ABSTRACT

Title of dissertation: DEBUGGING AND REPAIR
OF OWL ONTOLOGIES

Aditya Kalyanpur, Doctor of Philosophy, 2006

Dissertation directed by: Professor James Hendler
Department of Computer Science

With the advent of Semantic Web languages such as OWL (Web Ontology Language), the expressive Description Logic $\mathcal{SHOIN}$ is exposed to a wider audience of ontology users and developers. As an increasingly large number of OWL ontologies become available on the Semantic Web and the descriptions in the ontologies become more complicated, finding the cause of errors becomes an extremely hard task even for experts. The problem is worse for newcomers to OWL who have little or no experience with DL-based knowledge representation. Existing ontology development environments, in conjunction with a reasoner, provide some limited debugging support, however this is restricted to merely reporting errors in the ontology, whereas bug diagnosis and resolution is usually left to the user.

In this thesis, I present a complete end-to-end framework for explaining, pinpointing and repairing semantic defects in OWL-DL ontologies (or in other words, a $\mathcal{SHOIN}$ knowledge base). Semantic defects are logical contradictions that manifest as either *inconsistent* ontologies or *unsatisfiable* concepts. Where possible, I show extensions to handle related defects such as unsatisfiable roles, unintended entailments and non-entailments, or defects in OWL ontologies that fall outside the DL scope (OWL-Full).

The main contributions of the thesis include:

- Definition of three novel OWL-DL debugging/repair services: *Axiom Pinpointing*, *Root Error Pinpointing* and *Ontology Repair*. This includes formalizing the notion of *precise justifications* for arbitrary OWL entailments (used to identify the cause of the error), *root/derived* unsatisfiable concepts (used to prune the error space) and *semantic/syntactic* relevance of axioms (used to rank erroneous axioms).
- Design and Analysis of decision procedures (both *glass-box* or reasoner dependent, and *black-box* or reasoner independent) for implementing the services
- Performance and Usability evaluation of the services on realistic OWL-DL ontologies, which demonstrate it's practical use and significance for OWL ontology modelers and users

DEBUGGING AND REPAIR OF OWL ONTOLOGIES

by

Aditya Kalyanpur

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Commmittee:

Professor James Hendler
Professor Ashok Agrawala
Professor Mark Austin
Professor Ian Horrocks
Dr. Ryusuke Masuoka

ACKNOWLEDGMENTS

First, I would like to thank my advisor, James Hendler, for his continuous support, encouragement and guidance over the years and for providing me the freedom to pursue my interests and goals. I am also grateful to my committee members Ashok Agrawala, Mark Austin, Ian Horrocks and Ryusuke Masuoka for their assistance and support, and a special thanks to Ian for his detailed comments and suggestions.

I would like to especially thank Bijan Parsia for his endless help in all aspects of my research. Besides being a tireless supporter of my work, and providing plenty of insight, ideas and suggestions for improvement, Bijan has acted as my mentor as well, pushing me to become a better researcher. On a similar note, I am also very grateful to Evren Sirin and Bernardo Cuenca Grau with whom I have shared numerous helpful discussions and have learnt a great deal from.

I have also received a lot of support from my MINDSWAP group members and I am grateful for all their efforts. Thanks to Jennifer Golbeck, Ron and Amy Alford, David Taowei Wang, Christian Halaschek, Vladimir Kolovski, Yarden Katz and others I may have forgotten. It has been a pleasure working with all of you.

On the personal front, I am eternally grateful to my parents, Anand and Vidya Kalyanpur, for their endless love and support in every step of life, and for providing me with all the opportunities in the world to excel. I would also like to thank my brother, Trushant Kalyanpur, who has been one of my closest friends and biggest supporters over the years. Finally, a very special thanks to my partner, Priya Joshi, who has been by my side throughout the duration of my thesis, and whose tremendous love, care and patience has been instrumental in the completion of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction and Overview
1.1 Introduction

1.1.1 Semantic Web and OWL

The Semantic Web [12], [19] is an extension of the current World Wide Web in which information is given precise meaning, making it easy to exchange, integrate and process data in a systematic, machine-automated manner. Using standardized languages, published as World Wide Web Consortium (W3C) recommendations, semantic web data can explicitly describe the knowledge content underlying HTML pages, specify the implicit information contained in media like images and videos, or be a publicly accessible and usable representation of an otherwise inaccessible database.

The standardized languages which are the basis of the Semantic Web form a layered stack, at the bottom of which lies the Resource Description Framework (RDF) [66]. RDF is a simple assertional language that is designed to represent information in the form of triples, i.e., statements of the form: subject, predicate, object. RDF predicates may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. RDF however, contains no mechanisms for describing these predicates, nor does it support description of relationships between predicates and other resources. This is provided by the RDF vocabulary description language, RDF Schema (RDFS [17]). RDFS allows the specification of classes (generalized categories or unary relations) and properties (predicates or binary relations), which can typically be arranged in a simple taxonomy (hierarchy). In addition, it allows simple typing of properties by imposing constraints on its domain and range.

The Web Ontology Language (OWL) [27], released as a W3C recommendation in February 2004, is an expressive ontology language that is layered on top of RDF and RDFS. OWL can be used to define classes and properties as in RDFS, but in addition, it provides a rich set of constructs to create new class descriptions as logical combinations (intersections, unions, or complements) of other classes; define value and cardinality restrictions on properties (e.g., a restriction on a class to have only one value for a particular property) and so on.

OWL is unique in that it is the first ontology language whose design is based on the Web architecture, i.e., it is open (non-proprietary); it uses Universal Resource Identifiers (URIs) to unambiguously identify resources on the Web (similar to RDF and RDFS); it supports the linking of terms across ontologies making it possible to cross-reference and reuse information; and it has an XML syntax (RDF/XML) for easy data exchange.

One of the main benefits of OWL is the support for automated reasoning, and to this effect, it has a formal semantics based on *Description Logics* (DL). DLs are typically a decidable subset of First Order Logic (FOL)[1], being restricted to the 2-variable fragment

[1]There have been DLs considered which are not strict subsets of FOL. For example, DLs have been

of FOL (L2) and including counting quantifiers (thereby corresponding to the logic C2), and are formalisms tailored towards Knowledge Representation (KR) [3], i.e., they are suitable for representing structured information about concepts, concept hierarchies and relationships between concepts. The decidability of the logic ensures that sound and complete DL reasoners can be built to check the consistency of an OWL ontology, i.e., verify whether there are any logical contradictions in the ontology axioms. Furthermore, reasoners can be used to derive inferences from the asserted information, e.g., infer whether a particular concept in an ontology is a subconcept of another (a.k.a. *concept classification*), or whether a particular individual in an ontology belongs to a specific class (a.k.a. *realization*). Popular existing DL reasoners in the OWL community include Pellet [97], FaCT [50] and RACER [104].

In addition to reasoners, numerous OWL ontology browsers/editors such as Protege [76], KAON [78] and Swoop [57] have been built to aid in the design and construction of OWL ontology models. The latter - Swoop - has been developed as part of this dissertation. Most of these OWL tools have expanded their functionality beyond basic editing to include features such as change management and query handling, and in a lot of cases included a reasoner for consistency checking of the ontology. For example, Swoop has integrated Pellet for reasoning and additionally provides the ability to automatically partition, collaboratively annotate and version control OWL ontologies.

1.1.2 Motivation: Lack of OWL Debugging Support

While OWL tools have focused on various aspects of ontology engineering, the support for *debugging* defects in OWL ontologies has been fairly weak. Common defects include *inconsistent* ontologies and *unsatisfiable* concepts. An unsatisfiable concept is one that cannot possibly have any instances, i.e., it represents the empty set (and is equivalent to the bottom concept or in the OWL language, `owl:Nothing`). Both these errors, inconsistent ontologies and unsatisfiable concepts, signify logical contradictions in the ontology and can be detected automatically using a DL reasoner. However, reasoners simply report the errors, without explaining *why* the error occurs or *how* it can be resolved correctly.

