ is given in Figure 3.4.

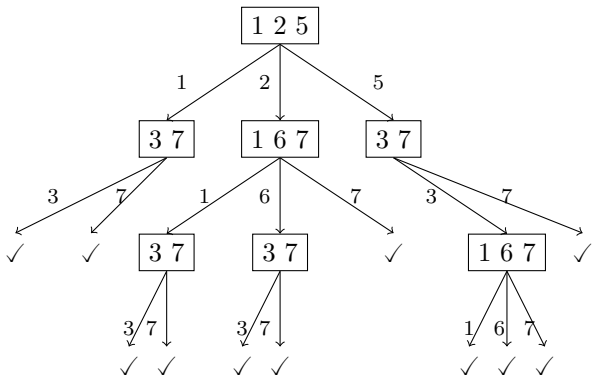

Figure 3.4: Justification Tree for unsatisfiability of C in Example 3.5

The repairs, $\mathcal{R} = \{\{1, 3\}, \{1, 7\}, \{2, 3, 6\}, \{2, 7\}, \{5, 3, 6\}, \{5, 7\}\}$, for the unsatisfiable concept C correspond to the minimal hitting sets contained in the justification tree.

Axiom Ranking Approach:

The penultimate step to repairing an error in the ontology is to select an *appropriate* repair to apply from the computed alternatives. In general, this step is left to the ontology engineer and domain expert who may decide on criteria to identify which repairs are more appropriate for the ontology. However, Kalyanpur et al. [2006b] devised an approach which, prior to computing the repairs for the unsatisfiable concept, *ranks* the axioms appearing in its justifications according to their importance in the ontology. The lower ranked axioms are those that can be removed if needed for repair while the higher ranked axioms are more critical and are thus not modified or removed for repair purposes.

Kalyanpur then presents a method (based on a modified version of Reiter's hitting set algorithm) for generating a list of repairs for the unsatisfiable concept which takes into account the computed axiom rankings. The result is that this method generates repairs based on the effects that each repair has on the ontology and therefore aids the ontology engineer in identifying more intelligent solutions to eliminating the unsatisfiable concept.

The heuristics for determining the rankings for the axioms as described above, are *frequency*, *semantic relevance*, *syntactic relevance* and *test cases*. These heuristics are described as follows:

If ontology $\mathcal{O}$ has a set of unsatisfiable concepts $\mathcal{C}$ and the set $\mathcal{J}$ contains all justifications for each concept in $\mathcal{C}$ then the frequency heuristic for an axiom α determines the number of elements in $\mathcal{J}$ in which α appears. The higher the frequency, the lower the ranking for α. A higher frequency means that α appears in more justifications for the unsatisfiable concepts. This means that if α appears in n justifications in $\mathcal{J}$ then removing α could potentially repair n unsatisfiable concepts from $\mathcal{C}$.

The semantic relevance criterion determines what impact the removal of axiom α has on the entailments in the ontology. If removing α causes many entailments to be eliminated from (or added to) the ontology then the semantic relevance of α is high and therefore its ranking is higher. Syntactic relevance determines how frequently the terms in $\mathsf{Sig}(\alpha)$ appear in the ontology. The more pervasive the usage of these terms in the ontology, the higher the ranking of α.

The test cases criterion is a user-centered heuristic in which the ontology engineer provides a set of *desired* axioms which he/she deems important in the ontology. The axioms which form part of a repair are then ranked according to the number of desired axioms which are eliminated by removing the repair axioms. The higher the number of desired axioms removed, the higher the ranking of the repair axiom. Optionally the ontology engineer is also able to specify axioms which he/she would *not* like to see follow from the ontology. The repair axioms which *retain* the undesired axioms in the ontology when they are removed, would then be given a higher ranking.

3.7.2 Resolving a List of Errors

This section presents the most prominent existing method for repairing multiple unsatisfiable concepts in an ontology. If the ontology engineer has a number of unsatisfiable concepts in the ontology, a naïve approach to eliminating all the unsatisfiable concepts would be to repair each one individually. However, it is frequently the case that the unsatisfiability of a concept C may *cause* the unsatisfiability of another concept D.

Example 3.7 Consider the following ontology:

$$\mathcal{O} = \left\{ \begin{array}{l} 1.\ A \sqsubseteq \neg D, \\ 2.\ B \sqsubseteq D, \\ 3.\ A \sqsubseteq B \sqcap \exists R.E, \\ 4.\ C \equiv A, \\ 5.\ F \sqsubseteq A \sqcup C \end{array} \right\}$$

Ontology $\mathcal{O}$ is A-unsatisfiable, C-unsatisfiable and F-unsatisfiable. It is straightfoward to see that the unsatisfiability of C is caused by the unsatisfiability of A and the unsatisfiability of F is caused by the unsatisfiability of both A and C. □

In these cases it would be useful to detect these dependencies [Kalyanpur et al., 2005b]. That is, it would be useful to determine which unsatisfiable concepts cause others to be unsatisfiable. By repairing these concepts *first*, the other concepts which are unsatisfiable because of these may also be automatically repaired.

Root and Derived Unsatisfiable Concepts

Kalyanpur [2006] provides methods to detect dependencies between unsatisfiable concepts that indicate which concept's unsatisfiability *causes* other concepts to be unsatisfiable in the ontology. These can be identified in two ways: by comparing the justifications for the unsatisfiability of the concepts or (more optimally), by examining the structure of the axioms in the ontology using a *structural tracing* technique and generating an error dependency graph. This procedure categorizes unsatisfiable concepts into *Root* and *Derived* unsatisfiable concepts and indicates the dependencies that these concepts have with respect to one another.

Essentially, the unsatisfiability of a root unsatisfiable concept is not caused by the unsatisfiability of another concept in the ontology and a derived unsatisfiable concept is caused by the unsatisfiability of at least one other concept in the ontology. The formal definition for these terms follows:

Definition 3.5 (Root and Derived unsatisfiable concepts)
Let $\{C_1, \ldots, C_n\}$ be a set of unsatisfiable concepts. Let $\mathcal{J}_i$ be the set of all justifications for the unsatisfiability of the concept C_i. C_i is a *derived* unsatisfiable concept (with respect to C_j) if and only if there is a justification $J \in \mathcal{J}_i$ such that $J \supseteq J'$, where $J' \in \mathcal{J}_j$ and $(j \neq i)$. If an unsatisfiable concept is not a derived unsatisfiable concept then it is a *root* unsatisfiable concept.

The principle of root and derived unsatisfiable concepts is illustrated in the following example.

Example 3.8 Recall the ontology $\mathcal{O}$ from Example 3.7. The reasons (justifications) for the unsatisfiabilities of concepts A, C and F are indicated in Table 3.1 below. Each unsatisfiability has only one justification with respect to the ontology.

JUST($A \equiv \bot$)	JUST($C \equiv \bot$)	JUST($F \equiv \bot$)
$B \sqsubseteq D$ $A \sqsubseteq \neg D$ $A \sqsubseteq B \sqcap \exists R.E$	$B \sqsubseteq D$ $A \sqsubseteq \neg D$ $A \sqsubseteq B \sqcap \exists R.E$	$B \sqsubseteq D$ $A \sqsubseteq \neg D$ $A \sqsubseteq B \sqcap \exists R.E$
	$C \equiv A$	$C \equiv A$ $F \sqsubseteq A \sqcup C$

Table 3.1: Justifications for unsatisfiable concepts in Example 3.7.

The function JUST(α) for some axiom α used in Table 3.1 returns (represents) the set of all justifications for $\mathcal{O} \models \alpha$. We observe that each unsatisfiable concept in Table 3.1 has only one justification and we also observe that JUST($F \equiv \bot$) $\supseteq$ JUST($C \equiv \bot$) $\supseteq$ JUST($A \equiv \bot$). Therefore, by Definition 3.5, we make the following observations:

(i) C is a derived unsatisfiable concept with respect to A.

(ii) F is a derived unsatisfiable concept with respect to A and C.

(iii) A is a root unsatisfiable concept with respect to C and F. □

These kinds of observations illustrated in (i) to (iii) above, are usually derived from a generated error-dependency graph (*EDG*) [Kalyanpur et al., 2005b]. An *EDG* is a set of labelled nodes $v \in \mathcal{V}$ and unlabelled edges $e \in \mathcal{E}$ with $v.label$ denoting the label of a node v. Each node is labelled with an unsatisfiable concept name from the ontology. If some justification for the concept name in $v.label$ is a superset of some justification for a concept name in label $v'.label$ then a directed edge e is drawn from v to v'. e represents a dependency between the concepts in these nodes. In the resulting *EDG*, the root unsatisfiable concepts are represented by those nodes which *do not have an outgoing edge*. A derived unsatisfiable concept is represented by a node which has *at least one outgoing edge*. An *EDG* for the unsatisfiable concepts in Example 3.7 is given in Figure 3.5 below.

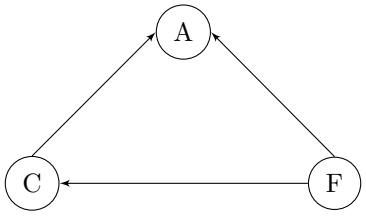

Figure 3.5: Error-dependency graph illustrating the dependencies between unsatisfiable concepts from the ontology in Example 3.7.

Repairing a root unsatisfiable concept may result in all derived unsatisfiable concepts (with respect to the root) being repaired as well such as in the case of Example 3.7. The repairs

for $A \equiv \bot$ generated from JUST($A \equiv \bot$) are $\{B \sqsubseteq D\}$, $\{A \sqsubseteq \neg D\}$ and $\{A \sqsubseteq B \sqcap \exists R.E\}$. It is clear that if any of these three repairs are applied, A, C and F all turn satisfiable. A helpful consequence of the *EDG* graph is that it indicates this information visually. That is, if one repairs a concept with label C in the *EDG* graph then all other concepts which "point" to C in the graph are candidates to be automatically repaired as well.

Therefore the identification of root and derived unsatisfiable concepts in a list of unsatisfiable concepts is very useful for repair purposes. In conclusion, computing root and derived unsatisfiable concepts is an improved solution to resolving multiple unsatisfiable concepts in the ontology. Given an ontology $\mathcal{O}$ and a set of unsatisfiable concepts $\mathcal{C}$ in $\mathcal{O}$, the sequence/procedure for resolving a list of unsatisfiable concepts in an ontology, using the method discussed, is depicted in Figure 3.6.

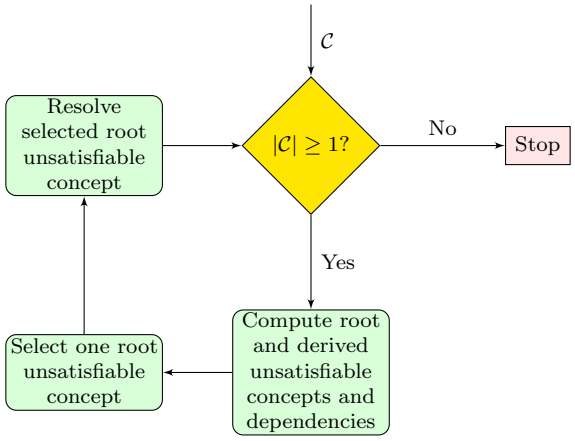

Figure 3.6: Flow diagram showing the sequence of actions for eliminating a list of unsatisfiable concepts $\mathcal{C}$ from some ontology, using the root and derived unsatisfiable approach.

3.8 Open Issues and Limitations

This section discusses a few open areas and drawbacks of the existing approaches to ontology debugging and repair presented in this chapter.

Debugging of general entailments

The focus in current ontology debugging approaches is mainly on removing *unsatisfiable concepts* from ontologies and repairing ontologies which are *inconsistent*. However there are other types of semantic defects such as unintended inferences (also called unwanted axioms) which also require repair. The following ontology $\mathcal{O} = \{\mathsf{EmperorPenguin} \sqsubseteq \mathsf{Penguin}, \mathsf{Penguin} \sqsubseteq \mathsf{Bird}, \mathsf{Bird} \sqsubseteq \mathsf{FlyingAnimal}\}$ illustrates the principle of unwanted axioms. It is fairly straightfoward to see that $\mathcal{O} \models \mathsf{Penguin} \sqsubseteq \mathsf{FlyingAnimal}$ and $\mathcal{O} \models \mathsf{EmperorPenguin} \sqsubseteq \mathsf{FlyingAnimal}$. While it is clear that these consequences logically follow from the ontology, they do not accurately reflect information about the chosen domain of the ontology. In the context of a biological domain, it is known that penguins cannot fly and hence these axioms are *unwanted*.

While the reasoner is able to detect these inferences, it is obviously not able to determine if they are intended or not. Therefore it is the responsibility of the ontology engineer together with the domain expert to study the inferences of their ontology and decide which of the inferences are desired or not desired. While the current work is in agreement with how to identify the unwanted axioms and the foundation is there for extending repair to these entailments, there is no comprehensive repair method specifically dealing with them.

Computing Ontology Solutions

Identifying justifications for errors in ontologies has been dealt with to a large extent in the literature. Identifying repairs for these errors has also been dealt with to a lesser extent. However there is no significant work on using these principles to identify ontology solutions (Definition 3.4) directly.

Historically, the need for explanation services in ontology development tools which help the ontology engineer understand why certain consequences occur in the ontology, encouraged the focus on justifications rather than on ontology solutions. However, the computing of ontology solutions is simply the dual of computing a justification. That is, a justification is a *minimal* subset of the ontology from which a consequence α follows. Whereas, an ontology solution is a *maximal* subset of the ontology from which the consequence does *not* follow.

Since the notion of a justification and ontology solution are closely linked, it is relatively straightfoward to modify most algorithms for computing justifications to compute ontology solutions.

Repairing multiple errors

Identifying root and derived unsatisfiable concepts is helpful for efficiently resolving multiple unsatisfiable concepts in an ontology. If root unsatisfiable concepts are repaired first, all unsatisfiable concepts which are derived from this root may be automatically repaired. However, a drawback to this method is that one has to continually recompute (update) the dependencies between the remaining unsatisfiable concepts after repairing each root. For example, consider the left-hand side EDG in Figure 3.7.

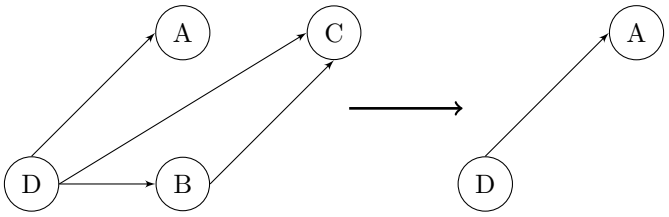

Figure 3.7: EDGs indicating the updating of dependencies after repairing unsatisfiable concept name C.