For example, consider the case of the Tambis OWL ontology, a biological science ontology developed by the TAMBIS[2] project. As shown in Figure 1.1, more than a third of the classes in the ontology are unsatisfiable:

Here, the tool has exposed the errors in the ontology, though understanding their cause and arriving at a repair solution is left to the user. Also, the fact that there are so many errors makes the debugging task seem all the more overwhelming.

When modelers encounter cases such as this, they are often at a loss at what to do. This also has a negative general consequence which inhibits the adoption and effective use of OWL – namely, ontology authors (especially newcomers to OWL) tend to *underspecify* their models to "avoid" errors. Typically, this is done by getting rid of negation in the ontology since contradictions mainly arise due to it. For example, in the Tambis OWL

enriched with the epistemic operator (K) in order to provide for nonmonotonic reasoning and procedural rules that cannot be characterized in a standard first-order framework.

[2]http://imgproj.cs.man.ac.uk/tambis/

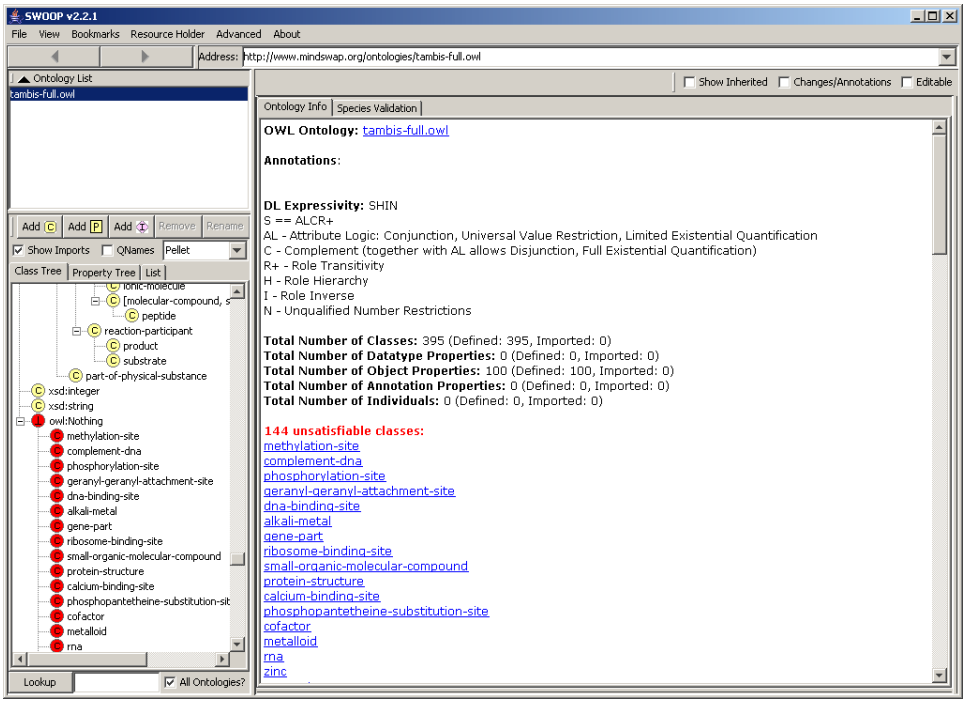

Figure 1.1: OWL version of the Tambis ontology as viewed in the Swoop editor and tested using the Pellet Reasoner

ontology, the unsatisfiable concepts metal and non-metal are defined to be disjoint from one another (using the owl : disjointWith construct), implying that an individual cannot be a member of both concepts at the same time. In this case, there is an inherent negation in the concept definitions, i.e., metal is a subclass of the negation of non-metal. Here, removing the disjointness between the two concepts eliminates numerous unsatisfiable concept errors in the ontology, though this is probably undesired.

Thus, it is evident that OWL ontology tools have to go much further in organizing and presenting the information supplied by the reasoner and existing in the ontology. For example, tools used to debug unsatisfiable classes in ontologies could pinpoint the problematic axioms in the ontology responsible for the errors. By highlighting the minimal set of axioms responsible for the error, the modeler is aware of a possible solution – edit or remove any one of the possibly erroneous axioms.

Similarly, when there are a large number of unsatisfiable concepts in an ontology (as is the case of the Tambis ontology seen earlier), tools can detect and highlight interdependencies between unsatisfiable classes to help differentiate the root bugs from the less critical ones, e.g., when a class is asserted to be a subclass of another unsatisfiable class, automatically rendering it unsatisfiable, we need to focus on the latter concept which is the actual source of the error.

Having found defects in the ontology, resolution can be non-trivial as well, requiring an exploration of remedies with a cost/benefit analysis. For example, one cost metric could be the impact on the ontology, in terms of the information lost, when a particular axiom is removed from it as part of the repair solution. In this case, one would like to gen-

erate repair solutions that impact the ontology minimally. Also, the non-local effects of axioms in an OWL ontology means modifications done to eliminate one inconsistency (by editing certain axioms) can cause additional inconsistencies to appear somewhere else in the ontology. Thus, particular care and effort must be taken to ensure that ontology repair is carried out efficiently.

The goal of this dissertation is to develop a set of services for OWL (DL) that cater towards debugging and repair, on the lines of the solutions mentioned above.

1.1.3 Defects in OWL

In this section, we briefly look at the various types of defects in OWL ontologies and discuss factors that make them susceptible to errors.

Broadly speaking, defects in OWL fall into three main categories:

- *Syntactic Defects*: Syntactic issues loom large in OWL for a number of reasons including the baroque exchange syntax, RDF/XML and the use of URIs (and their abbreviations). Hence, any non well-formed XML ontology document is syntactically incorrect.

  Additionally, the OWL language comes in three increasingly expressive sub-languages or "species" - OWL-Lite, OWL-DL and OWL-Full, and detecting which species an OWL document falls in is done strictly syntacticly, i.e., there are a number of restrictions imposed on the RDF graph form for it to count as an instance of a particular species. Thus, building an ontology that falls outside the desired species level can be considered as a syntactic defect.
- *Semantic (or Logical) Defects*: Given a syntactically correct OWL ontology, semantic defects are those which can be detected by an OWL reasoner. As noted earlier, these include unsatisfiable classes and inconsistent ontologies. For example, class A in an ontology is unsatisfiable if it is a subclass of both, class C and the complement of class C (defined in OWL using the `owl:complementOf` operator), since it implies a direct contradiction. On the other hand, if an ontology asserts that an unsatisfiable class contains an instance, the ontology itself is inconsistent.
- *Style Defects*: These are defects that are not necessarily invalid, syntactically or semantically, yet are discrepancies in the ontology or unanticipated results of modeling, which require the modelers' attention before use in a specific domain or application scenario. Examples include *unintended* inferences, and *unused* classes or properties with no reference anywhere else in the ontology.

We now discuss factors specific to the nature and design rationale of OWL, which makes it possible for errors to arise.

- *Difficulty in understanding modeling*: Note that OWL is based on an expressive DL and thus one of the main causes for errors, especially semantic errors, is the difficulty that comes from modeling accurately in an expressive and complex ontology language. OWL users and developers are not likely to have a lot of experience with description logic based KR, and without adequate tool support for training

and explanation, engineering ontologies can be a hard task for such users. As ontologies become larger and more complex, highly non-local interactions in the ontology (e.g., interaction between local class restrictions on properties and its global domain/range restrictions) make modeling, and analyzing the effects of modeling non-trivial even for domain experts.

- *Interlinking of OWL Ontologies*: The idea behind Web ontology development is different from traditional and more controlled ontology engineering approaches which rely on high investment, relatively large, heavily engineered, mostly monolithic ontologies. For OWL ontologies, which are based on the Web architecture (characterized as being open, distributed and scalable), the emphasis is more on utilizing this *freeform* nature of the Web to develop and share (preferably smaller) ontology models in a relatively ad hoc manner, allowing ontological data to be reused easily, either by linking models (using the numerous mapping properties available in OWL) or merging them (using the `owl:imports` command).

  However, when related domain ontologies created by separate parties are merged using `owl:imports`, the combination can result in modeling errors. This could be due to ontology authors either having different views of the world, following alternate design paradigms, or simply, using a conflicting choice of modeling constructs. An example is when the two upper-level ontologies, CYC and SUMO are merged leading to a large number of unsatisfiable concepts due to disjointness statements present in CYC [92].
- *Migration to OWL*: Since OWL is a relatively new standard, one can expect that existing schema/ontologies in languages pre-dating OWL such as XML, DAML, KIF etc. will be migrated to OWL, either manually or using automated translation scripts. A faulty migration process can lead to an incorrect specification of concepts or individuals in the resultant OWL version. For example, the OWL version of the Tambis ontology seen earlier contains 144 unsatisfiable classes (out of 395) due to an error in the transformation script used in the conversion process.

1.1.4 Debugging OWL Defects

Depending on the type of defect as seen in the previous section, there are different ways to debug and resolve it. Syntactic defects are the easiest to fix, since most XML parsers (e.g. Xerces [3]) or RDF validators[4] directly pinpoint the line in the document (and the specific characters in it) which make the document syntactically invalid. Thus, by inspecting the exception log or trace, the ontology designer can easily fix such syntax errors.

For detecting which species an OWL document falls in, there exists specialized OWL *Species Validation* tools [5], which report the species level and the OWL language constructs used in the document, or the RDF graph constraints violated that force it to belong to a particular species. An interesting facility is provided by the Pellet [97] reasoner,

[3] http://xerces.apache.org/xerces-j/

[4] RDF Validator: http://www.w3.org/RDF/Validator/

[5] OWL Validator: http://phoebus.cs.man.ac.uk:9999/OWL/Validator

which in addition to species validation, incorporates a number of heuristics to detect "DL-izable" OWL-Full documents in order to repair them. The heuristics implemented in Pellet attempt to guess the correct type for an untyped resource, e.g., a resource used in the predicate position is inferred to be a property. Using this feature, a user can automatically add a set of triples to the document to bring it to the desired species level.

For style defects, effective debugging requires the expression of intent to the system since defectiveness here is very dependent on the modeler's intent. Thus, *testing* and test cases form the right modality for dealing with some of these defects. There exist simple lint-like debugging tools such as Chimaera [70], which are helpful for identifying some style discrepancies in the KB (such as cycles in class definitions) but just as in the case of syntactic defects, exposing the "bug" here is usually a direct pointer to the solution.

The hardest defects to debug and resolve correctly are semantic defects, just as logical errors in programs are hard to understand and fix. The problem is compounded by the fact that reasoners simply report them without providing any explanation. Thus, the main focus of this dissertation is on debugging and resolving semantic defects in OWL ontologies, and the goal is to formalize and build debugging services for them that are useful and understandable to ontology modelers.

1.2 Contributions

In this thesis, I have developed a complete end-to-end framework for debugging and repairing all types of semantic defects in OWL-DL ontologies. More specifically, I have,

- Designed and evaluated a novel DL explanation service: *Axiom Pinpointing*
  - Formalized the notion of *precise justifications* for arbitrary entailments in OWL ontologies
  - Devised a set of algorithms, both, *glass-box* or reasoner dependent, and *black-box* or reasoner independent, to find all the precise justifications (Axiom Pinpointing)
  - Analyzed the computational complexity of Axiom Pinpointing algorithms
  - Implemented the service in an OWL-DL reasoner (Pellet), performed a timing evaluation of the service on a set of OWL Ontologies and demonstrated that the service is practically feasible
  - Provided a UI for the service in the context of an OWL Ontology Engineering environment (Swoop) and performed a user-evaluation to test its effectiveness
- Designed and evaluated a novel DL debugging service: *Root Error Pinpointing*
  - Formalized the notion of *root* and *derived* unsatisfiable classes
  - Devised a set of glass and black box algorithms to separate root/derived errors
  - Analyzed the computational complexity of the algorithms and performed a timing evaluation of the service on ontologies containing a large number of unsatisfiable classes to demonstrate its significance

- Designed and evaluated a novel DL repair service: *Ontology Repair*
  - Formalized the notion of *semantic* and *syntactic* relevance in the context of axiom ranking
  - Devised algorithms to compute axiom ranks and subsequently generate repair solutions based on the ranks calculated. Modified the algorithm to incorporate axiom *rewrites* in the final solution.
  - Provided an interactive UI for the repair service and performed a user-evaluation to test its effectiveness

1.2.1 Scope and Limitations

The scope of this thesis is the debugging and repair of semantic defects in OWL-DL Ontologies. As noted earlier, semantic defects are mainly two types: unsatisfiable classes and inconsistent ontologies, which result due to logical contradictions present in the ontology.

Unsatisfiable roles, which are not as common as unsatisfiable classes, are easy to detect as well using the techniques developed. This is because roles correspond to two-place predicates in First Order Logic (FOL), while classes are one-place predicates. Thus, given a role R, the problem of checking the satisfiability of R reduces to the problem of checking the satisfiability of the class $(\geq 1.R)$.

Also, while the techniques are applicable for OWL-DL which is the known decidable sub-language of OWL, OWL-Full, which is the most expressive language of the OWL family, is also decidable under contextual (or π) semantics [73] with some additional constraints[6]. π semantics is essentially equivalent to the standard first-order semantics, wherein the role of a symbol can be inferred from its position in a formula, so the set of constant, function and predicate symbols need not be strictly disjoint. A DL reasoner can reason over certain OWL-Full ontologies successfully by enforcing π semantics. Thus, all the techniques described in this thesis for diagnosing and repairing contradictions in an ontology are directly applicable in the OWL-Full case (given the constraints) without any changes.

The techniques proposed in this thesis are of two types: *glass-box* or reasoner dependent, and *black-box* or reasoner independent. The black-box approach only relies on the availability of a sound and complete reasoner for a DL, and is thus not restricted to any particular logic. The glass-box algorithms are based on the expressive description logic $\mathcal{SHOIN}(\mathcal{D})$, which is the basis of the language OWL-DL.

Note that the debugging of syntactic and style defects in OWL is beyond the scope of the thesis. As mentioned earlier, there already exist tools providing support for such defects, whose resolution is either straightforward or strongly depends on the modeler's intent.