We observe that A and C are the root unsatisfiable concepts with B and D being derived unsatisfiable. If we repair C *first*, then B is automatically repaired because it has no other dependencies other than on C and the revised EDG is shown on the right-hand side in the figure.

This chapter has given an introduction to the prominent strategies for ontology debugging and repair. Firstly, the focus in existing approaches is primarily on *explanation* and to a

lesser extent on *repair* for the errors in the ontology. Secondly, the type of errors which are mainly concentrated on are unsatisfiable concepts and ontology inconsistency. In the next chapter we contribute an alternative approach to debugging and repair in which we consider the problem of a set of *unwanted axioms* in the ontology. We generalize the different types of semantic defects in DL ontologies to the notion of an unwanted axiom. We then provide a method for repairing (eliminating) these unwanted axioms in the ontology. The specific approach we follow is based on some of the principles used in repairing multiple unsatisfiable classes in the ontology using the *root and derived unsatisfiable concepts* approach.

Chapter 4

Ontology Repair Using Root Justifications

This chapter presents the main contribution of the dissertation which is an alternative approach to ontology debugging and repair in which we generalize the different semantic defects in DL ontologies into a common form as *unwanted axioms*. Hence, the ontology repair problem that we focus on in this chapter is eliminating a *set* of unwanted axioms from the ontology. In order to do this we introduce a special kind of justification, which we term a *root justification*. These justifications are useful because they help us to determine repairs for *all* the unwanted axioms in a list and not just for a *single* one.

In Chapter 3, the standard approaches to resolving ontology errors were discussed. These approaches focus on detecting and repairing the most common semantic defects (unsatisfiable concepts and inconsistent ontologies). In this chapter we also consider another type of semantic defect known as an *unintended inference* or *unintended consequence*. We show that this semantic defect can also be represented as an unwanted axiom.

The outline of the chapter is as follows. We first discuss how the different semantic defects in DL ontologies can be represented in a common form as unwanted axioms. We then briefly introduce the problem of a set of such unwanted axioms in the ontology. The principle of a root justification is then defined and discussed followed by an algorithm for computing these justifications. We then show how root justifications are helpful for generating repairs for a set of unwanted axioms in the ontology. Thereafter we show how these repairs can be identified from the root justifications. It is illustrated that one only needs to consider the root justifications in order to identify *all* the repairs for eliminating the entire set of unwanted axioms from the ontology. Finally, a conclusion is given to summarize the ontology repair solution that is discussed in this chapter.

4.1 Generalization of Errors

In this section we show that common semantic defects like unsatisfiable concepts and inconsistent ontologies are special cases of a more general error which we call an *unwanted axiom*. We also show that another semantic defect known as an *unintended inference* is equivalent to an unwanted axiom. We define an *unwanted axiom* as any logical consequence

or implicit axiom which follows from the ontology, that is not desired by the ontology engineer and domain expert.

4.1.1 Unsatisfiable Concepts

In Section 3.2 the axiom-driven view of ontologies was mentioned. An alternative to this is the *concept-driven* view of an ontology. In this view the ontology is seen as a set of concepts. Each of these concepts has a *concept definition*. The concept definition for some concept name C depicts the relationships that C has with other concepts in the ontology. These relationships can be specified through expressing complex concepts such as the concept $\exists R.C$, or through subsumption-based relationships such as "D is a superconcept of C", "C is equivalent to D" and "C and D are disjoint".

Therefore in a concept-driven view of the ontology, it makes sense to talk about *unsatisfiable concepts* rather than *unwanted axioms*. However, there is definitely a correlation between the two types of error. This is because all the information in the concept-driven view can be expressed in an axiom-driven view as well. For example the axiom $C \sqsubseteq D$ captures the same meaning as the concept-driven definition "D is a superconcept of C". Similarly $C \equiv D$ has the same meaning as "C is equivalent to D" and $C \sqcap D \equiv \bot$ the same as "C and D are disjoint". Therefore it should be noted that the unsatisfiability of a concept can also be represented in an axiom. This is illustrated in Chapter 2. For example, if concept C is unsatisfiable with respect to an ontology $\mathcal{O}$, then it follows that $\mathcal{O} \models C \equiv \bot$.

4.1.2 Unintended Inferences

Another important error in ontologies is that of the unintended inference. While the reasoner is able to *make* inferences from the ontology, it is not able to distinguish which are intended and which are not, for all *types* of such inferences.

Example 4.1 Consider the following ontology:

$$\mathcal{O} = \left\{ \begin{array}{l} 1.\ \mathsf{EmperorPenguin} \sqsubseteq \mathsf{Penguin}, \\ 2.\ \mathsf{Penguin} \sqsubseteq \mathsf{Bird}, \\ 3.\ \mathsf{Bird} \sqsubseteq \mathsf{FlyingAnimal} \end{array} \right\}$$

There are two unwanted consequences of $\mathcal{O}$. These are $\mathsf{Penguin} \sqsubseteq \mathsf{FlyingAnimal}$ and hence $\mathsf{EmperorPenguin} \sqsubseteq \mathsf{FlyingAnimal}$. It is obvious that both these axioms logically follow from the ontology, however, they do not accurately reflect information about the chosen biological domain. In such a domain it is accepted that penguins cannot fly. These axioms are therefore termed *unintended inferences*. □

Unintended inferences are identified by the ontology engineer together with an expert in the domain. They ultimately study the inferences of the ontology which are provided by the reasoner and decide which of these are actually unintended and thus become unwanted axioms (selected/flagged for repair).

4.1.3 Inconsistent Ontologies

An inconsistent ontology $\mathcal{O}$ has no models. Therefore *every* axiom follows from the ontology as explained in Chapter 3. A very useful property of ontology repair is that *any* repair will result in a consistent ontology, since there is at least one axiom which does not follow from the resulting ontology after the repair is applied.

The inconsistency of an ontology $\mathcal{O}$ is indicated (on the axiom level) by $\mathcal{O} \models \top \sqsubseteq \bot$. The axiom in this entailment can therefore be used to represent the inconsistency of the ontology, in a set of unwanted axioms from the ontology.

4.1.4 Unwanted Axioms

Unwanted axioms are the most general type of ontology error because all the other types of errors discussed thus far can be represented as one or more unwanted axioms. Therefore, all the "heterogeneous" semantic defects in some ontology can be given a common representation and be collated in a "homogeneous" unwanted axiom list. An example follows.

Example 4.2 Consider the following ontology representing a domain of biological organisms:

$$\mathcal{O} = \left\{ \begin{array}{l} 1.\ \mathsf{Mammal} \sqcap \mathsf{NonMammal} \equiv \bot, \\ 2.\ \mathsf{Penguin} \sqsubseteq \mathsf{Bird}, \\ 3.\ \mathsf{Platypus} \sqsubseteq \mathsf{Mammal}, \\ 4.\ \exists.\mathsf{laysEggs} \sqsubseteq \mathsf{NonMammal}, \\ 5.\ \mathsf{Platypus} \sqsubseteq \mathsf{AquaticAnimal} \sqcap \exists.\mathsf{laysEggs}, \\ 6.\ \mathsf{Bird} \sqsubseteq \mathsf{FlyingAnimal} \sqcap \exists.\mathsf{laysEggs} \end{array} \right\}$$

The concept referred to by Platypus is unsatisfiable in $\mathcal{O}$ because of Axioms 1, 3, 4 and 5. Furthermore, if we add the axiom Platypus(pete) to $\mathcal{O}$, then $\mathcal{O}$ becomes inconsistent. Another consequence of $\mathcal{O}$ is Penguin $\sqsubseteq$ FlyingAnimal which follows from Axioms 2 and 6. This last error is an unintended inference as mentioned in Example 4.1 (if we consider $\mathcal{O}$ to represent the same domain as in Example 4.1). □

All the errors in ontology $\mathcal{O}$ and $\mathcal{O} \cup \{\mathsf{Platypus}(pete)\}$ from Example 4.2, are currently treated as different problems and hence some existing approaches use different methods to identify, explain and repair each type. However, these errors can be given a common representation in the ontology and thus be handled by the same ontology repair method. The following list contains the ontology errors from Example 4.1 represented as unwanted axioms.

1. $\top \sqsubseteq \bot$ (*Inconsistency of* $\mathcal{O}$)
2. Platypus $\sqsubseteq \bot$ (*Unsatisfiability of* Platypus)
3. Penguin $\sqsubseteq$ FlyingAnimal (*Unintended inference*)

As we can see, this common representation of the errors in the ontology translates the problem of ontology repair for different types of semantic defects into the more uniform

problem of eliminating unwanted axioms from the ontology. The added advantage of this representation is that one need not devise entirely different ad hoc approaches to eliminate each type of error in the ontology. We extend the definition of repair (Definition 3.3) to a set of unwanted axioms as follows:

Definition 4.1 (Repair for list of unwanted axioms)
Let $\mathcal{O}$ be an ontology and $\mathscr{U}$ a set of unwanted axioms in $\mathcal{O}$. A set of axioms $\mathcal{O}' \subseteq \mathcal{O}$ is a *repair* for $\mathscr{U}$ with respect to $\mathcal{O}$ if and only if $\mathcal{O} \backslash \mathcal{O}' \not\models \alpha$ for each $\alpha \in \mathscr{U}$ and there exists no $\mathcal{O}'' \subset \mathcal{O}'$ such that $\mathcal{O} \backslash \mathcal{O}'' \not\models \alpha$ for each $\alpha \in \mathscr{U}$.

Therefore a repair for a set of unwanted axioms with respect to some ontology is a minimal subset of the ontology which can be removed to eliminate each unwanted axiom in the list. In the next section we introduce a type of justification which helps us to identify these kinds of repairs.

4.2 Root Justifications

In this section we define a special kind of justification for a set of unwanted axioms. This kind of justification is characterized from the work by Kalyanpur et al. [2005b] on root and derived unsatisfiable concepts. In the root/derived approach, the focus is on determining which unsatisfiable concepts in a list are root unsatisfiable and which are derived unsatisfiable with respect to each other.

From Definition 3.5 it is observed that if one has a list of concepts $\mathcal{C}$ which are unsatisfiable with respect to some ontology, then $A \in \mathcal{C}$ is a *derived* unsatisfiable concept if and only if it has at least one justification, say J, which is a strict superset of another justification J' for some other concept in $\mathcal{C}$ (Recall from Definition 2.7 that two justifications for the same entailment cannot be subsets or supersets of each other).

We call the justification J a *derived justification* (with respect to $\mathcal{C}$) because it is a *strict superset* of another justification for some other concept in $\mathcal{C}$. Conversely a *root justification* is a justification which is not a derived justification. Intuitively this means that a root justification (with respect to a list of unsatisfiable concepts $\mathcal{C}$) does not have any strict subset which is also a justification for some other concept in $\mathcal{C}$.

In this work we do not deal with a list of unsatisfiable concepts but rather a set of *unwanted axioms*. However, the intuitive definitions for root and derived justifications apply in the same general way to this problem because justifications are applicable to any *general entailment* and not only the specific entailment of an unsatisfiable concept (see Definition 2.7).

We now give a formal definition for root and derived justifications for a set of unwanted axioms:

Definition 4.2 (Root justification)
Given an ontology $\mathcal{O}$, a set of unwanted axioms $\mathscr{U} = \{\alpha_1, \ldots, \alpha_n\}$ in $\mathcal{O}$ and a collection $\mathscr{J} = \{\mathcal{J}_1, \ldots, \mathcal{J}_n\}$, where $\mathcal{J}_i$ is the set of all justifications for $\mathcal{O} \models \alpha_i$. A set $J \in \bigcup_{i=1}^{n} \mathcal{J}_i$ is a *root justification* for $\mathscr{U}$ with respect to $\mathcal{O}$ if and only if there is no $J' \in \bigcup_{i=1}^{n} \mathcal{J}_i$ such that $J' \subset J$. If J is not a root justification then it is a *derived justification*.

In the root/derived *unsatisfiable concepts* approach, the set inclusion relationships between the justifications of unsatisfiable concepts in a list are used to identify dependencies between the concepts. This is done so that ultimately one can detect the root and derived unsatisfiable concepts from the list/set (order is not important). However, in our approach the focus is not on identifying root and derived unsatisfiable concepts but rather on generating all the *root justifications* for a set of unwanted axioms with respect to some ontology. This latter approach, as we will demonstrate, allows one to look at repair for an entire *set* of unwanted axioms which the former approach does not allow for. For brevity, in the remainder of this chapter, we drop the suffix "with respect to some ontology" when referring to root justifications in certain contexts. Wherever this suffix is absent, we assume that it is implied.

4.2.1 Computing Root Justifications

In this section we present algorithms for computing root justifications for a set of unwanted axioms. We start by giving a simple unoptimized algorithm for computing a single root justification.

Computing a Single Root Justification

Here we present an algorithm for computing a single root justification for an unwanted axiom set. The algorithm is based on the naïve pruning algorithm described in Section 3.3.1. We give this unoptimized version first to highlight the important aspects of the algorithm without distracting the reader with optimizations.

Algorithm 4: (Single root justification)

Input: Ontology $\mathcal{O}$, unwanted axiom set $\mathscr{U}$ ($|\mathscr{U}| \geq 1$)
Output: Root justification J for $\mathscr{U}$
Uses: $entailedAxioms(\mathcal{O}, U)$, which returns $\{\alpha \in U \mid \mathcal{O} \models \alpha\}$

```
1 J := O;
2 foreach α ∈ J do
3 |   if |entailedAxioms(J\{α}, U)| ≥ 1 then
4 |   |   J := J\{α};
5 |   end
6 end
7 return J;
```

The key difference between Algorithm 4 and the naïve pruning algorithm is that we are now considering the entailment of *all* unwanted axioms in a set (in procedure *entailedAxioms*), rather than just a single axiom.

It is clear that Algorithm 4 terminates for all finite inputs of $\mathcal{O}$ and $\mathscr{U}$. This follows from Line 2 of the algorithm which shows that the loop only considers the axioms in the finite set J. We now give an example to demonstrate how Algorithm 4 computes a root justification for a set of unwanted axioms.

Example 4.3 Consider an ontology $\mathcal{O}$ with ten axioms, i.e., $\mathcal{O} = \{1, \ldots, 10\}$.

Let $\mathcal{O}$ have three unwanted consequences (unwanted axioms). We call these axioms γ_1, γ_2 and γ_3. Let us assume that the justifications for these unwanted axioms are as follows. The justifications for γ_1 are {1, 2 ,3} and {4, 5}, γ_2 has one justification which is {1, 3} and the justifications for γ_3 are {4, 5, 7} and {6, 7, 8}.

Applying Definition 4.2 we know that justifications {4, 5} for γ_1, {1, 3} for γ_2 and {6, 7, 8} for γ_3 are *root justifications* for $\{\gamma_1, \gamma_2, \gamma_3\}$. We now demonstrate the functioning of Algorithm 4.2 by applying it to the example above to compute one of these root justifications.

We begin with the input $\mathcal{O}$ with ten axioms $\{1, \ldots, 10\}$ and the unwanted axiom set $\mathscr{U} = \{\gamma_1, \gamma_2, \gamma_3\}$ such that $\mathcal{O} \models \alpha$ for each $\alpha \in \mathscr{U}$. The algorithm starts by assigning the set of axioms in $\mathcal{O}$ to the initial set J which will constitute the root justification when the algorithm terminates. At this point in our example, $J = \{1, \ldots, 10\}$. In Lines 3 and 4, the algorithm loops through each axiom α in J, removing α from J if and only if $J \backslash \{\alpha\}$ entails *at least one* axiom from $\mathscr{U}$. When Axioms 1, 2 and 3 are removed it is still the case that $J \models \{\gamma_1, \gamma_3\}$. This is because the justification {4, 5} still holds for γ_1 and the justifications {4, 5, 7} and {6, 7, 8} still hold for γ_3. Through similar reasoning, after removing Axioms 4 and 5 we find that $J \models \{\gamma_3\}$. At last, when we remove Axiom 6 from J we find that J does not entail any of the unwanted axioms in $\mathscr{U}$. Therefore, Axiom 6 *remains* in J. The same holds for Axioms 7 and 8. Finally, after removing Axioms 9 and 10, it is the case that $J \models \{\gamma_3\}$ and the result is that J constitutes a *root justification* for $\mathscr{U}$. □

The correctness of Algorithm 4 is proved by the following theorem.

Theorem 4.1 *Let $\mathcal{O}$ be an ontology, $\mathscr{U}$ a set of unwanted axioms, and J the output of Algorithm 4 with inputs $\mathcal{O}$ and $\mathscr{U}$. Then J is a root justification for $\mathscr{U}$ with respect to $\mathcal{O}$.*

Proof:
Let us assume that the output set J as computed by Algorithm 4 is *not* a root justification for $\mathcal{U}$. If this is the case, then there is a justification $J' \subset J$ such that J' is a justification for some unwanted axiom(s) in $\mathscr{U}$. But if this is the case then Lines 3 and 4 of Algorithm 4 ensure that the axioms in $J \cap J'$ are removed from the ontology and therefore the resulting J would be such that $J \subseteq J'$. This is a contradiction with $J' \subset J$ and therefore, the assumption that the set J returned by Algorithm 4 is *not* a root justification for $\mathscr{U}$ is false. Therefore J is indeed a root justification for $\mathscr{U}$. □

Before the algorithm begins we know that the ontology $\mathcal{O}$ entails *all* axioms in $\mathscr{U}$. Therefore, from the definition of a justification (see Definition 2.7), we know that there is at least one

justification in $\mathcal{O}$ for each axiom in $\mathscr{U}$. The algorithm considers *each* axiom and removes them one by one from the ontology while it is the case that the resulting ontology entails at least one axiom. This ensures that the output set J entails at least one axiom from $\mathscr{U}$ as well. Therefore J is a justification (and also a root justification) if and only if there is no proper subset of J which is also a justification for some entailment in $\mathscr{U}$.

Of course, Algorithm 4 is computationally intensive because we only consider a single axiom at a time in the ontology and for each consideration we require between 1 and $|\mathscr{U}|$ entailment tests. Therefore we give a more optimized version of this procedure in Algorithm 5 based on the sliding window technique [Kalyanpur, 2006] discussed in Chapter 3, for computing "regular" justifications. The correctness of this algorithm follows from the correctness of the sliding window algorithm for computing a single "regular" justification and Algorithm 4.

Algorithm 5: (Single root justification - Optimized)

Input: Ontology $\mathcal{O}$, unwanted axiom set $\mathscr{U}$ ($|\mathscr{U}| \geq 1$) and window size $k \geq 1$
Output: Root justification J for $\mathscr{U}$
Uses: $entailedAxioms(\mathcal{O}, U)$, which returns $\{\alpha \in U \mid \mathcal{O} \models \alpha\}$

```
1  J := O;
2  W := ∅;
3  while k ≥ 1 do
4  |  W := getNextWindow(J, k);
5  |  while |W| ≠ 0 do
6  |  |  if |entailedAxioms(J\W, U)| ≥ 1 then
7  |  |  |  J := J\W;
8  |  |  end
9  |  |  W := getNextWindow(J, k);
10 |  end
11 |  k := ⌊k/2⌋;
12 end
13 return J;
```

We have presented an algorithm for computing a *single* root justification for an unwanted axiom set. We now describe an approach to identify *all* root justifications for the unwanted axiom set, provided that we have a method for computing a *single* root justification.

Computing all Root Justifications

In order to ensure that we are able to generate *complete* repairs for an unwanted axiom set in an ontology, it is necessary to know *all* the root justifications for the unwanted axiom set. In Chapter 3, we have given the description for a variant of Reiter's Hitting Set Algorithm which computes *all* the "regular" justifications for a single entailment [Kalyanpur, 2006]. In this section we show how this variant algorithm can be used (with some slight modifications) to compute all *root* justifications for a *set* of unwanted axioms.

Recall that the variant algorithm constructs a justification tree which is analogous to an HS-tree in Reiter's original algorithm. Each node v in the justification tree represents a justification for the unwanted entailment with respect to ontology $\mathcal{O}\backslash P(v)$ where $\mathcal{O}$ is the original ontology and the function $P(v)$ returns the set of axioms on the path from the root node to node v in the tree.

We are able to construct a similar tree to the one described above which we call a *root justification tree* for an unwanted axiom set $\mathscr{U}$, with respect to some ontology $\mathcal{O}$. A root justification tree is analogous to a justification tree and can be defined in the following way. A root justification tree, $T_{\mathcal{R}}$, is a set of nodes $\mathcal{V}_{\mathcal{R}}$ and edges $\mathcal{E}_{\mathcal{R}}$. Each node $j \in \mathcal{V}_{\mathcal{R}}$ has a label $j.label$ which is a root justification for the unwanted axiom set $\mathscr{U}$ with respect to the ontology $\mathcal{O}$. Each edge $e \in \mathcal{E}_{\mathcal{R}}$ has label $e.label \in \bigcup_{j.label \in T_{\mathcal{R}}} j.label$ i.e., $e.label$ is an axiom of some root justification. $P(j)$, the path function, returns the set of edge labels (axioms) on the path from the root node to node j.

We construct a root justification tree $T_{\mathcal{R}}$, in a breadth-first fashion using the following rules:

(i) Recall that the first step in the construction of the justification tree using the variant algorithm was to generate a root node, labelled with a justification for a single entailment. The corresponding first step in the the construction of the *root* justification tree is to generate a root node j_{root} for $T_{\mathcal{R}}$ which is labelled with a *root* justification for $\mathcal{U}$ with respect to $\mathcal{O}$. We can use Algorithm 4 or 5 to generate such a justification.

(ii) If a node j in the root justification tree is labelled with a root justification J, then for each axiom $\alpha \in J$, a successor node j_α is attached to j via an edge e_α which is labelled with α.

(iii) Each successor node j_α in the justification tree is labelled with a root justification J' for $\mathscr{U}$. J' is generated using the same method as in rule (i). However the difference now is that J' is computed with respect to the ontology $\mathcal{O}\backslash P(j_\alpha)$ and *not* $\mathcal{O}$. If it turns out that $\mathcal{O}\backslash P(j_\alpha)$ does not entail any axiom in $\mathscr{U}$, then of course there is no justification for any axiom in $\mathscr{U}$ and hence no *root* justification for $\mathscr{U}$ with respect to $\mathcal{O}\backslash P(j_\alpha)$. In this case we label j_α with '✓' indicating a terminating/leaf node with no successors.

When the algorithm (construction of the root justification tree) terminates, each unique node label represents a root justification for the entailment. Furthermore the set of all node labels in the tree represent *all* root justifications for the entailment in question [Kalyanpur, 2006, Theorem 4].

We now give an example of a root justification tree and thereafter analyze its properties. Figure 4.1 depicts an example of an unwanted axiom set and the justifications for each unwanted axiom in this set (reused from the demonstration of Algorithm 4). An example root justification tree constructed for this unwanted axiom set is depicted in Figure 4.2.

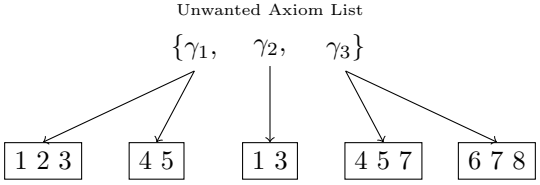

Figure 4.1: All justifications for the set of unwanted axioms γ_1, γ_2 and γ_3

From Figure 4.1 and Definition 4.2 for root justification, it is clear that the root justifications for the unwanted axiom set are {4, 5}, {1, 3} and {6, 7, 8}. It is also clear that each

distinct node label in the root justification tree, in Figure 4.2, corresponds to these root justifications.

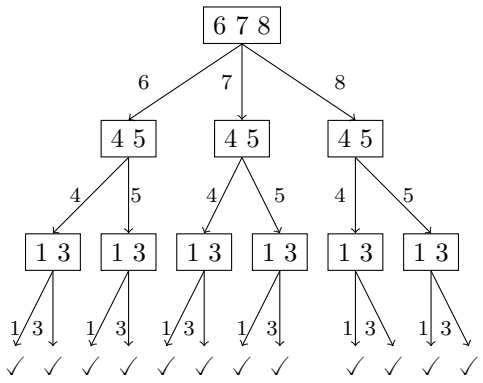

Figure 4.2: Root Justification Tree for the set of unwanted axioms $\{\gamma_1, \gamma_2, \gamma_3\}$

In the next section we show how repairs for the unwanted axiom set can be identified from the root justifications for the unwanted axiom set.

4.2.2 Repairs From Root Justifications

Once we have computed all root justifications for an unwanted axiom set, there are two suggested methods to identify the repairs (according to Definition 4.1) for the unwanted axiom set. The choice of method depends on the algorithm which was used to compute *all* the root justifications. The first method is as follows.

If the root justifications were identified by constructing the root justification tree as described in Section 4.2.1 then the repairs can be identified directly from the tree itself (similar to the first approach for identifying all repairs from "regular" justifications described in Section 3.7). Specifically, *all* repairs for the unwanted axiom set correspond to all minimal hitting sets of the root justifications in the root justification tree. That is, all the repairs for the unwanted axiom set correspond to the sets $P(v)$ in the root justification tree such that v is a terminating node and there is no terminating node v' such that $P(v') \subset P(v)$. This result is formalized in the following theorem.

Theorem 4.2 *Let $\mathcal{J}_R$ be the set of all root justifications for an unwanted axiom set $\mathscr{U}$ with respect to some ontology $\mathcal{O}$. Then the minimal hitting sets of $\mathcal{J}_R$ correspond to all repairs for $\mathscr{U}$ with respect to $\mathcal{O}$.*

Proof:
Let $\mathscr{U}$ be a set of unwanted axioms and $\mathcal{J}_{\mathscr{U}}$ be the set containing all justifications for each axiom in $\mathscr{U}$.

(i) We know from Section 3.7.1 that all repairs (Definition 3.3) for a single unwanted axiom correspond to the minimal hitting sets for its set of justifications [Kalyanpur, 2006, Theorem 3].

(ii) We know from Definition 4.2 that each justification in $\mathcal{J}_{\mathscr{U}}$ is either a *root* justification or a *derived* justification for $\mathscr{U}$.

We prove theorem 4.2 by contradiction: Assume that we have the set of all root justifications $\mathcal{J}_R$ for $\mathscr{U}$ with respect to $\mathcal{O}$ and the minimal hitting sets for $\mathcal{J}_R$ such that these minimal hitting sets *do not* correspond to all repairs for $\mathscr{U}$ with respect to $\mathcal{O}$.

(iii) From (i), it follows that the set of *all* repairs for each axiom in $\mathscr{U}$ correspond to the minimal hitting sets for all justifications in $\mathcal{J}_{\mathscr{U}}$.

(iv) From our assumption that the minimal hitting sets for $\mathcal{J}_R$ do *not* correspond to all repairs for $\mathscr{U}$ it follows that there is at least one repair for $\mathscr{U}$ which is not a minimal hitting set for $\mathcal{J}_R$.

(v) Therefore, from (iii) and (iv), it follows that there is a $\mathcal{J}$ where $\mathcal{J}_R \subset \mathcal{J} \subseteq \mathcal{J}_{\mathscr{U}}$ such that the minimal hitting sets of $\mathcal{J}$ correspond to all repairs for $\mathscr{U}$. But we know from (ii) that $\mathcal{J} \backslash \mathcal{J}_R$ contains only *derived* justifications for $\mathscr{U}$ and each derived justification $J_d \in \mathcal{J}$ is such that $J_d \supset J_r$ for some $J_r \in \mathcal{J}_R$. Therefore, the minimal hitting sets for $\mathcal{J}$ are also the minimal hitting sets for $\mathcal{J}_R$ (minimal hitting sets for conflict sets are the same as for the *minimal* conflict sets [Reiter, 1987, p.68]).

A contradiction arises because we have stated that the minimal hitting sets for $\mathcal{J}$ correspond to all the repairs for $\mathscr{U}$ (see (v)) but those for $\mathcal{J}_R$ do not (see (iv)). Therefore the supposition that Theorem 4.2 is false is incorrect and therefore Theorem 4.2 holds. □

The result of Theorem 4.2 is significant because it means that we only need to consider the root justifications for repairing (eliminating all the unwanted axioms from) the ontology.

The second method for identifying the repairs may be chosen if an alternative method (entirely different from the root justification tree approach) is used to compute all root justifications. In this case one trivially applies Reiter's original hitting set algorithm to find all minimal hitting sets of all the root justifications computed (similar to the second approach for identifying all repairs from "regular" justifications described in Section 3.7). These minimal hitting sets then correspond to the complete list of repairs for the unwanted axiom set.

4.3 Comments

In this chapter, we have presented an alternative approach to ontology debugging and repair in which we generalize all the different semantic defects in DL ontologies to a common defect which we call an unwanted axiom. The specific ontology debugging and repair problem which was characterized is eliminating a *set of unwanted axioms* from the ontology.

We presented an approach to resolving the above mentioned problem by introduced a special kind of justification called a *root justification*. This justification was illustrated to be useful for resolving the entire set of unwanted axioms in the ontology. We then presented algorithms for *identifying* the root justifications for the set of unwanted axioms and finally demonstrated how all the *repairs* for the unwanted axiom set can be identified from its root justifications.

In the next chapter, we discuss the implementation of a Protégé plugin which we have developed to compute root justifications and repairs for a set of unwanted axioms in some ontology, which is specified by the ontology engineer.