[6] Certain restrictions are required to yield a decidable logic, such as on simple roles in number restrictions [49]

1.3 Organization of Thesis

The thesis is organized as follows:

- In Chapter 2, we provide the formal background this work is grounded in. The chapter discusses the the Web Ontology Language (OWL), the World Wide Web Consortium (W3C) standard for creating ontologies on the Web; and briefly reviews the field of Description Logics (DLs), with emphasis on the expressive logic $\mathcal{SHOIN}(\mathcal{D})$ (which corresponds to the sub-language OWL-DL). Finally, it provides an overview of common reasoning services for description logics such as consistency checking, classification etc., and describes tableau-based decision procedures used to implement these services.
- In Chapter 3, we review other related approaches in existing logic-based systems such as logic programming systems, rule-based expert systems, deductive databases, automated theorem provers and finally description logic-based knowledge bases. We also look at two classical theories of diagnosis and revision, and describe the relation between these generic theories and the debugging/repair services devised in this thesis.
- Chapters 4-6 constitute the main contribution of this thesis. In Chapter 4, we describe the Axiom Pinpointing Service that is used to find (precise) justifications for arbitrary entailments in OWL-DL. Chapter 5 describes the Root Error Pinpointing Service, which can be used to separate the root or critical errors in the KB from the derived or dependent ones. Finally, Chapter 6 describes the Ontology Repair Service which generates repair solutions based on various criteria for ranking erroneous axioms.
- Chapter 7 discusses implementation details of the debugging and repair services formulated in Chapters 4-6, and presents results of performance and usability evaluations which demonstrate the practical significance of these services.
- Chapter 8 enumerates some of the open issues in our OWL debugging work and outlines areas for future research. The latter includes some preliminary ideas to deal with the problem of debugging non-subsumptions.
- Finally, the Appendix A discusses specific features in the OWL Ontology Editor, Swoop [57], that are tailored towards the understanding and analysis of OWL ontologies.

Chapter 2
Foundations
2.1 Description Logics

Description Logics (DL) [4], [21] are a family of logic-based knowledge representation formalisms. They are typically used to represent terminological knowledge of an application domain, where the data can be accessed and reasoned with in a principled manner. DLs are usually a (decidable) subset of First Order Predicate Logic (FOL), and thus have a well-defined, formal semantics.

The basic building blocks in DL are:

- **atomic concepts**: which correspond to 1-place (unary) predicates in FOL and denote a set or a class of objects, e.g., $Person(x)$, $Male(x)$.
- **atomic roles**: which correspond to 2-place (binary) predicates in FOL and denote relations between objects, e.g., $hasBrother(x, y)$.
- **individuals**: which correspond to constants in FOL, e.g., $Jack, John$ and denote objects in the domain

A DL provides a set of operators, called *constructors*, which allow to form complex concepts and roles from atomic ones. For example, by applying the concept conjunction constructor ($\sqcap$) on the atomic concepts $Person$ and $Male$, the set of all 'Male People' can be represented as follows: $Person \sqcap Male$.

The Boolean Concept Constructors are, apart from concept conjunction ($\sqcap$), concept disjunction ($\sqcup$) and concept negation ($\neg$). A Description Logic that provides, either implicitly or explicitly, all the boolean operators is called *propositionally closed*. DLs that are not propositionally closed are typically called *sub-boolean*. In this work, only propositionally closed DLs will be considered.

In addition to the booleans, DLs typically provide concept constructors that use roles to form complex concepts. The basic constructors of this kind are *existential* ($\exists$) and *universal* ($\forall$) restrictions operators, which represent restricted (guarded) forms of quantification. For example, we can describe a complex concept to denote fathers of only male children: $Father \sqcap \forall hasChild.Male$; or mothers who have at least one female child: $Mother \sqcap \exists hasChild.Female$.

Apart from concept and role constructors, which allow to define complex concepts and roles, a DL also provides means for representing axioms (logical sentences) involving concepts and roles. For example, we can specify a concept inclusion axiom of the form: $Father \sqsubseteq Person$, which states that a father is also a person.

Description Logic knowledge bases (KB) typically consist of:

- A **TBox** containing concept inclusion axioms of the form $C_1 \sqsubseteq C_2$, where both C_1, C_2 are concepts.
- A **RBox** containing role inclusion axioms of the form $R_1 \sqsubseteq R_2$ with R_1, R_2 roles.

- An **ABox** containing axioms of the form C(a), called concept assertions and R(a, b), called role assertions, where a, b are object names, R is a role and C a concept.

In its simplest form, a TBox consists of a restricted form of concept inclusion axioms called concept definitions: sentences of the form $A \sqsubseteq C$ or $A \equiv C$ (where A is atomic), which describe necessary or necessary and sufficient conditions respectively for objects to be members of A. Restricting a TBox to concept definitions, which are both *unique* (each atomic concept appears only once on the left hand side of a concept inclusion axiom) and *acyclic* (the right hand side of an axiom cannot refer, either directly or indirectly, to the name on its left hand side) greatly simplifies reasoning [51].

However, TBox axioms can also be used to describe more complex sentences, i.e., *general concept inclusion* axioms (GCIs). In a GCI of the form $C_1 \sqsubseteq C_2$, the concepts C_1, C_2 are not restricted to be atomic. GCIs are typically used to represent general constraints on the TBox, i.e. background knowledge. For example, the axiom: $Person \sqcap \exists hasChild.\top \sqsubseteq Father \sqcup Mother$ states that any person that has a child is either a father or a mother. Here $\top$ is used to denote the 'Top' concept which represents the universal set of all individuals in the domain (every concept in the KB is implicitly contained in $\top$).

Similar axioms can be used to represent assertions about roles in the RBox. For example, the role inclusion axiom : $hasSon \sqsubseteq hasChild$ states that the relation represented by $hasSon$ is contained in the relation represented by the role $hasChild$.

Finally, the ABox formalism provides means for instantiating concepts and roles. A concept assertion $C(a)$ states that the object a belongs to the concept C, e.g., the axiom $Father(Jack)$ states that Jack is a father; while a role assertion $R(a, b)$ is used to state that two objects a, b are related by a role R, e.g., the axiom $hasBrother(Jack, John)$ states that Jack and John are brothers.

The DL community has categorized various description logics by constructing mnemonic names that encode the precise expressivity of the particular logic. For a list of mnemonics with DL's they characterize, see Table 2.1.

Mnemonic	**DL Expressivity**
$\mathcal{AL}$	Attribute Logic [A, $\neg A$ (atomic), $C \sqcap D$, $\exists R.\top$, $\forall R.C$]
$\mathcal{ALC}$	Attribute Logic + Full Complement [allowing $C \sqcup D$ and $\exists R.C$]
R^+	Transitive Roles
$\mathcal{S}$	$ALCR^+$
$\mathcal{H}$	Role Hierarchy
$\mathcal{I}$	Inverse Roles
$\mathcal{O}$	Nominals (individuals used in class expressions)
$\mathcal{N}$	Unqualified Cardinality Restriction [$\geq nR$, $\leq nR$, $= nR$]
$\mathcal{Q}$	Qualified Cardinality Restriction [$(\geq nR).C$, $(\leq nR).C$, $(= nR).C$]
$\mathcal{D}$	Datatypes
$\mathcal{F}$	Functional Roles

Table 2.1: Mnemonics for DLs

The basic description logic that provides the boolean concept constructors plus the

existential and universal restriction constructors is called $\mathcal{ALC}$. Many applications require an expressive power beyond $\mathcal{ALC}$ and thus several DL extensions have been defined on top of it. For example, $\mathcal{ALC}$ allows GCIs in the TBox and concept and role assertions in the ABox, however, it provides no role constructors and disallows role inclusion axioms, hence forcing the RBox to be empty. The first obvious way for extending $\mathcal{ALC}$ is to provide new concept and role constructors.

A prominent example of concept constructors that are available in all modern DL systems are the so-called number restrictions [46] . In their most general form, number restrictions are called *qualified number restrictions*, which allow to build the complex concepts $\geq nR.C$ and $\leq nR.C$ from a role R, a natural number n and a concept C. Qualified number restrictions can be used to represent, for example, the women with less than two daughters: $Woman \sqcap \leq 2hasChild.Woman$.

Some DLs introduce a restricted form of number restrictions, called *unqualified* number restrictions that force the concept description C to be precisely the universal concept $\top$. Using unqualified number restrictions it is possible to describe, for example, the persons who have more than 10 friends: $Person \sqcap \geq 10.hasFriend$.