Chapter 5

Implementation and Evaluation

This chapter discusses the implementation and evaluation of a software plugin called *OntoRepair* for the ontology editor Protégé [Knublauch et al., 2004]. The plugin considers a user-specified list of unwanted axioms in the ontology and uses the approach discussed in Chapter 4 to provide a two-fold service: (i) Computation of the root justifications for a specified list of unwanted axioms in the ontology and (ii) Identification of all the repairs for this list, i.e., all minimal sets of axioms such that if they are removed from the ontology will cause *all* unwanted axioms in the list to be eliminated from the ontology.

In the first part of this chapter, we introduce the Protégé ontology editor and the Java™-based OWL API[1] for developing OWL ontologies which is used for manipulating and reasoning with ontologies in *OntoRepair*. We then give a brief overview of the different kinds of Protégé plugins and what they look like.

In the second part of the chapter we give a concise description of *OntoRepair* with illustrations of its functioning. We then analyze the results of applying *OntoRepair* to eliminate a list of unwanted axioms in a number of ontologies with varying properties. Finally, we compare the results of using *OntoRepair* to the results of using a *naïve approach* which eliminates each unwanted axiom in the list *individually*.

5.1 Protégé

Protégé is a free and open source ontology editor and knowledge base framework, originally developed by the Stanford Center for Biomedical Informatics Research at the Stanford University School of Medicine (other ontology editors include SWOOP[2] [Kalyanpur et al., 2005a], Apollo[3] and KAON[4]). There are two major versions, Protégé 3.x and Protégé 4.x. The former provides support for working with older frame-based ontologies along with ontologies expressed using the now dated OWL 1.0 languages. Protégé 4 was developed as part of the CO-ODE project[5] by the University of Manchester in collaboration with Stanford Medical Informatics. The Protégé 4.x versions do not allow frame-based ontology

[1] `http://owlapi.sourceforge.net`
[2] `http://www.mindswap.org/2004/SWOOP`
[3] `http://apollo.open.ac.uk`
[4] `http://kaon.semanticweb.org`
[5] `http://www.co-ode.org`

editing and are tailored for the latest OWL 2 standard of ontology languages. Protégé 4.x requires an ontology API for loading and manipulating ontologies. We introduce our choice of API later on in this section.

5.1.1 Protégé User Interface

Protégé makes use of a "tabbed" graphical user interface (see Figure 5.1). Each tab displays a different "ontology view". The most frequently used tabs are the following:

- Classes
- Object Properties
- Individuals
- Entities

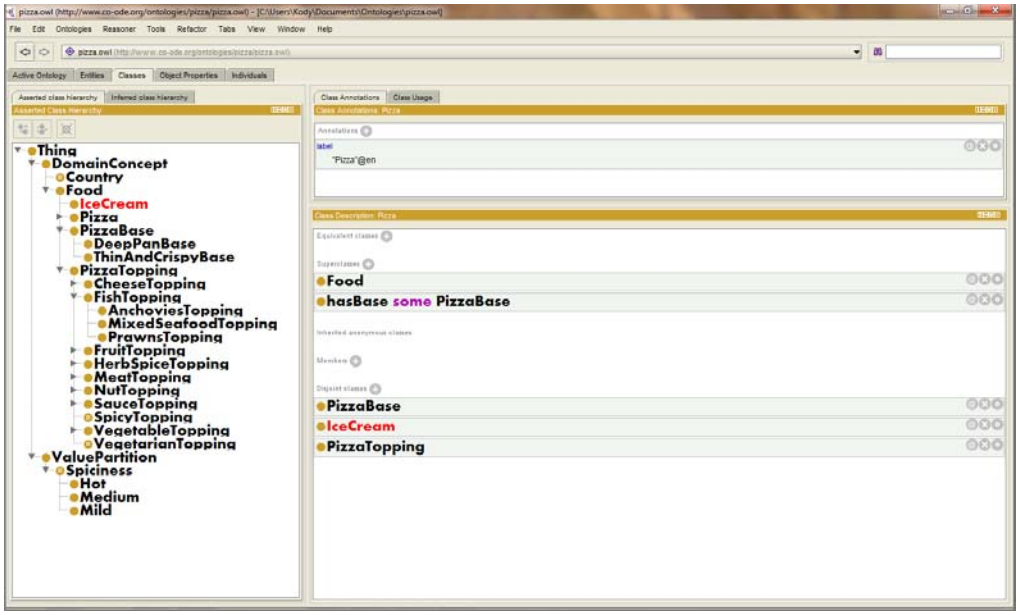

Figure 5.1: Protégé 4 ontology editor.

The Classes tab offers a class-driven (concept-driven) view of the ontology. The focus in this tab is therefore on the "asserted" class hierarchy and the definitions for the classes (concept names) in the hierarchy. The asserted class hierarchy displays the taxonomy of the concept names in the ontology *as specified by the ontology engineer* as opposed to the "inferred" class hierarchy which is the (possibly) revised hierarchy after the inferences made by the reasoner are incorporated into the asserted hierarchy. Concept names in Protégé are indicated by a preceding tan-coloured circular icon in the interface.

The Object Properties tab displays the object properties (role names) in the ontology and the relevant information pertaining to them. Object properties are differentiated from other entities in the ontology by a preceding blue-coloured rectangular icon. The Individuals tab displays the ABox focused information about the instances/individuals in the ontology. Individuals are indicated by a preceding purple-coloured, diamond-shaped icon.

It should be noted that information in the tabs described are likely to overlap because of the "connected" nature of the entities in the ontology. However, an important thing to notice is that each tab views the ontology from a different *perspective*, i.e., from a class, object property or individuals perspective. The other commonly used tab, from an ontology editing point of view, combines the previously discussed tabs so that all the perspectives can be viewed and browsed simultaneously. This is called the Entities tab.

Protégé also has support for a variety of ontology reasoners which can be installed as plugins in the Protégé system and be accessed via the "Reasoner" menu in the main toolbar. Protégé makes effective use of color in the interface to indicate the inferences made by the reasoner after important tasks such as classification. For example, after ontology classification, *inferred* information is displayed in the ontology editing tabs against a pale yellow background. Also, concept names which are found to be unsatisfiable in the ontology are displayed using red text.

5.1.2 The OWL API

The OWL API [Horridge and Bechhofer, 2010] is a Java™-based API developed for creating, editing and managing OWL ontologies. The latest version (version 3 as of writing) follows, very closely, the OWL 2 structural specification [Motik et al., 2009] recommended by the W3C. This is the main reason for choosing OWL API for the development of *OntoRepair*, as opposed to other java-based ontology APIs such as Jena[6] or Sofa[7]. The reader is reminded that there is a correlation between several constructs in OWL languages and DLs. Figure 5.1 illustrates a few of these correlations.

OWL Syntax	**DL Syntax**
owl:subClassOf	$\sqsubseteq$
owl:complementOf	$\neg C$
owl:intersectionOf	$\sqcap$
owl:unionOf	$\sqcup$
owl:someValuesFrom	$\exists R.C$
owl:allValuesFrom	$\forall R.C$

Table 5.1: Correlation between OWL and DL syntax.

In the OWL API, ontologies are viewed as a set of axioms and annotations (meta-information about entities in the ontology). The API serves as a set of interfaces for manipulating and reasoning with OWL ontologies. The OWLOntologyManager interface in particular, provide the means for loading, creating, and editing the ontologies. The OWLOntologyManager also provides access to the OWLReasoner interface, which in turn, provides access to the inference services of the selected ontology reasoner. Figure 5.2 below, shows the main aspects of the OWL API which are used in the development of *OntoRepair*.

The OWLOntology interface provides access to all the axioms in an ontology. It also allows for selective access to axioms by various criteria such as axiom type (equivalence axioms, subclass axioms, disjointness axioms etcetera) or axiom signature. The loading/saving, creation and manipulation of ontologies is handled by the OWLOntologyManager. The

[6] `http://jena.sourceforge.net`
[7] `http://sofa.projects.semwebcentral.org`

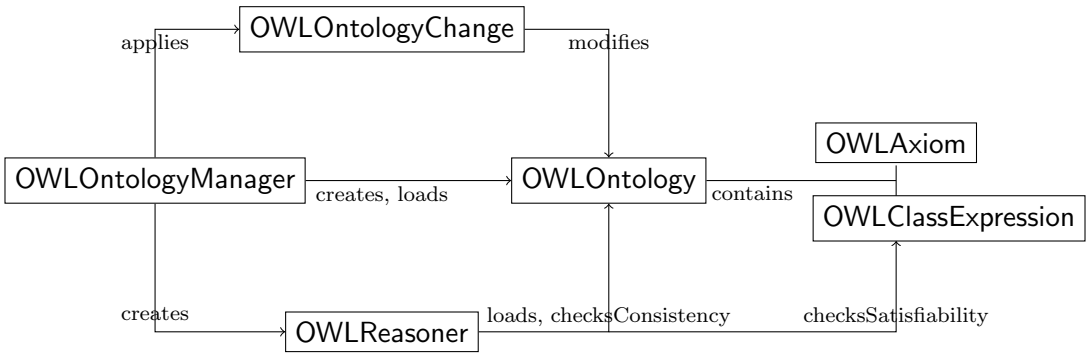

Figure 5.2: Functioning of the main aspects of OWL API which are used in *OntoRepair*.

manipulation (removal or addition of axioms), in particular, is also done by the OWLOntologyManager through the OWLOntologyChange objects RemoveAxiom and AddAxiom.

On the reasoning side, OWLReasoner provides support for the standard reasoning procedures offered by many of the DL reasoners currently available, for example, ontology classification, consistency checking and checking class satisfiability. There are also various convenience methods to perform other reasoning tasks such as finding the "ancestors" or "descendents" of a particular class.

5.1.3 Protégé Plugins

Protégé plugins are categorized by their topic and type. The topic of the plugin explains the specific service which the plugin provides. For example, the topics of Protégé can range from simple user interface tweaks to more involved ontology browsing/maintenance tools such as Ontology Visualization tools, Ontology Debugging and Repair tools, Ontology Querying tools etc.

Workspace tab plugins, as the name suggests, adds their functionality to new tabs in the Protégé interface. This is the case for plugins such as the DL Query Tab[8] and our *OntoRepair* plugin. A view component plugin allows one to use or extend the default ontology views, provided by Protégé, in the plugin implementation. For example, one can write a view component plugin to display the classes in the ontology in a "tabbed" arrangement[9] (i.e., the classes of the ontology represented in a list with the subclasses indented with respect to their superclasses).

5.2 Implementation of OntoRepair Plugin

In this section, we discuss issues related to the implementation and evaluation of our *OntoRepair* Protégé plugin.

[8] `http://protegewiki.stanford.edu/wiki/DLQueryTab`
[9] `http://www.co-ode.org/downloads/protege-x/plugin-code-example.php`

5.2.1 Problem/Purpose

The focus in many current ontology debugging tools and systems, is placed on helping the user understand the reasons or causes for the erroneous consequences which occur in the ontology. The way these tools achieve this is by offering explanation services which are based on computing justifications (as discussed in Chapter 3) for the errors in the ontology. Furthermore, the main errors which current tools focus on, are unsatisfiable concepts and inconsistent ontologies.

We have identified a more general scenario during ontology development. At a particular stage of ontology construction, the ontology engineer may pause to consider the consequences which follow from that particular version of the ontology (identified by the chosen ontology reasoner). The engineer then notices that there are a number of heterogeneous consequences (ranging from an inconsistent ontology to unsatisfiable concept names to unintended inferences) following from the ontology which are not desired. The engineer wishes to eliminate all these unwanted consequences in the ontology but finds that different strategies are required to fix the different kinds of consequences which are not desired.

It is clear from this scenario that a holistic approach for ontology repair is required to treat all unwanted consequences in the ontology equally and to remove *all* such consequences from the ontology. The *OntoRepair* plugin implements such an approach. It gathers the set of unwanted consequences (of various types) in the ontology which are specified by the ontology engineer, and generates a list of repairs such that each of these repairs (when applied to the ontology) eliminates the *entire* list of unwanted consequences from the ontology. Figure 5.3 illustrates the sequence of events followed by using *OntoRepair* to eliminate the unwanted consequences.

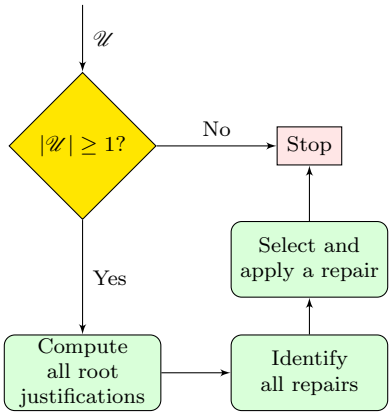

Figure 5.3: Flow diagram showing the sequence of actions for eliminating a list of unwanted axioms $\mathscr{U}$ in some ontology, using *OntoRepair*.

5.2.2 Interface

OntoRepair has an easy to use interface. The important parts of the interface are listed as follows:

1. Repair Toolbar
2. Unwanted axiom list
3. Results list

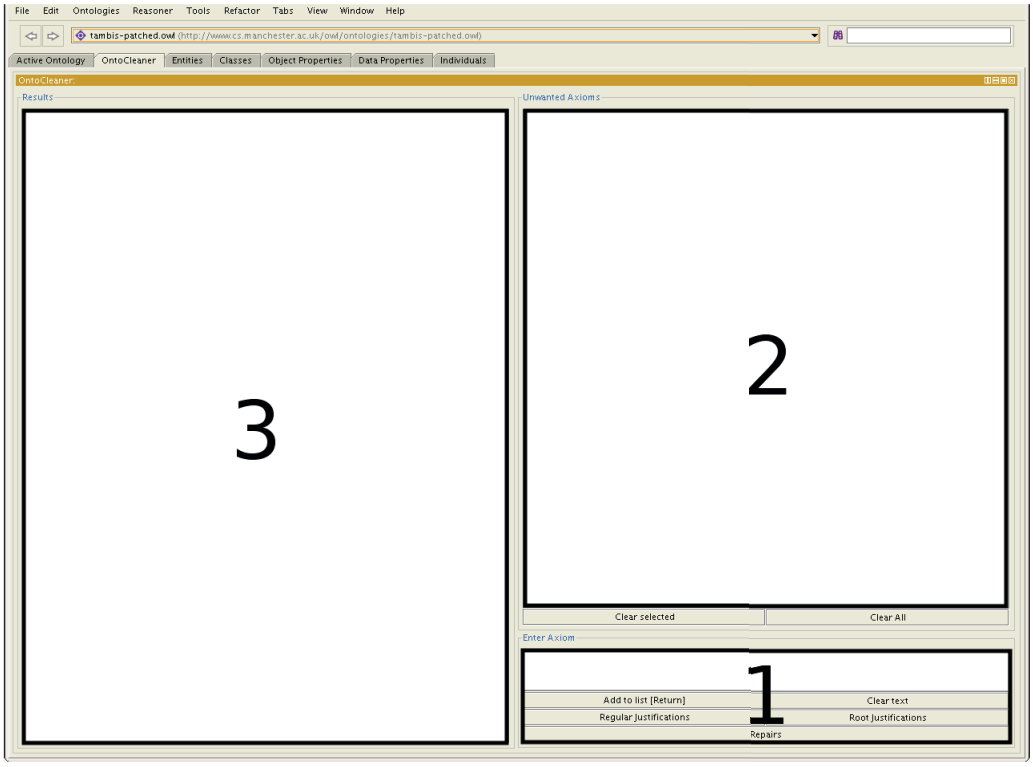

Figure 5.4: *OntoRepair* interface.

The repair toolbar is the control panel of the *OntoRepair* tool. It allows the user to add axioms to the unwanted axiom list, to query for both the regular and root justifications and identify the repairs for eliminating the entire list of unwanted axioms. The results list displays the results of a query. For example, all the root justifications are displayed in this list when the "Root justifications" command from the repair toolbar, is invoked. Figure 5.5 gives a sample output for this action. Similarly the repairs for the unwanted axiom list can be identified by invoking the "Repairs" in the repair toolbar. Sample output is given for this action in Figure 5.6.

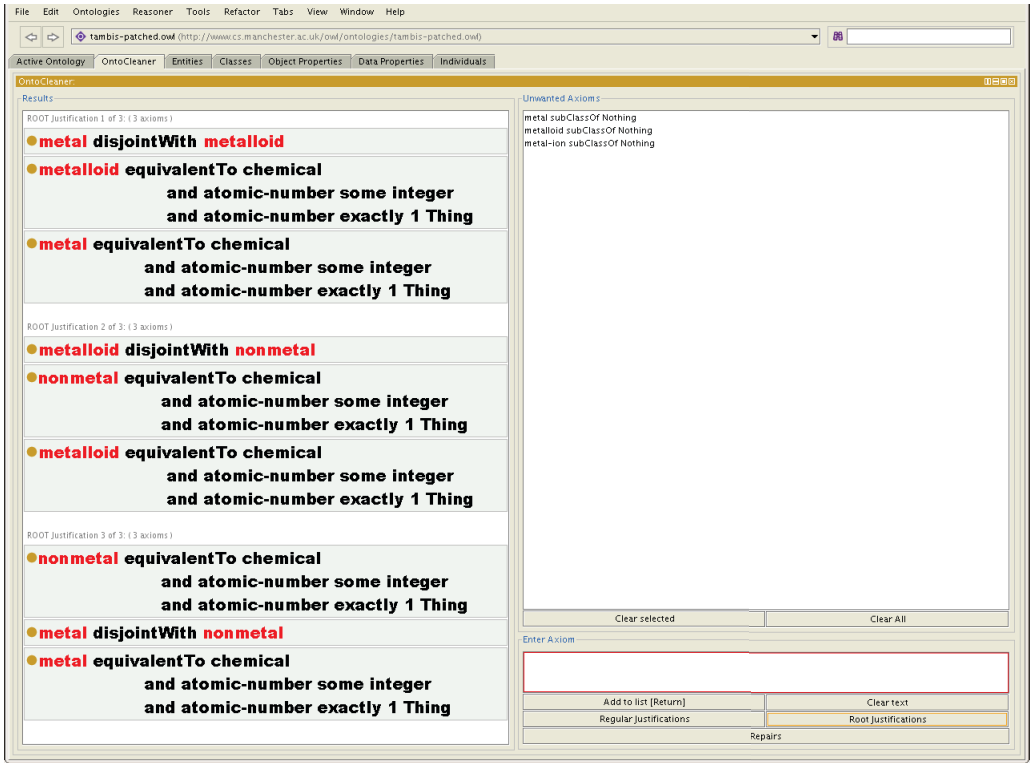

Figure 5.5: *OntoRepair* interface, computing root justifications.

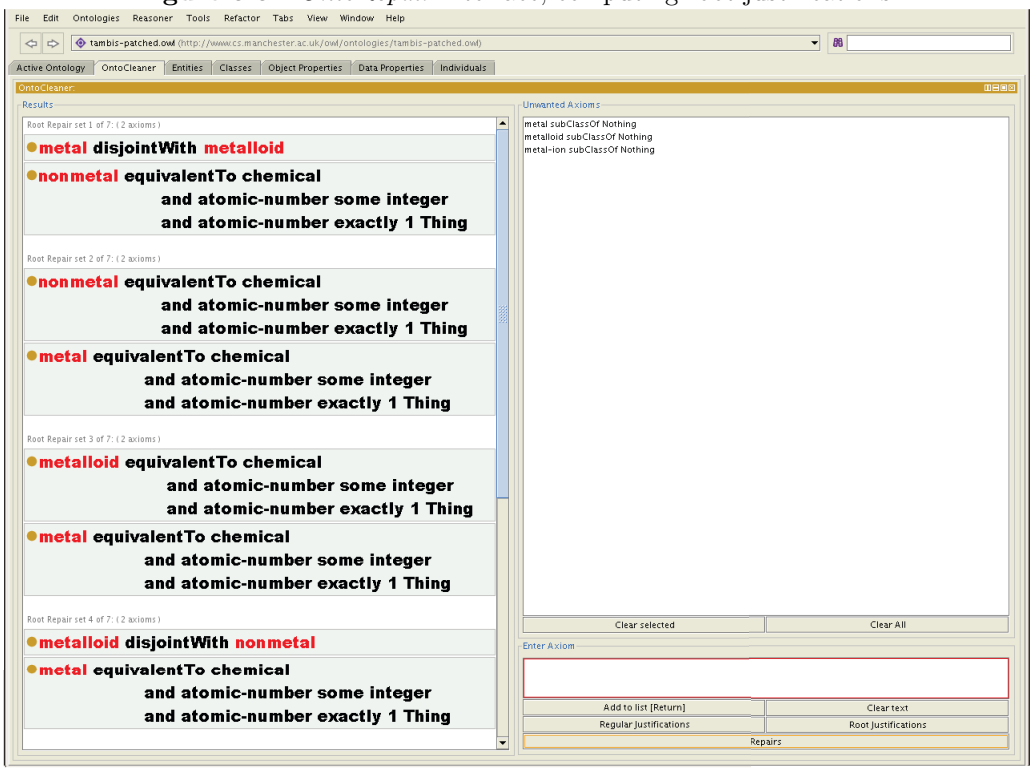

Figure 5.6: *OntoRepair* interface, identifying repairs.

5.3 Evaluation of OntoRepair Plugin

In this section we evaluate the performance of the *OntoRepair* plugin. We start by describing the test cases and data sets which we use to perform the evaluation. The *OntoRepair* plugin and all the test ontologies used in the evaluation are available for download. Please see Appendix A for details.

5.3.1 Test cases

In our experiments we use three sample ontologies of varying size, structure, and application. We have three test cases, one for each ontology and in each case we select four different unwanted axiom lists from the ontology. Therefore each test case has four experiments (one for each unwanted axiom list). In each experiment, we perform a *control* computation. This control computation uses a *naïve approach* to identify all *regular* justifications for each axiom in the unwanted axiom list. We record the *timing* for this as well as the total *number* of regular justifications computed. Thereafter we compute all *root* justifications for the unwanted axiom list and record the same data of *timing* and total *number* of root justifications computed. The results of the latter approach are then compared to the results of the control computation.

Note: Evaluation is not done for identifying the repairs since repairs are generated during the process of computing the justifications (see Sections 3.7 and 4.2.2).

Test Case 1:

In the first test case, a version of the Pizza ontology [Rector et al., 2004] is used. This ontology has roughly seven hundred axioms. Four different unwanted axiom lists of between three and five axioms each are arbitrarily selected from the ontology for the experiments.

Test Case 2:

For the second test case we use the Travel[10] ontology which has roughly one hundred axioms. Four different unwanted axiom lists of between three and five axioms each are arbitrarily selected from the ontology for the experiments.

Test Case 3:

For the last test case, we use the Tambis [Baker et al., 1999] ontology which has roughly six hundred axioms. Four different unwanted axiom lists of between three and five axioms each, are arbitrarily selected from the ontology for the experiments (as in the previous two test cases).

5.3.2 Results

The following tables display the results of each of the test cases discussed in the previous section. The left axis of the "Performance" graphs indicate the time taken for computation (in milliseconds).

[10] `http://protege.cim3.net/file/pub/ontologies/travel/travel.owl`

Test Case 1

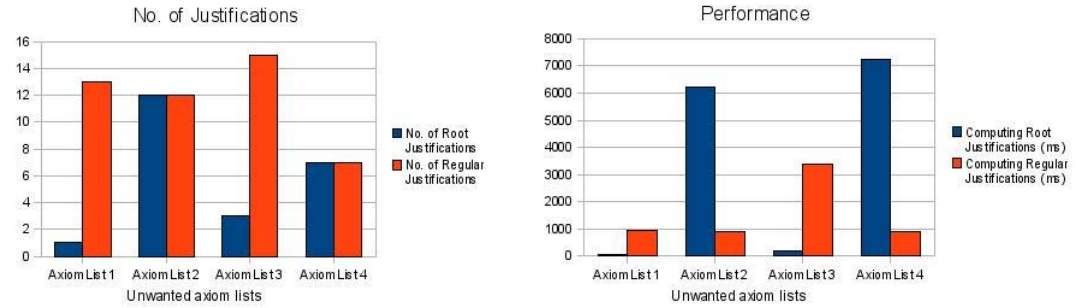

Table 5.2: Results for Test Case 1.

Test Case 2

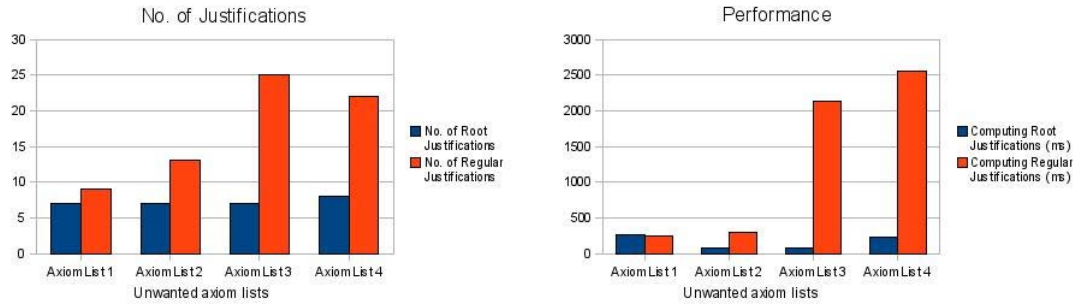

Table 5.3: Results for Test Case 2.

Test Case 3

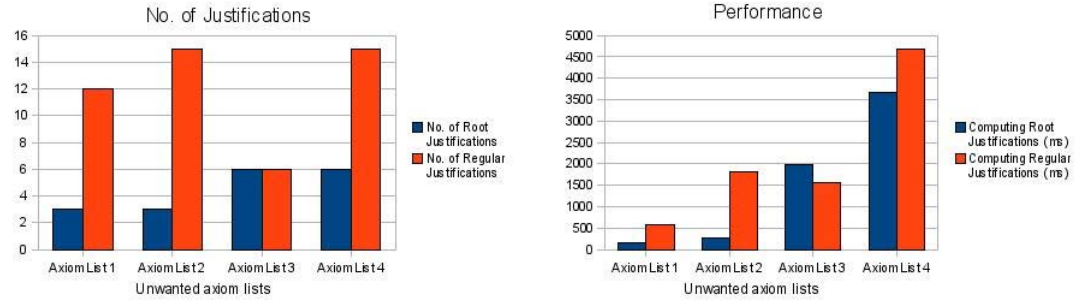

Table 5.4: Results for Test Case 3.

5.3.3 Analysis and Conclusion

It is apparent from the results depicted in the above figures that *OntoRepair* generally performs much better than the naïve approach in the cases where the total number of root justifications for the unwanted axiom list is less than the total number of regular justifications for all the axioms in the list. The only notable exception is axiom list 1 in test case 2. In the cases where the total number of root justifications is the same as the total number of regular justifications the *OntoRepair* approach performs worse than the naïve approach (axiom lists 2 and 4 in test case 1 and axiom list 3 in test case 3).

The overall best performance by the *OntoRepair* approach is observed in test case 2.

This is also the only test case in which the number of root justifications is less than the number of regular justifications in all the experiments. The overall worst performance, was observed in test case 3 where there are two out of four instances in which the number of root justifications and regular justifications are equal. Possibly the main reason for the performance being worse in this case is that we have to keep track of the entailment of *each* of the axioms in the list when computing a single *root* justification, whereas we only have to keep track of the entailment of *one* axiom when computing a single *regular* justification. We call this overhead inherent in computing root justifications the *root-of-list* overhead. Recall also that an entailment check is very computationally expensive and many such checks (depending on the size of the ontology) are needed to compute a (regular or root) justification.

The worst performance in a *single* experiment is for axiom list 4 in test case 1. The reason for this is likely a combination of three things: (a) *root-of-list* overhead (the number of root justifications are equal to the number of regular justifications), (b) the justifications for the axioms in axiom list 4 are *similar*, meaning that they share a significant amount of axioms, and (c) the number of axioms in some justifications is large. In particular, one axiom in axiom list 4 in this test case, has four justifications, each of which has at least eleven axioms. Factors (b) and (c) can cause a considerable performance hit because if two justifications share a significant amount of axioms (or if they have a large amount of axioms) then more entailments will be required because computing both justifications requires a more fine-grained look at which axioms are *unique* to each justification and also the more axioms in the final justification the more entailment checks will be required.

The best performance in a *single* experiment is for axiom list 3 in test case 2. The most likely reasons for this are that (a) the number of root justifications are far less than the number of regular justifications, and (b) the justifications for the axioms in axiom list 3 are not similar (do not share many axioms).

In conclusion, the performance of *OntoRepair* depends on the kinds of axioms contained in the unwanted axiom list. This is because the specific axioms in the list influence the *justifications* which are computed and thus also the number of root justifications for the list versus the number of regular justifications for the axioms in the list. Nine out of twelve of our experiments show instances where the number of root justifications is less than the number of regular justifications. This frequency justifies the use of *OntoRepair* which provides improved overall performance (over the naïve approach) in all these instances. However, there is scope for optimizing the computation of root justifications in *OntoRepair* to be comparable to the performance of the naïve approach, for those cases in which the number of root and regular justifications are equal. Finally, it is important to mention that since the empirical analysis was conducted on the basis of just three examples, the results are unlikely to be statistically significant and conclusive.