Finally, it is possible to restrict the expressivity of unqualified number restrictions by constraining the natural numbers that can be used in the constructor. In logics providing *functional number restrictions* (denoted by the mnemonic $\mathcal{F}$), the only number restriction operators allowed are ($\geq 2R$) and ($\leq 1R$). For example, a person with (strictly) more than one brother would be described by the following concept: $Person \sqcap \geq 2hasBrother$. The logic obtained from $\mathcal{ALC}$ by providing qualified number restrictions is called $\mathcal{ALCQ}$. On the other hand, adding unqualified and functional number restrictions to $\mathcal{ALC}$ results in the logics $\mathcal{ALCN}$ and $\mathcal{ALCF}$ respectively.

The *Nominal* constructor [48], [89] transforms the object name o into the complex concept o, which is interpreted as a singleton with o as its single element. Nominals can be used to enumerate all the elements of a class: for example,
$Continents \equiv \{Africa, Antarctica, Asia, Australia, Europe, NorthAmerica, SouthAmerica\}$.

The logic obtained by extending $\mathcal{ALC}$ with nominals is called $\mathcal{ALCO}$.

More expressive DLs can also be obtained by allowing new kinds of axioms in the RBox. For example, *transitivity* allows role axioms to be interpreted as transitive binary relations, e.g., if the role $locatedIn$ is transitive and the assertions $locatedIn(CP, Maryland)$ and $locatedIn(Maryland, USA)$ are contained in the ABox, then the assertion $locatedIn(CP, USA)$ would be inferred from the knowledge base. The extension of $\mathcal{ALC}$ with transitive roles is called $\mathcal{ALCR}^+$. This logic is also abbreviated as S because of the correspondence between $\mathcal{ALCR}^+$ and the multimodal logic $\mathbf{S4}$.

Another useful role axiom is *inversion*, which allows the use of relations in 'both directions'. For example, if the relations $hasChild$ and $isChildOf$ are defined as inverses of each other, then given the assertion $hasChild(Jack, Mary)$ in the ABox, one can infer $isChildOf(Mary, Jack)$.

Finally, several extensions of DLs [65], [64] have been investigated for describing concepts in terms of *datatypes*, such as numbers or strings, which is crucial for many applications. The main approach has been to provide DLs with an interface to 'concrete' domains , which consist of a set (such as the natural numbers or strings), together with a set of built-in predicates, which are associated with a fixed extension on that set, such as

$\geq +, *$ for the natural numbers. The interface between the DL and the concrete domain is achieved by defining a new kind of roles, called concrete roles, which relate objects from the 'DL-side" with data values from the concrete domain; and enriching the DL with a new concept constructor associated to those concrete roles. Using these constructors, it is possible, for example, to describe a set of all people whose weight is less than 50 kg: $\exists weight.\ \leq_{20}$. The mnemonic $\mathcal{D}$ is used to represent DLs that have been extended with datatype support. Note that the Web Ontology Language OWL-DL, which we shall see later, is a syntactic variant of the description logic $\mathcal{SHOIN}(\mathcal{D})$ and thus is a very expressive language.

2.1.1 Syntax and Semantics of $\mathcal{SHOIQ}(\mathcal{D})$

In this section, we describe the syntax and semantics of the logic $\mathcal{SHOIQ}(\mathcal{D})$. We start with the definition of roles.

Definition 1 *(SHOIQ(D)-roles)*

Let V_R, V_R^c be the disjoint sets of abstract and concrete atomic roles respectively. The set of $\mathcal{SHOIQ}(\mathcal{D})$ abstract roles is the set $V_R \cup \{R^- | R \in V_R\}$. The set of concrete roles is just V_R^c. A role inclusion axiom is an expression of the form $R_1 \sqsubseteq R_2$, where R_1, R_2 are abstract roles or an expression of the form $u_1 \sqsubseteq u_2$, where u_1, u_2 are concrete roles. A transitivity axiom is an expression of the form $Trans(R)$, where $R \in V_R$. An RBox is a finite set of role inclusion axioms and transitivity axioms.

Notation Remarks: In order to avoid considering roles of the form R^{--}, we define the function $Inv(R)$ that returns the inverse of an abstract role R. Let R be a RBox; we introduce the symbol $\sqsubseteq_R^*$ to denote the reflexive-transitive closure of $\sqsubseteq$ on R $\cup\{Inv(R_1) \sqsubseteq Inv(R_2) | R_1 \sqsubseteq R_2 \in R\}$. We use $R_1 \equiv_R R_2$ as an abbreviation for $R_1 \sqsubseteq_R^* R_2$ and $R_2 \sqsubseteq_R^* R_1$. Note that inverses cannot be defined on concrete roles.

We define the function $Tr(S, R)$ that returns $true$ if S is a transitive abstract role (atomic or not). Formally, $Tr(S, R) = true$ if, for some P with $P \equiv_R S$, $Trans(P) \in R$ or $Trans(Inv(P)) \in R$. The function returns $false$ otherwise. Note the difference between the function Tr(S,R), which maps roles to boolean values, and the axiom Trans(R), which states that the atomic role R is transitive. A concrete role, on the other hand, cannot be made transitive.

An abstract role R is *simple* w.r.t. the RBox R if $Tr(S, R) = false$ for all $S \sqsubseteq_R^* R$.

Definition 2 *($\mathcal{SHOIQ}(\mathcal{D})$-concepts and knowledge bases)*

Let V_C and V_I be sets of atomic concepts and object names respectively, and let R be an RBox. The set of $\mathcal{SHOIQ}(\mathcal{D})$-concepts is the smallest set such that:

- *Every atomic concept $A \in V_C$ is a concept.*
- *If C, D are concepts and R is a role, then ($C \sqcap D$) (Intersection), ($C \sqcup D$) (Union), ($\neg C$) (Negation), ($\forall R.C$) (Universal Restriction) and ($\exists R.C$) (Existential Restriction) are also concepts.*
- *If n is a natural number and S is a simple role, then ($\geq nS.C$) (at-most Number Restriction) and ($\leq nS.C$) (at-least Number Restriction) are also concepts.*

- *If Φ is a datatype and $\mathbf{u}$ a concrete role, then ($\exists \mathbf{u}.\Phi$), ($\forall \mathbf{u}.\Phi$) are also concepts.*
- *If $a \in V_I$, the nominal $\{a\}$ is a concept.*

For C,D concepts, a concept inclusion axiom is an expression of the form $C \sqsubseteq D$. A TBox T is a finite set of concept inclusion axioms. The use of nominals allows the encoding of ABox assertions as TBox axioms. Hence, a $\mathcal{SHOIQ}$ knowledge base, *$\mathcal{K}$, simply consists of a TBox and an RBox, i.e., $\mathcal{K} = (\mathcal{T}, \mathcal{R})$.*

Definition 3 *($\mathcal{SHOIQ}(\mathcal{D})$ interpretations)*

An interpretation I is a pair I = $(W, .^I)$, where W is a non-empty set, called the ***domain of the interpretation****, which is disjoint from the concrete domain W_D, and $.^I$ is the* ***interpretation function****. The interpretation function maps:*

- *Each atomic concept A to a subset A^I of W*
- *Each abstract atomic role R to a subset R^I of $W \times W$*
- *Each concrete atomic role $\mathbf{u}$ to a subset $\mathbf{u}^I$ of $W \times W_D$*
- *Each object name a to an element a^I of W*

The interpretation can be extended to $\mathcal{SHOIQ}(\mathcal{D})$-abstract roles as follows. Let R be an abstract atomic role, then:

$$(Inv(R))^I = (a, b) \in W \times W | (b, a) \in R^I$$

The interpretation function is extended to concept descriptions as follows:

- $(C \sqcap D)^I = C^I \cap D^I$
- $(C \sqcup D)^I = C^I \cup D^I$
- $(\neg C)^I = W - C^I$
- $(\exists R.C)^I = \{a \in W | \exists b \in W$ *with* $(a, b) \in R^I$ *and* $b \in C^I\}$
- $(\forall R.C)^I = \{a \in W | \forall b \in W$ *if* $(a, b) \in R^I$*, then* $b \in C^I\}$
- $(\geq nR.C)^I = \{a \in W$ *such that* $||\{b|(a, b) \in R^I$ *and* $b \in C^I\}|| \geq n\}$
- $(\leq nR.C)^I = \{a \in W$ *such that* $||\{b|(a, b) \in R^I$ *and* $b \in C^I\}|| \leq n\}$
- $\{a\}^I = \{a^I\}$
- $(\exists \mathbf{u}.\phi)^I = \{a \in W | \exists \phi \in W_D$ *with* $(a, \phi) \in \mathbf{u}^I$ *and* $\phi \in \Phi^D\}$
- $(\forall \mathbf{u}.\phi)^I = \{a \in W | \forall \phi \in W_D$ *if* $(a, \phi) \in \mathbf{u}^I$ *then* $\phi \in \Phi^D\}$
- $(\geq n\mathbf{u}.\phi)^I = \{a \in W$ *such that* $||\{\phi|(a, \phi) \in \mathbf{u}^I$ *and* $\phi \in \Phi^D\}|| \geq n\}$
- $(\leq n\mathbf{u}.\phi)^I = \{a \in W$ *such that* $||\{\phi|(a, \phi) \in \mathbf{u}^I$ *and* $\phi \in \Phi^D\}|| \leq n\}$

The interpretation function is applied to the axioms in a $\mathcal{SHOIQ}(\mathcal{D})$ KB according to the following definition:

Definition 4 *(Semantics of $\mathcal{SHOIQ}(\mathcal{D})$ Knowledge Bases)*

The $\mathcal{SHOIQ}(\mathcal{D})$ interpretation I satisfies the role inclusion axiom $R_1 \sqsubseteq R_2$ if $(R_1)^I \subseteq (R_2)^I$ and it satisfies the inclusion axiom $u_1 \sqsubseteq u_2$ if $u_1^I \subseteq u_2^I$. The interpretation

satisfies a transitivity axiom Trans(R) if the following condition holds:

$\forall a, b, c \in W$*, if* $(a, b) \in R^I$ *and* $(b, c) \in R^I$*, then* $(a, c) \in R^I$

The interpretation is a model of the RBox R, denoted by $I \models R$*, if it satisfies all its axioms.*

An interpretation I satisfies a concept inclusion axiom $C \sqsubseteq D$ *if* $C^I \subseteq D^I$*. The interpretation is a model of the TBox T, denoted by* $I \models \mathcal{T}$ *, iff it satisfies every concept inclusion axiom in* $\mathcal{T}$*. Finally, the interpretation is a model of the knowledge base* $\mathcal{K}$*= (*$\mathcal{T}$*,*$\mathcal{R}$*), denoted by* $I \models K$*, iff I is a model of both the TBox T and the RBox R.*

Thus an **inconsistent** KB $\mathcal{K}$= ($\mathcal{T}$, $\mathcal{R}$) is one for which **there exists no possible model**, i.e., there is no interpretation $\mathcal{I}$ that satisfies the semantics of all the axioms in $\mathcal{T}$ and $\mathcal{R}$. Inconsistent KBs are one of the key semantic (logical) defects considered in the thesis (the other being unsatisfiable concepts as we shall see below).

Typical inferences in $\mathcal{SHOIQ}(\mathcal{D})$ are concept subsumption and satisfiability w.r.t. a knowledge base:

Definition 5 *(Inferences)*

Let C,D be concepts, a, b object names and $\mathcal{K}$ *a knowledge base. We say that C is* ***satisfiable*** *relative to* $\mathcal{K}$*iff there is a model* $\mathcal{I}$*of* $\mathcal{K}$*, such that* $C^I \neq \emptyset$*. We say that C* ***subsumes*** *D* ***relative to*** $\mathcal{K}$*iff, for every model I of* $\mathcal{K}$*,* $C^I \subseteq D^I$*.*

Thus, an **unsatisfiable** concept is one for which there exists no model, i.e., its interpretation is the empty set in every model of the KB. Obviously, if the KB itself is inconsistent then all the atomic concepts in it are unsatisfiable.

2.2 Web Ontology Language (OWL)

The Web Ontology Language (OWL) [27], is an integral component of the Semantic Web, as it can be used to write ontologies or formal vocabularies which form the basis for semantic web data markup and exchange.

OWL is a fairly recent language, released as a W3C (World Wide Consortium) recommendation in February 2004. As part of the Semantic Web stack of languages, OWL is layered on top of RDF (basic assertional language) and RDFS (schema language extension for RDF) which themselves are layered over XML [16]. From its relationship with RDF comes the official OWL exchange syntax, namely RDF/XML [10]. In fact, OWL shares many features in common with RDF such as the use of Universal Resource Identifiers (URI) to unambiguously refer to web resources (as we shall see later).

From a modeling and semantic point of view, OWL shares a strong correspondence with Description Logics borrowing many logical constructs as shown in Table 2.2. The table lists the language constructs of OWL with the corresponding DL representation. Note that in OWL terms, `owl:class, owl:ObjectProperty, owl:DatatypeProperty, owl:Individual` and `owl:Datatype` correspond to concept, role, concrete role, object and concrete domain respectively in DLs.

OWL Construct	DL representation	Example
`owl:equivalentTo` (C,D)	$C \equiv D$ ($C \sqsubseteq D$ and $D \sqsubseteq C$)	$Person \equiv Human$
`rdfs:subClassOf` (C,D)	$C \sqsubseteq D$	$Parent \sqsubseteq Person$
`owl:complementOf` (C,D)	$C \equiv \neg D$ (negation)	$Male \equiv \neg Female$
`owl:disjointWith` (C,D)	$C \sqsubseteq \neg D$	$Father \sqsubseteq \neg Mother$
`owl:intersectionOf` (C,D)	$C \sqcap D$ (conjunction)	$Parent \sqcap Male$
`owl:unionOf` (C,D)	$C \sqcup D$ (disjunction)	$Father \sqcup Mother$
`owl:oneOf` (I_1, I_2)	$\{I_1\} \sqcup \{I_2\}$	$\{Jack\} \sqcup \{Jill\}$
`owl:someValuesFrom`(P,C)	$\exists P.C$ (existential)	$\exists hasChild.Daughter$
`owl:allValuesFrom`(P,C)	$\forall P.C$ (universal)	$\forall hasChild.Son$
`owl:hasValue` (P,I_1)	$\exists P.\{I_1\}$	$\exists hasChild.\{Jill\}$
`owl:cardinality`(P,n)	$= n.P$	$= 2.hasParent$
`owl:minCardinality`(P,n)	$\geq n.P$	$\geq 1.hasDaughter$
`owl:maxCardinality`(P,n)	$\leq n.P$	$\leq 2.hasChildren$

Table 2.2: Correspondence between OWL and DL (Note: C, D refer to OWL Classes; P refers to an OWL Property; I_1, I_2 refer to OWL Individuals; and n refers to a non-negative integer.)

OWL comes in three increasingly expressive **sub-languages** or "species", OWL-Lite, OWL-DL and OWL-Full.

- **OWL-Lite**: The motivation for OWL Lite is to support users primarily needing a classification hierarchy and simple constraints. These expressivity limitations ensure that it provides a minimal useful subset of language features, which are relatively straightforward for tool developers to support.

  Interestingly, OWL-Lite corresponds to an expressive description logic $\mathcal{SHIF}(D)$. This is because while many of the constructs that are allowed in $\mathcal{SHIF}(\mathcal{D})$ (for example, concept disjunction) are explicitly disallowed in the syntax of OWL-Lite, they can be 'recovered' by encoding them using General Concept Inclusion Axioms (GCIs).