Chapter 6

Conclusion

This chapter is divided into two parts. The first part gives a broad summary of the work we have covered throughout this dissertation and the second part concludes the chapter with a discussion about some open issues regarding the ontology repair solution introduced in Chapter 4 and the implementation thereof discussed in Chapter 5.

6.1 Summary of Contribution

In Chapters 2 and 3 we have given the state-of-the-art for the area of debugging and repairing description logic ontologies. We have observed that the predominant trend in this field is on presenting explanations – through the use of justifications – for the semantic defects in the ontology. The semantic defects which are concentrated on are mainly, the unsatisfiable concepts in the ontology and the inconsistency of the ontology. Therefore, the two key areas which we have identified for improvement, is to (a) extend the principles of DL ontology debugging and repair to deal with other kinds of semantic defects as well (such as unintended inferences), and (b) to provide strategies for *eliminating* these semantic defects from the ontology (that is, not just focusing on explaining why they follow from the ontology).

In Chapter 4, we presented an alternative approach to debug and repair DL ontologies, which incorporates the above-mentioned improvements. We extended the list of semantic defects that we deal with to include *unintended inferences*. We then generalized the different kinds of semantic defects (unsatisfiable concepts, inconsistent ontologies and unintended inferences) to a uniform defect which we call an *unwanted axiom*. We show that all the different semantic defects can be represented as an unwanted axiom and we delineate a general problem which we focus on resolving, in our ontology repair solution. The general problem is that of eliminating a *list of unwanted axioms* from the ontology.

We introduced a special type of justification called a *root* justification which was illustrated to be useful for identifying strategies to eliminate all the unwanted axioms in the list, from the ontology. We gave algorithms for computing the root justifications for a list of unwanted axioms and finally, we showed how to identify a set of alternative repair solutions from the root justifications, for eliminating the entire list of unwanted axioms from the ontology.

6.2 Open Issues and Future Work

We now discuss some open issues regarding the ontology debugging and repair approach discussed in Chapters 4 and 5.

In Chapter 5, it was shown that the computation of root justifications for a list of unwanted axioms is faster than computing all the regular justifications for each axiom in the list, in the case where there are fewer root justifications than regular justifications. However in the (worst) case where the number of root and regular justifications are equal, the performance of the method we use for computing these root justifications is far lower than the naive approach of computing all the regular justifications. The main reasons for this are the lack of optimizations in the algorithms for computing the root justifications. We propose two optimizations here to be implemented in future versions of *OntoRepair*. The first optimization is for the algorithm for computing a single root justification. This algorithm monitors the entailment of all the axioms in the unwanted axiom list as axioms are removed from the ontology. Instead of checking the entailment of each axiom in the list at every pass, a caching system can be implemented to keep track of which axioms follow from which subsets of the ontology and reusing this information in subsequent passes of the algorithm. This system will save many entailment checks which are computationally intensive as mentioned in Chapter 3. The second optimization is for computing all root justifications using Reiter's algorithm (Section 4.2.1). The optimization is reused from Reiter's original hitting set algorithm. It is called *node re-labeling*. It applies to the construction of the root justification tree in the following way. If a node v, labelled with a root justification J has been generated previously in the root justification tree, then a node v' which is about to be generated in the tree can be labelled with J as well (without resorting to compute a new root justification label) if and only if the path $P(v') \cap J = \emptyset$. This optimization significantly reduces the instances of recomputing the same root justification in the tree and thus potentially saves a lot of time in computing all the root justifications. We believe that these optimizations have the potential to make the computation of root justifications in the worst case (number of root justifications equal to number of regular justifications) comparable to the naive approach which computes all the regular justifications. Future versions of *OntoRepair* will include these optimizations.

The circumstances, under which the root justification approach to ontology repair constitutes an advantage over naive methods, are not conclusive from our empirical analysis. We intend to investigate this in a more thorough manner by conducting further experiments.

Precise justifications do not include superfluous parts of axioms in the justifications for an entailment. That is, the axioms in a precise justification for some entailment only contains parts which are relevant to the entailment. The general approach by methods for computing precise justifications is to re-write (split) the complex axioms in the ontology into simpler axioms. After this, any standard method for computing justifications can be used to find the precise justifications for the entailment with respect to the new ontology. In future versions of *OntoRepair*, the capability of computing precise justifications will be introduced, making root justifications more precise and, in turn, making repairs more concise as well.

Another planned service which is not currently offered by *OntoRepair* is to aid the user in selecting from the repairs generated by the plugin. Similar heuristics to the axiom ranking

strategy Kalyanpur et al. [2006b] are planned, to determine which repairs may be more appropriate than others.

Finally, we plan to organize *OntoRepair* into a more complete repair solution by providing the user with more advanced meta information during explanation and repair. Effective use of colour and visual cues are proposed to indicate advanced information about explanations and repairs which are generated. This information should help users better understand why the unwanted axioms in the list follow from the ontology and also what effect each repair has on the ontology when it is applied. Lastly, the facility of directly applying repairs to the ontology (through axiom deletions) will be included in future versions of *OntoRepair*.

Appendix

A.1 Mini Tutorial: OntoRepair

In this section we give a step-by-step tutorial on how to set up and run Protégé 4 with the *OntoRepair* plugin. We divide the tutorial into three subsections: First we give a step-by-step guide on how to download and install Protégé 4 and thereafter we show how to download, install *OntoRepair* for Protégé. Finally, we show how to get started using *OntoRepair* in Protégé with an example.

A.1.1 Installing Protégé 4

The version of Protégé that we use in this tutorial is Protégé 4.0.2. While the latest version is Protégé 4.1, this version has some issues with importing certain of the test ontologies that we use. There are a number of different installation options for Protégé. We give the the easiest and safest method by using the platform-independent installer program:

Step 1: Download Protégé 4.0.2 *Installer*

The first step is to download the platform independent installer program for Protégé. For Protégé 4.0.2 this can be found at the following url: `http://protege.stanford.edu/download/protege/4.0/installanywhere`. The specific installer package suitable for your platform is automatically detected on the download page.

Important: Protégé requires the Java Virtual Machine (JVM) version 1.5 or later to run. If you do not have an appropriate version installed, select an installer which is bundled with the JVM. A screenshot of the download page is given in Figure A.1.

Step 2: Install Protégé 4.0.2

The next step is to run the installer program and to follow the prompts to install Protégé 4.0.2 in the directory of your choice. The commands for running the installer are given for each platform on the download page[1]. the most important part of the installation is choosing the JVM which is going to be used by Protégé (see Figure A.2).

Here you can choose from one of the existing JVMs installed on your system or if you have downloaded an installer which includes a JVM, then there will be an option to use this

[1] `http://protege.stanford.edu/download/protege/4.0/installanywhere`

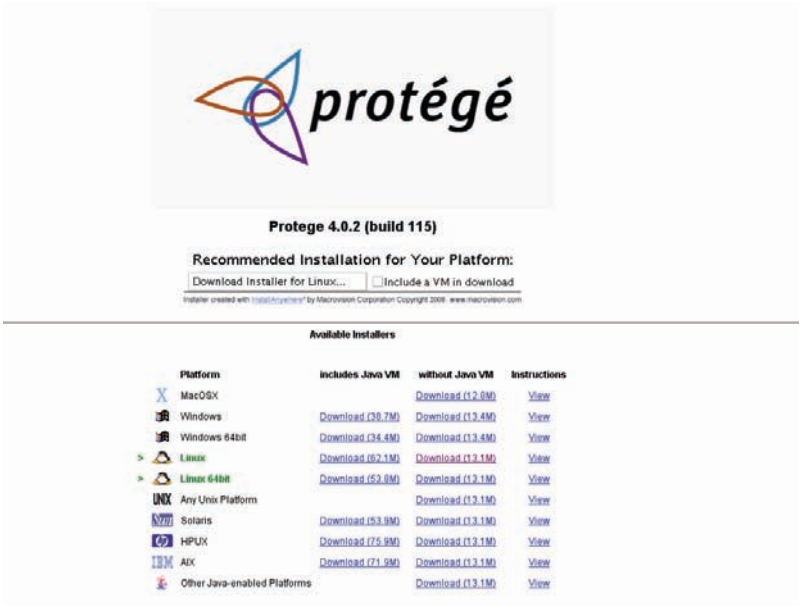

Figure A.1: Step 1: Download Protégé 4.0.2

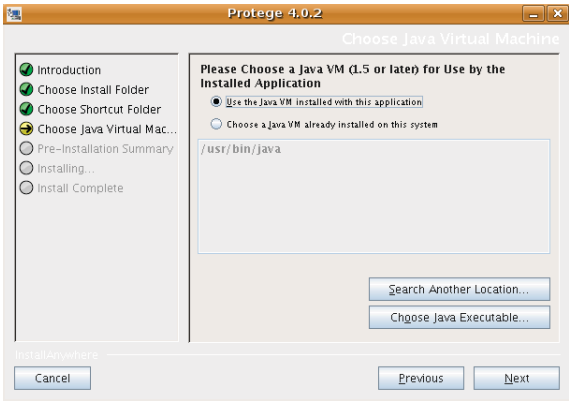

Figure A.2: Step 2: Install Protégé 4.0.2

pre-packaged JVM. Again it is important to make sure that if your choose to use your own existing JVM, then it is version 1.5 or later.

A.1.2 Installing OntoRepair

Protégé 4.0.2 should now be installed on your system. The next step is to install the *OntoRepair* plugin and some files required to demonstrate the functionality of the plugin.

Step 3: Download Required Files for OntoRepair

Download *OntoRepair*, Pellet and the sample ontologies located under the "links" section at the following url: `http://krr.meraka.org.za/people/kmoodley` (see Figure A.3).

Important: As mentioned in Chapter 5, Protégé allows one to install DL reasoners as

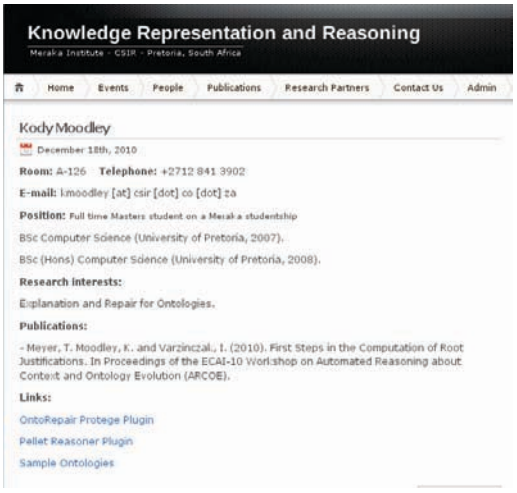

Figure A.3: Step 3: Download *OntoRepair*

plugins in the system to reason with loaded ontologies. Protégé 4.0.2 comes pre-bundled with the FaCT++ DL reasoner. However, *OntoRepair* is not compatible with this reasoner plugin due to an issue with its Java wrapper class for Protégé. *OntoRepair* is still compatible with other reasoners which do not have this issue such as Pellet and *Hermit*. For this reason, we download Pellet in this step.

Step 4: Install Plugins

Now we have to install the downloaded plugins from step 3 and place the sample ontologies in an appropriate location. The sample ontologies should be extracted to any convenient location in your system. Most importantly, the *OntoRepair* (org.meraka.ontorepair.jar) and Pellet (com.clarkparsia.protege.plugin.pellet.jar) plugins should be placed inside the **plugins** folder of your Protégé 4.0.2 installation directory ("/Protege_4.0.2/plugins/" is the default directory).

A.1.3 Getting Started with OntoRepair

Protégé 4.0.2, Pellet and *OntoRepair* should now be fully installed. We now illustrate how to use *OntoRepair* with an example.

Step 5: Fire up Protégé

Start up Protégé with the relevant executable or shell script (depending on your platform) found in the installation directory of Protégé 4.0.2. The Protégé welcome screen should be displayed as in Figure A.4.

Now select the second option on this screen "Open OWL ontology". In the file dialog, browse to the directory in which you saved the sample ontologies. Select the ontology file "Tambis-patched.owl" and click on "Ok". Note that you can also load the Pizza ("Pizza.owl") and Travel ("travel.owl") sample ontologies in the same fashion. Protégé will start up and it will load the Tambis ontology. You can now familiarize yourself with the Protégé

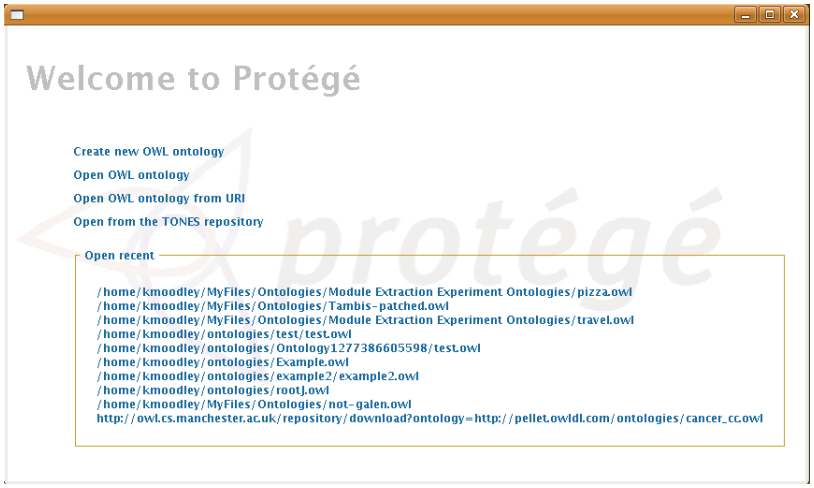

Figure A.4: Step 5: Start up Protégé

interface. The "Active Ontology" tab which is the default view, displays some ontology metrics (meta information about the ontology).

Step 6: Using OntoRepair

You should now be somewhat familiar with the Protégé interface and we can now start and use *OntoRepair*. Click on the "Tabs" menu in the main toolbar of Protégé (see Figure A.5) and select "OntoRepair".