- **OWL-DL**: OWL-DL supports those users who want the maximum expressiveness of the language without losing decidability. It includes all the OWL language constructs, but they can be used only under certain restrictions such as strict type separation (a class cannot be treated as an individual or a property, for example) and the inability to use transitive roles on number restrictions. OWL-DL corresponds to the description logic $\mathcal{SHIQN}(\mathcal{D})$. Hence, the debugging and repair techniques devised in this thesis have focused on this particular logic.

- **OWL-Full**: OWL-Full has the same vocabulary as OWL DL but it allows the free, unrestricted use of RDF constructs (e.g., classes can be instances). OWL-Full is thus a same syntax, extended semantics extension of RDF and is undecidable. Recently, however, [73] showed that under certain conditions (assuming contextual semantics), OWL-Full can be made decidable.

Finally, we discuss other key features of OWL that are important for its proper understanding and use.

- OWL provides a special construct, `owl:imports`, which allows one to bring in information from an external ontology. However, the only way that the construct works is by bringing into the original ontology *all* the axioms of the imported one. Therefore, the only difference between copying and pasting the imported ontology into the importing one and using an `owl:imports` statement is the fact that with imports both ontologies stay in different files. As of now, there is no mechanism in OWL for *partial* imports and this remains an interesting research problem.
- As noted earlier, OWL entities, ontologies and even the primitives of the language, are denoted using a URI. Interestingly, the meaning of the URI is relative to a particular RDF document [44]. In other words, the meaning of the same URI in other documents is not considered at all unless the document is imported. This is an important issue that OWL ontology modelers and users need to be aware of. For example, if we were building an OWL ontology dealing with the medical domain and wanted to reuse the concept $Cancer$ defined in the OWL version of the National Cancer Institute (NCI) Thesaurus, we cannot simply refer to the $Cancer$ URI in the NCI ontology to capture the concept meaning, instead, we need to import the entire NCI thesaurus into our ontology.
- OWL does not make the Unique Name Assumption (UNA). Given two object names a, b, it is generally assumed that they denote different things under DL semantics, i.e., $a^I \neq b^I$ for every interpretation $\mathcal{I}$. In OWL, however, different names could refer to the same object, which can lead to some non-intuitive inferences, e.g, suppose an OWL ontology contains the assertions $hasFather(Mary, Jack)$ and $hasFather(Mary, John)$, where $hasFather$ is a functional role, the resultant ontology is not inconsistent, but instead entails that $John$ and $Jack$ are the same object. To deal with the lack of UNA, OWL incorporates two additional primitives `owl:sameAs` and `owl:differentFrom` which respectively state that two objects are the same or distinct. The implementation of the UNA in DL reasoners is however quite straightforward.
- Since OWL semantics is based on DLs (which are usually subsets of FOL), OWL makes the open world assumption (OWA). Under OWA, any information not specified in the OWL KB is assumed *unknown* (as opposed to *false* under the closed world assumption). While this allows for partial or incomplete information to be represented, it can also lead to a source of confusion, especially for users familiar with closed world reasoning (e.g., users working with databases, logic programming, constraint languages in frame systems etc.). Consider the following example (taken from [87]): the class $MargheritaPizza \sqsubseteq Pizza \sqcap \exists hasTopping.Tomato \sqcap \exists hasTopping.Mozzarella$ is not classified as a $VegPizza$ even though both its specified toppings are vegetables. This is because, under OWA, we need to explicitly specify that the $MargheritaPizza$ has those two toppings *only and nothing else* for it to be classified correctly.

From a debugging standpoint, understanding the above features is key for ontology

authors as they represent crucial factors responsible for causing inconsistency errors and unintended inferences in the ontology [87].

2.3 Reasoning Services for OWL

Reasoning services for OWL are typically the same as that for DLs, and include:

- **Consistency Checking**: Check whether an OWL ontology $\mathcal{O}$ is logically consistent
- **Class Subsumption**: Given a pair of classes C, D in the ontology $\mathcal{O}$, check whether $\mathcal{O} \models C \sqsubseteq D$. Also related is the notion of **class satisfiability**: $C \sqsubseteq \bot$; and **class equivalence**: $C \equiv D$, which implies $C \sqsubseteq D$ and $D \sqsubseteq C$
- **Instantiation**: Given an individual a and a class C in the ontology $\mathcal{O}$, check whether a is an instance of C, i.e., $\mathcal{O} \models C(a)$. Also related is the notion of **retrieval**, i.e., obtain all individuals of class C

In propositionally closed DLs, subsumption can be reduced to satisfiability, since C subsumes D relative to $\mathcal{O}$ iff the concept $C \sqcap \neg D$ is unsatisfiable relative to $\mathcal{O}$.

Similarly, the instance problem can be reduced to the consistency problem: the object a is an instance of C relative to $\mathcal{O}$ iff the ontology $\mathcal{O}'$ obtained from $\mathcal{O}$ by adding to it the class assertion $\neg C(a)$ is inconsistent.

Finally, the concept satisfiability problem can be reduced to the ontology consistency problem: the concept C is consistent relative to the ontology $\mathcal{O}$ iff the knowledge base $\mathcal{O}'$ obtained from $\mathcal{O}$ by adding the concept assertion C(a) (with a a new object name) is consistent.

To solve this key consistency checking problem for OWL-DL ontologies (i.e., $\mathcal{SHOIN}(\mathcal{D})$ knowledge bases), there exist sound and complete decision procedures based on tableaux calculus [5].

2.4 Tableaux Algorithms

In this section, we briefly discuss the tableau algorithm for the DL $\mathcal{SHOIN}$.For a detailed description of the algorithm, we refer the reader to [52].

As noted earlier, the presence of nominals in $\mathcal{SHOIN}$ allows us to exclude the ABox from consideration, i.e., the KB $\mathcal{K}$consists of a general TBox $\mathcal{T}$and a Role Hierarchy $\mathcal{R}$. Additionally, the presence of transitive roles and role hierarchies in the logic allows reasoning with respect to general $\mathcal{T}$ and $\mathcal{R}$ to be reduced to reasoning w.r.t. $\mathcal{R}$ alone. This is because the entire TBox can be *internalized* [49] into a single concept description. Thus, the tableau decision procedure checks the consistency of an internalized concept D w.r.t $\mathcal{R}$.

DL tableau-based algorithms decide the consistency of D w.r.t $\mathcal{R}$ by trying to construct (an abstraction of) a model for it, called a **completion graph**. Each node x in the graph represents an individual, labeled with the set of concepts $\mathcal{L}(x)$ it has to satisfy, i.e, if $C \in \mathcal{L}(x), x \in C^I$. Each edge (x, y) in the graph is labeled with a set

of role names, and represents a pair occurring in the interpretation of the role, i.e., if $\mathcal{L}(x, y) = R, (x, y) \in R^I$.

The completion graph for a $\mathcal{SHOIN}$ KB is initialized as a *forest* of root nodes, each representing a nominal (individual) asserted in the ontology. Then, a series of *expansion rules* are applied in succession to build the graph, each adding new nodes or edges (and/or labels resp.), in keeping with the semantics of the concept and role descriptions. For example, if a concept $C \sqcap D$ is present in the label of a node x, then the individual that x represents must be an instance of both C and D and thus C, D are separately added to $\mathcal{L}(x)$ as well (this is handled by the $\sqcap$-rule). Similarly, if the concept $\exists R.E$ is present in the label of a node y, then there must exist at least one R-edge from the individual represented by y to another (arbitrary) individual of type E, and thus if no such edge already exists, an edge is created from node y to a new node z and the concept E is added to label of z (this is handled by the $\exists$-rule).