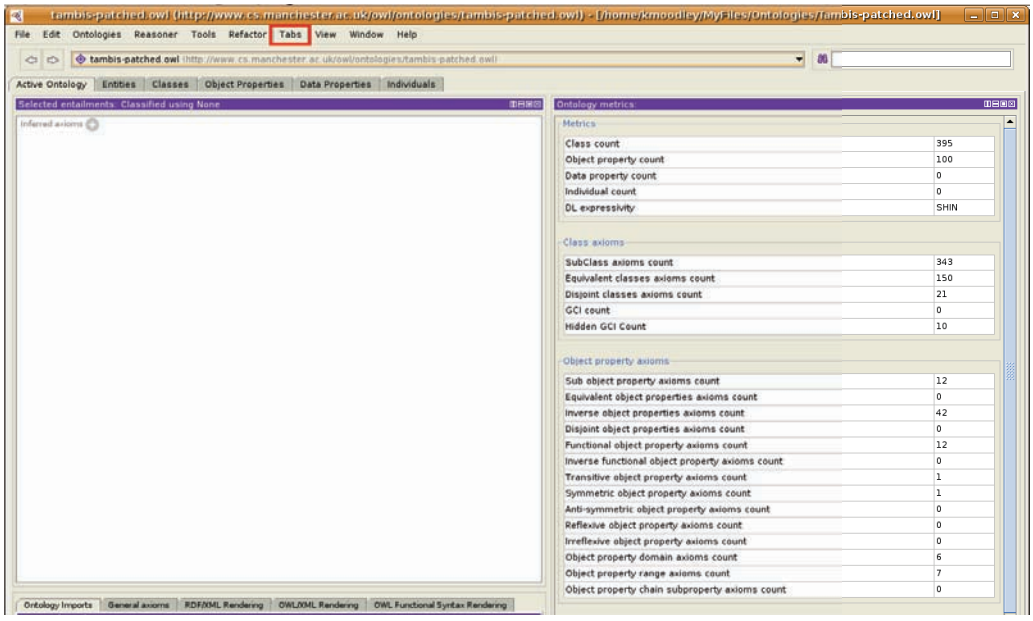

Figure A.5: Step 6: Tabs Menu

The interface for the *OntoRepair* tab is shown in Figure A.6.

After selecting the *OntoRepair* tab, pick a reasoner from the "Reasoner" menu in the main toolbar.

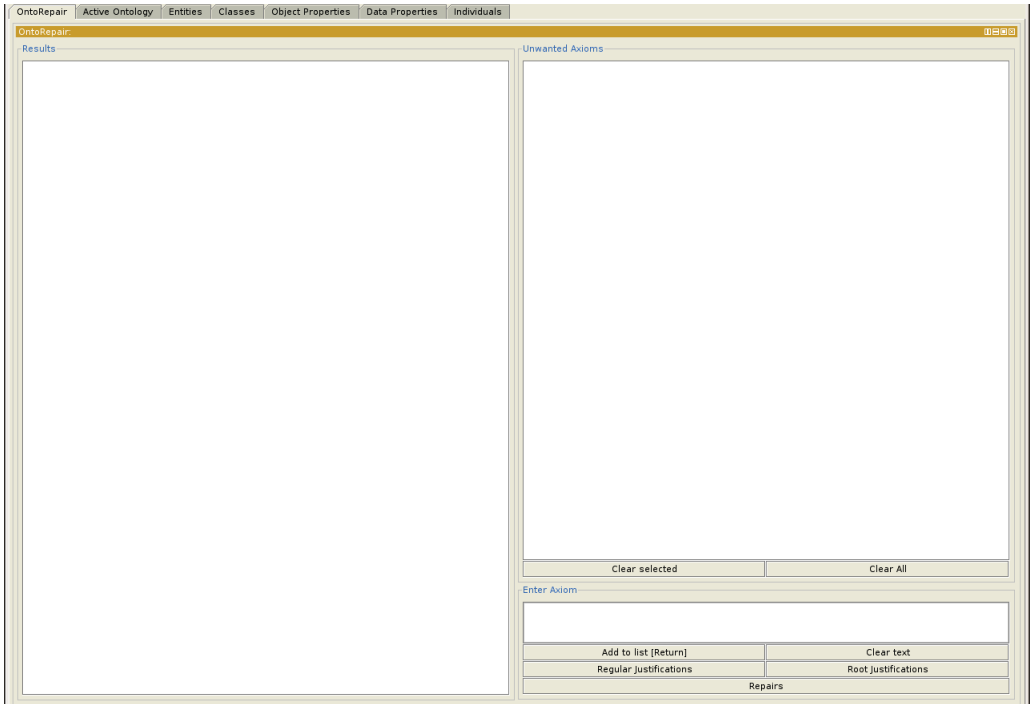

Figure A.6: Step 6: *OntoRepair* Interface

Important: You should have two reasoners installed namely FaCT++ and Pellet. Recall that FaCT++ is not compatible with *OntoRepair* and therefore you should only select Pellet.

After selecting a reasoner, Tambis will be classified (see Section 2.3.3 in Chapter 2). This might take a few seconds depending on your system. Once the ontology is classified, you can add the following axioms to the "Unwanted axioms" list in *OntoRepair*:

- metal SubClassOf Nothing
- nonmetal SubClassOf Nothing
- nickel SubClassOf Nothing

Make sure to type in these axioms verbatim because *OntoRepair* is strict on syntax. Note that the Protégé syntax corresponds closely with the OWL syntax and in turn some applicable OWL constructs correspond with DL constructs (see Table 5.1). Therefore the above "Protégé-OWL" axioms correspond to the following DL axioms:

- $\mathsf{Metal} \sqsubseteq \bot$
- $\mathsf{Nonmetal} \sqsubseteq \bot$
- $\mathsf{Nickel} \sqsubseteq \bot$

After adding these three axioms to the "Unwanted axioms" list (and making sure that we have selected Pellet in the Reasoner menu), we can compute the: regular justifications for each of these axioms, the root justifications for the entire list and the repairs for the entire list by clicking on the appropriate button in *OntoRepair*. First try computing the "Regular

justifications". The results that are displayed in the results pane should correspond to Figure A.7.

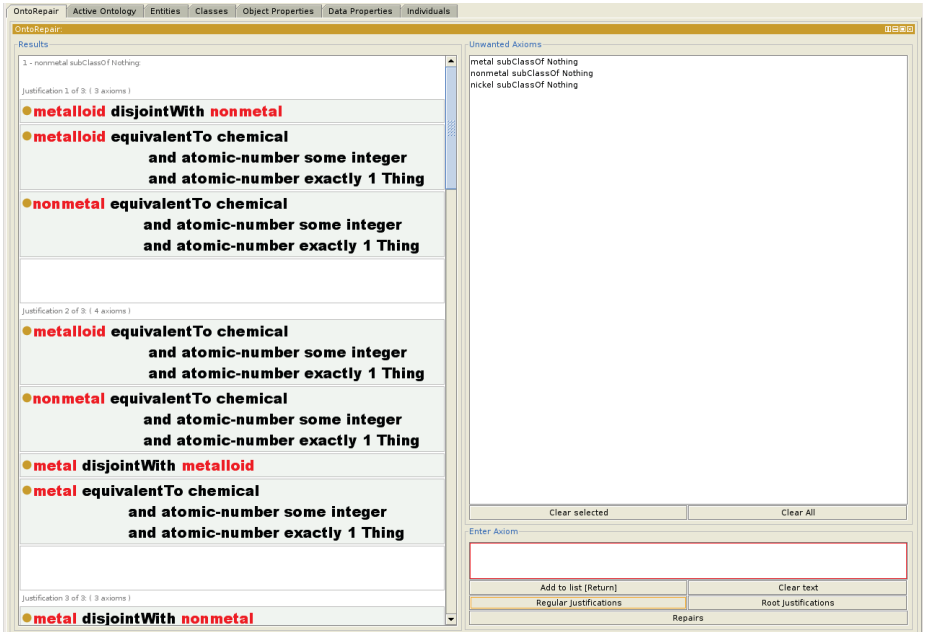

Figure A.7: Step 6: Computed regular justifications

From the results in Figure A.7 we can see that each of the unwanted axioms in the list has three justifications. This brings the total number of regular justifications computed to nine. Now try computing the "Root justifications". The results should reflect those shown in Figure A.8.

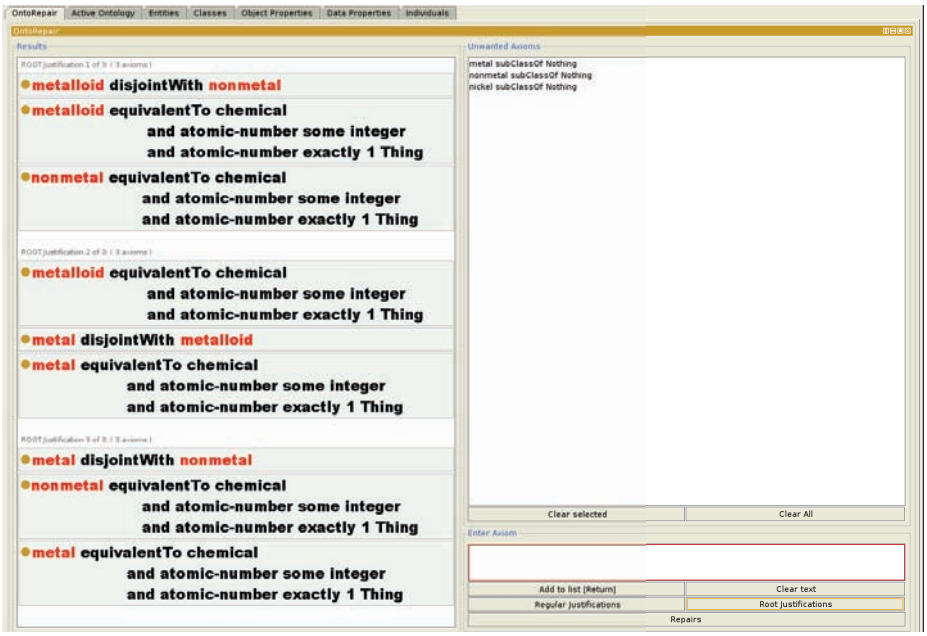

Figure A.8: Step 6: Computed root justifications

From Figure A.8 we observe that the total number of root justifications for the list is only three. Finally, to get the repairs for the list of unwanted axioms we select the "Repairs" button. These repairs are depicted in Figure A.9.

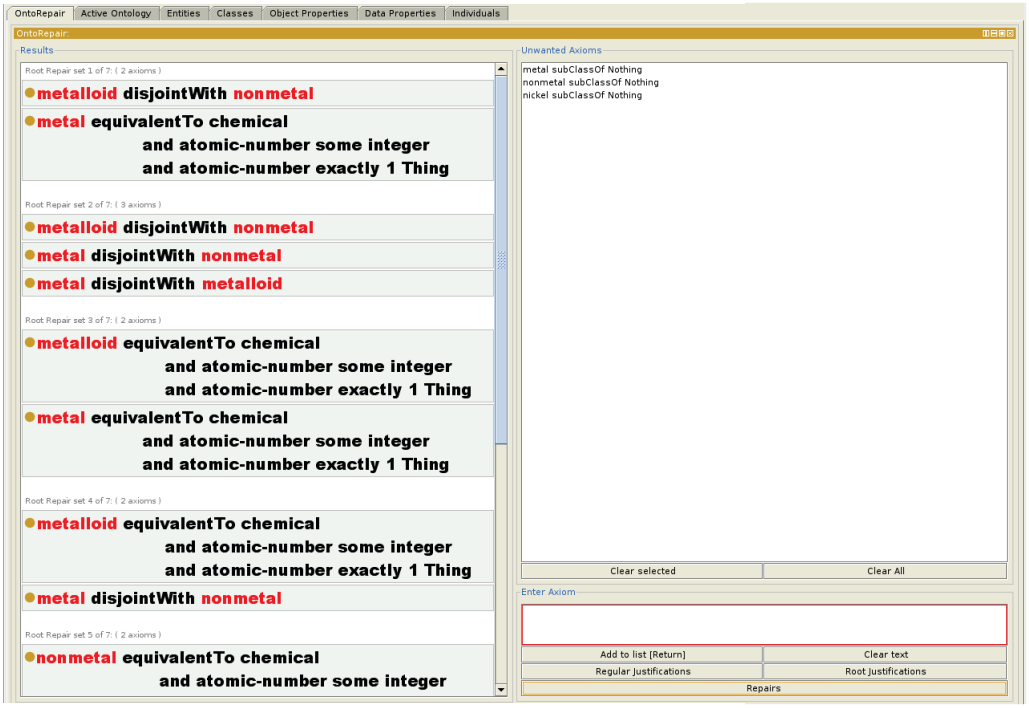

Figure A.9: Step 6: Computed repairs

Each of the seven repairs computed for this unwanted axiom list is such that if the axioms in this repair are removed from Tambis, then all the unwanted axioms in the list will be eliminated from Tambis.

To remove an axiom we have to first locate the axiom in either the entities/classes tab in Protégé. For example, to remove the axiom "metalloid disjointWith metal", we have to go to the concept description for either "metal" or "metalloid" and remove "metal" or "metalloid" from the Disjoint Classes section (see Figure A.10). In *OntoRepair*, to go to the concept description of a specific concept name, one can click on the concept name in the axiom in the results pane.

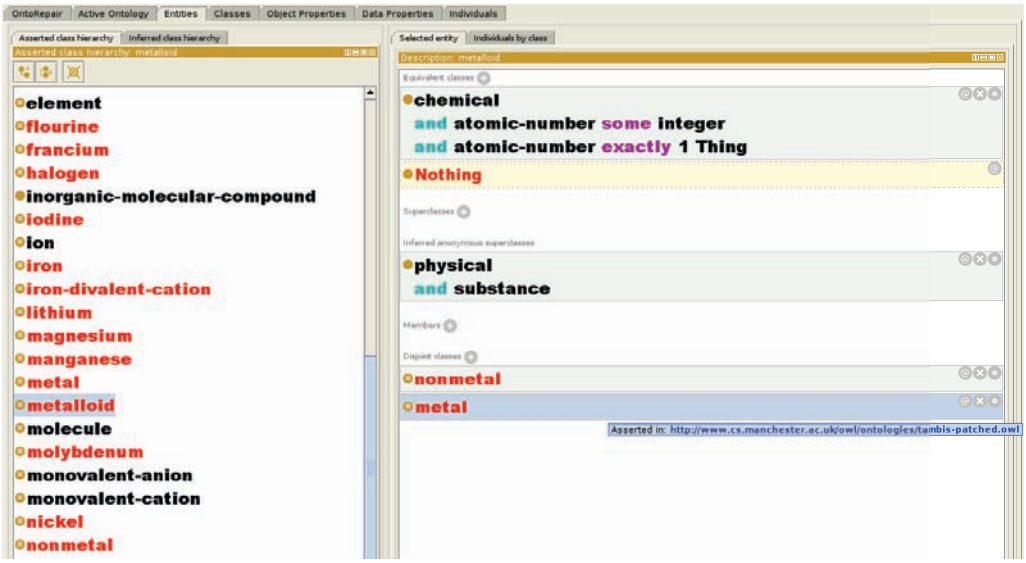

Figure A.10: Step 6: Apply Repair

Thus we have presented a mini-tutorial for getting up and running with Protégé and *OntoRepair*. For a more comprehensive guide on *modelling* OWL ontologies in Protégé (and getting more familiar with the Protégé interface) see the tutorial by Matthew Horridge, which can be found at the following url: `http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial`.

Bibliography

Artale, A., Franconi, E., Guarino, N., and Pazzi, L. (1996). Part-whole relations in object-centered systems: An overview. *Data and Knowledge Engineering*, 20(3):347–383.

Baader, F. (2003). Terminological cycles in a description logic with existential restrictions. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, volume 18, pages 325–330.

Baader, F., Brandt, S., and Lutz, C. (2005a). Pushing the $\mathcal{EL}$ envelope. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, volume 19, pages 364–369.

Baader, F., Buchheit, M., and Hollander, B. (1996). Cardinality restrictions on concepts. *Artificial Intelligence*, 88(1-2):195–213.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Baader, F., Hladik, J., and Peñaloza, R. (2008). Automata can show PSPACE results for description logics. *Information and Computation*, 206(9-10):1045–1056.

Baader, F. and Hollunder, B. (1995). Embedding defaults into terminological knowledge representation formalisms. *Journal of Automated Reasoning*, 14:149–180.

Baader, F., Küsters, R., and Molitor, R. (2000). Rewriting Concepts Using Terminologies. *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 297–310.

Baader, F., Milicic, M., Lutz, C., Sattler, U., and Wolter, F. (2005b). Integrating description logics and action formalisms for reasoning about web services. LTCS-Report LTCS-05-02, Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology.

Baader, F., Peñaloza, R., and Suntisrivaraporn, B. (2007). Pinpointing in the Description Logic $\mathcal{EL}$. In *Proceedings of the Thirtieth Annual German Conference on Artificial Intelligence (KI)*, pages 52–67. Springer-Verlag.

Baader, F. and Sattler, U. (2001). An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40.

Baader, F. and Suntisrivaraporn, B. (2008). Debugging SNOMED CT using axiom pinpointing in the description logic $\mathcal{EL}+$. In *Proceedings of the Third International Conference on Knowledge Representation in Medicine (KR-MED)*. http://www.kr-med.org/2008.

Baker, P. G., Goble, C. A., Bechhofer, S., Paton, N. W., Stevens, R., and Brass, A. (1999). An ontology for bioinformatics applications. *Bioinformatics*, 15(6):510–520.

Berners-Lee, T., Hendler, J., and Lassila, O. (May, 2001). The semantic web. In *Scientific American*. `http://www.sciam.com/2001/0501issue/0501berners-lee.html`.

Bobrow, D. G. and Winograd, T. (1977). An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1):3–46.

Brachman, R. J. (1983). What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer*, 16(10):30–36.

Brachman, R. J. and Schmolze, J. G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216.

Brandt, S., Küsters, R., and Turhan, A. (2002). Approximation and difference in description logics. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR)*. Morgan Kaufmann. http://www.kr.org/KR2002.

Calvanese, D., Lembo, D., Lenzerini, M., Poggi, A., and Rosati, A. (2007). Ontology-based database access. In *Proceedings of the Fifteenth Italian Conference on Database Systems (SEBD)*. http://www.sebd.org/2007/index.html.

Consortium, T. G. O. (2000). Gene ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29.

Cuenca-Grau, B., Horrocks, I., Kazakov, Y., and Sattler, U. (2007). Just the right amount: extracting modules from ontologies. In *Proceedings of the Sixteenth International World Wide Web Conference (WWW)*, pages 717–726. ACM.

Cuenca-Grau, B., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P. F., and Sattler, U. (2008). OWL 2: The next step for OWL. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):309–322.

Cuenca-Grau, B., Parsia, B., Sirin, E., and Kalyanpur, A. (2006). Modularity and web ontologies. In Doherty, P., Mylopoulos, J., and Welty, C., editors, *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 198–208. Morgan Kaufmann.

de la Banda, M. G., Stuckey, P. J., and Wazny, J. (2003). Finding all minimal unsatisfiable subsets. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP, pages 32–43. ACM.

Doran, P., Tamma, V., and Iannone, L. (2007). Ontology module extraction for ontology reuse: an ontology engineering perspective. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management (CIKM)*, pages 61–70. ACM.

Fikes, R. and Kehler, T. (1985). The role of frame-based representation in reasoning. *Communications of the Associaton for Computing Machinery (CACM)*, 28(9):904–920.

Herzig, A. and Varzinczak, I. (2006). A modularity approach for a fragment of $\mathcal{ALC}$. In Fisher, M., van der Hoek, W., Konev, B., and Lisitsa, A., editors, *Proceedings of the Tenth European Conference on Logics in Artificial Intelligence (JELIA)*, number 4160 in LNAI, pages 216–228. Springer-Verlag.

Hladik, J. (2004). A tableau system for the Description Logic $\mathcal{SHIO}$. In *Proceedings of the Second International Joint Conference on Automated Reasoning (IJCAR)*.

Horn, A. (1951). On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21.

Horridge, M. and Bechhofer, S. (2010). The OWL API: A Java API for OWL ontologies. *Semantic Web Journal*, pages 1–11. `http://www.semantic-web-journal.net/sites/default/files/swj107_2.pdf`.

Horridge, M., Parsia, B., and Sattler, U. (2008). Laconic and precise justifications in OWL. In *Proceedings of the Seventh International Semantic Web Conference (ISWC)*, volume 5318 of *Lecture Notes in Computer Science*, pages 323–338. Springer-Verlag.

Horridge, M., Parsia, B., and Sattler, U. (2009a). Explaining inconsistencies in OWL ontologies. In *Proceedings of the Third International Conference on Scalable Uncertainty Management (SUM)*, pages 124–137. Springer-Verlag.

Horridge, M., Parsia, B., and Sattler, U. (2009b). Lemmas for justifications in OWL. In *Proceedings of the Twenty Second International Workshop on Description Logics (DL)*. http://web.comlab.ox.ac.uk/DL2009.

Horrocks, I. (2005). Owl: A description logic based ontology language. In *Logic Programming*, volume 3668, pages 1–4. Springer-Verlag.

Horrocks, I., Kutz, O., and Sattler, U. (2006). The even more irresistible $\mathcal{SROIQ}$. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR)*. http://www.iccl.tu-dresden.de/announce/KR2006.

Horrocks, I. and Patel-Schneider, P. F. (2004). Reducing OWL entailment to description logic satisfiability. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):345–357.

Horrocks, I. and Sattler, U. (1999). A description logic with transitive and inverse roles and role hierarchies. *Journal of Logic and Computation*, 9(3):385–410.

Horrocks, I. and Sattler, U. (2004). Decidability of $\mathcal{SHIQ}$ with complex role inclusion axioms. *Artificial Intelligence*, 160(1–2):79–104.

Horrocks, I., Sattler, U., and Tobies, S. (1998). A PSPACE-algorithm for deciding $\mathcal{ALCI}_{R^+}$-satisfiability. LTCS-Report 98–08, LuFg Theoretical Computer Science, RWTH Aachen, Germany.

Horrocks, I., Sattler, U., and Tobies, S. (2000). Practical reasoning for very expressive description logics. *Logic Journal of IGPL*, 8(3):239–263.

Jimenez-Ruiz, E., Cuenca-Grau, B., Sattler, U., Schneider, T., and Berlanga, R. (2008). Safe and economic re-use of ontologies: A logic-based methodology and tool support. In *Proceedings of the Fifth Annual European Semantic Web Conference (ESWC)*, pages 185–199.

Kalyanpur, A. (2006). *Debugging and repair of OWL ontologies*. PhD thesis, University of Maryland.

Kalyanpur, A., Parsia, B., and Cuenca-Grau, B. (2006a). Beyond asserted axioms: Fine-grain justifications for OWL-DL entailments. In *Proceedings of the Nineteenth International Workshop on Description Logics (DL)*. http://dl.kr.org/dl2006.

Kalyanpur, A., Parsia, B., Horridge, M., and Sirin, E. (2007). Finding all justifications of OWL DL entailments. In *Proceedings of the Sixth International Semantic Web Conference (ISWC)*, pages 267–280.

Kalyanpur, A., Parsia, B., Sirin, E., and Cuenca-Grau, B. (2006b). Repairing Unsatisfiable Concepts in OWL Ontologies. In Sure, Y. and Domingue, J., editors, *The Semantic Web: Research and Applications*, volume 4011 of *Lecture Notes in Computer Science*, pages 170–184. Springer-Verlag.

Kalyanpur, A., Parsia, B., Sirin, E., Cuenca-Grau, B., and Hendler, J. (2005a). SWOOP: A web ontology editing browser. *Journal of Web Semantics*.

Kalyanpur, A., Parsia, B., Sirin, E., and Hendler, J. (2005b). Debugging unsatisfiable classes in OWL ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(4):268–293.

Keet, C. M. (2008). *A Formal Theory of Granularity*. PhD thesis, Free University of Bozen-Bolzano, Italy.

Kleene, S. C. (1968). *Mathematical Logic*. Wiley.

Knublauch, H., Fergerson, R. W., Noy, N. F., and Musen, M. A. (2004). The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In *Proceedings of the Third International Semantic Web Conference (ISWC)*, volume 3298 of *Lecture Notes in Computer Science*. http://iswc2004.semanticweb.org.

Konev, B., Lutz, C., Walther, D., and Wolter, F. (2008). Semantic modularity and module extraction in description logics. In *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI)*, pages 55–59. IOS Press.

Kremen, P. and Kouba, Z. (2009). Incremental approach to error explanations in ontologies. In Pellegrini, T., Auer, S., Tochtermann, K., and Schaffert, S., editors, *Networked Knowledge: Networked Media*, volume 221 of *Studies in Computational Intelligence*, pages 171–185. Springer-Verlag.

Lam, J., Sleeman, D., Pan, J., and Vasconcelos, W. (2008). A fine-grained approach to resolving unsatisfiable ontologies. In Spaccapietra, S., editor, *Journal on Data Semantics X*, volume 4900 of *Lecture Notes in Computer Science*, pages 62–95. Springer-Verlag.

Lassila, O. and Swick, R. R. (1998). Resource description framework (rdf) model and syntax.

Levesque, H. J. and Brachman, R. J. (1985). A fundamental tradeoff in knowledge representation and reasoning. *Readings in Knowledge Representation*, pages 42–70.

Levesque, H. J. and Brachman, R. J. (1987). Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3:78–93.

McGuinness, D. L., Fikes, R., Rice, J., and Wilder, S. (2000). The chimaera ontology environment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI)*. http://www.aaai.org/Conferences/AAAI/aaai00.php.

Meyer, T., Lee, K., Booth, R., and Pan, J. Z. (2006). Finding maximally satisfiable terminologies for the description logic $\mathcal{ALC}$. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI)*. AAAI press. http://www.aaai.org/Conferences/AAAI/aaai06.php.

Meyer, T., Moodley, K., and Varzinczak, I. (2010). First steps in the computation of root justifications. In *Proceedings of the ECAI-10 Workshop on Automated Reasoning about Context and Ontology Evolution (ARCOE)*. http://www.arcoe.org/2010.

Motik, B., Patel-Schneider, P. F., and Parsia, B. (2009). OWL 2 Web Ontology Language structural specification and functional style syntax. In *W3C Recommendation, W3C World Wide Web Consortium*.

Nebel, B. (1990). Reasoning and revision in hybrid representation systems. In *Lecture Notes in Artificial Intelligence*, volume 422. Springer-Verlag.

Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Rosati, R. (2008). Journal on data semantics x. chapter Linking data to ontologies, pages 133–173. Springer-Verlag.

Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns. In *Engineering Knowledge in the Age of the SemanticWeb*, volume 3257 of *Lecture Notes in Computer Science*, pages 63–81. Springer-Verlag.

Rector, A. and Horrocks, I. (1997). Experience building a large, re-usable medical ontology using a description logic with transitivity and concept inclusions. In *Proceedings of the Workshop on Ontological Engineering, AAAI Spring Symposium*. AAAI Press. http://www.aaai.org/Press/Proceedings/aaai97.php.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95.

Rumbaugh, J., Jacobson, I., and Booch, G. (1997). *The Unified Modeling Language user guide*. Addison-Wesley.

Schaerf, A. (1994). Reasoning with individuals in concept languages. *Data Knowledge Engineering*, 13(2):141–176.

Schlobach, S. and Cornet, R. (2003). Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 355–360. Morgan Kaufmann.

Schmidt-Schauß, M. and Smolka, G. (1991). Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26.

Shearer, R., Motik, B., and Horrocks, I. (2008). Hermit: A highly-efficient OWL reasoner. In *Proceedings of the Fifth International Workshop on OWL: Experiences and Directions (OWLED)*. http://www.webont.org/owled/2008.

Sirin, E., Parsia, B., Cuenca-Grau, B., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2).

Sowa, J. F. (1976). Conceptual graphs for a database interface. *IBM Journal of Research and Development*, 20(4):336–357.

Sowa, J. F. (1991). *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann.

Suntisrivaraporn, B. (2008). Module extraction and incremental classification: A pragmatic approach for $\mathcal{EL}+$ ontologies. In *Proceedings of the Fifth Annual European Semantic Web Conference (ESWC)*, pages 230–244.

Suntisrivaraporn, B., Qi, G., Ji, Q., and Haase, P. (2008). A modularization-based approach to finding all justifications for OWL DL entailments. In *Proceedings of the Third Asian Semantic Web Conference (ASWC)*, pages 1–15. Springer-Verlag.

Tobies, S. (2000). The complexity of reasoning with cardinality restrictions and nominals in expressive description logics. *Journal of Artificial Intelligence Research (JAIR)*, 12:199–217.

Tobies, S. (2001). *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH Aachen.

Tsarkov, D. and Horrocks, I. (2006). FaCT++ Description Logic Reasoner: System description. In *Proceedings of the Third International Joint Conference on Automated Reasoning (IJCAR)*, pages 292–297. Springer-Verlag.

Varzi, A. C. (1996). Parts, wholes, and part-whole relations: The prospects of mereotopology. *Data and Knowledge Engineering*, 20(3):259–286.

Varzinczak, I. (2008). Action theory contraction and minimal change. In Lang, J. and Brewka, G., editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 651–661. AAAI Press/MIT Press.

Wang, H., Horridge, M., Rector, A., Drummond, N., and Seidenberg, J. (2005). Debugging OWL-DL ontologies: A heuristic approach. In *Proceedings of the Fourth International Semantic Web Conference (ISWC)*, pages 745–757. Springer-Verlag.

Winston, M. E., Chaffin, R., and Herrmann, D. (1987). A taxonomy of part-whole relations. *Cognitive Science*, 11(4):417–444.

Woods, W. A. (1975). What's in a link: Foundations for semantic networks. *Representation and Understanding: Studies in Cognitive Science*, pages 35–82.