Note that the expansion rules are *non-deterministic*. For example, if the disjunction $C \sqcup D$ is present in the label of a node, the algorithm chooses either C or D to be added to the node label before proceeding. To account for this non-determinism, we consider a tree of completion graphs Δ instead of a single graph, i.e., the application of a non-deterministic rule results in the creation of a new completion graph, added to Δ, for each possible non-deterministic choice (for this purpose, we also maintain a set Σ of edges to be added at the next level of the tree).

The expansion rules for the $\mathcal{SHOIQ}$ consistency checking algorithm are shown in Table 2.3[1]. A summary of the terminology used in the rules is as follows:

- If (x, y) is an edge in the completion graph, then y is called a *successor* of x and x is called a *predecessor* of y. *Ancestor* is the transitive closure of predecessor, and *descendant* is the transitive closure of successor. A node y is called an *R-successor* of a node x if, for some R' with $R' \sqsubseteq^*_R R$, $R \in \mathcal{L}(x, y)$. A node y is called a neighbor (R-neighbor) of a node x if y is a successor (R-successor) of x or if x is a successor (Inv(R)-successor) of y.

  For a role S and a node x in G, we define the set of x's *S-neighbors* with C in their label, $S^G(x, C)$, as follows: $S^G(x, C)$:= $\{y|\ y$ is an S-neighbor of x and $C \in \mathcal{L}(y)\}$.
- A node x is a *nominal* node if $\mathcal{L}(x)$ contains a nominal. A node that is not a nominal node is a *blockable* node. An R-neighbor y of a node x is *safe* if (i) x is blockable or if (ii) x is a nominal node and y is not blocked.
- In order to ensure termination when dealing with infinite models, the algorithm uses a special condition known as *blocking*. A node x is label blocked if it has ancestors x_0, y and y_0 such that

  1. x is a successor of x_0 and y is a successor of y_0,
  2. y, x and all nodes on the path from y to x are blockable,
  3. $\mathcal{L}(x) = \mathcal{L}(y)$ and $\mathcal{L}(x_0) = \mathcal{L}(y_0)$, and

[1]Note: In Table 2.3, $\mathsf{Add}(\mathsf{C}, \mathsf{x})$ is an abbreviation for $\mathcal{L}(x) \leftarrow \mathcal{L}(x) \cup \{C\}$, $\mathsf{Add}(\mathsf{S}, \langle \mathsf{x}, \mathsf{y} \rangle)$ is an abbreviation for $\mathcal{L}(x, y) \leftarrow \mathcal{L}(x, y) \cup \{S\}$

4. $\mathcal{L}(x', x) = \mathcal{L}(y', y)$.

In this case, we say that y blocks x. A node is blocked if either it is label blocked or it is blockable and its predecessor is blocked; if the predecessor of a safe node x is blocked, then we say that x is indirectly blocked.

- In some rules, e.g., $\leq$-rule, we *merge* one node y into another node x. This involves adding $\mathcal{L}(y)$ to $\mathcal{L}(x)$, 'moving' all the edges leading to y so that they lead to x and 'moving' all the edges leading from y to nominal nodes so that they lead from x to the same nominal nodes; we then remove or *prune* y (and blockable sub-trees below y) from the completion graph. Details of the Merge and Prune operations are in [52].

A completion graph in Δ is said to contain a clash if:

- both the concepts C, $\neg C$ are present in the label of the same node
- A node that contains the concept $\leq nS$ (where S is a role) has more than n distinct S-neighbors
- A nominal node o which can only represent one distinct individual in a model is said to belong to two distinct nodes in the graph, i.e., $o \in \mathcal{L}(x) \sqcap \mathcal{L}(y)$ where $x \neq y$

Each time a clash is detected, the algorithm jumps to the next graph in Δ at the same level. Once all the leaf graphs in Δ have been explored (i.e., all non-deterministic choices have been considered) and/or no more expansion rules can be applied, the algorithm terminates.

If all the leaf completion graphs in Δ contain a clash or a contradiction, the algorithm returns *inconsistent* as no model can be found. Otherwise, any one clash-free completion graph generated by the algorithm represents one possible model for the concept and thus the algorithm returns *consistent*.

2.4.1 Optimizations

Non-deterministic tableau algorithms for expressive DLs are intractable (e.g., the worst case complexity of the $\mathcal{SHOIQ}$ algorithm is 2NExpTime [103]). As a consequence, there exists a significant gap between the design of a decision procedure and the achievement of a practical implementation. Naive implementations are doomed to failure. In order to achieve acceptable performance, modern DL reasoners, such as RACER [104], FaCT++ [50] and Pellet [97], implement a suite of optimization techniques [51], [40], [39], [96]. These optimizations lead to a significant improvement in the empirical performance of the reasoner and have proved effective in wide variety of realistic applications.

We briefly summarize some of the key optimizations for DL tableau algorithms:

- Pre-processing Optimizations:
  - *Normalization* and *Simplification*: Normalization is the syntactic transformation of a concept expression into a normal form. For example, in the negation normal form (NNF), a negation appears only before an atomic concept. Any

$\sqcap$-rule: **if** $(C_1 \sqcap C_2) \in \mathcal{L}(x)$, x not indirectly blocked, and $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$,
$\mathsf{Add}(\{\mathsf{C_1}, \mathsf{C_2}\}, \mathsf{x})$).

$\sqcup$-rule: **if** $(C_1 \sqcup C_2) \in \mathcal{L}(x)$, x not ind. blocked, and $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$,
Generate graphs $\mathbf{G}_i := \mathbf{G}$ for each $i \in \{1, 2\}$
$\Delta := \Delta \cup \{\mathbf{G}_1, \mathbf{G}_2\}$; $\Sigma := \Sigma \cup \{\mathbf{G} \prec \mathbf{G}_1, \mathbf{G} \prec \mathbf{G}_2\}$
$\mathsf{Add}(\mathsf{C_i}, \mathsf{x})$ in $\mathbf{G}_i$ for each $i \in \{1, 2\}$

$\exists$-rule: **if** $\exists S.C \in \mathcal{L}(x)$, x not blocked, and no S-neighbor y with $C \in \mathcal{L}(y)$
Create y, $\mathsf{Add}(\mathsf{S}, \langle \mathsf{x}, \mathsf{y} \rangle)$, $\mathsf{Add}(\mathsf{C}, \mathsf{y})$

$\forall$-rule: **if** $\forall S.C \in \mathcal{L}(x)$, x not ind. blocked, y S-neighbor of x and $C \notin \mathcal{L}(y)$:
$\mathsf{Add}(\mathsf{C}, \mathsf{y})$

$\forall^+$-rule: **if** $\forall S.C \in \mathcal{L}(x)$, x not ind. blocked, y R-neighbor of x with $\mathsf{Trans}(\mathsf{R})$ and $R \sqsubseteq S$:
if $\forall S.C \notin \mathcal{L}(y)$, $\mathsf{Add}(\forall \mathsf{S.C}, \mathsf{y})$

$\geq$-rule: **if** $(\geq nS) \in \mathcal{L}(x)$, x not blocked: and no safe S-neighbors $y_1, .., y_n$ of x with $y_i \neq y_j$
Create $y_1, .., y_n$; $\mathsf{Add}(\mathsf{S}, \langle \mathsf{x}, \mathsf{y_i} \rangle)$; $\neq(\mathsf{y_i}, \mathsf{y_j})$

$\leq$-rule: **if** $(\leq nS) \in \mathcal{L}(x)$, x not ind. blocked, $y_1, .., y_m$ S-neighbors of x, $m > n$:
For each possible pair y_i, y_j, $1 \leq i, j \leq m; i \neq j$:
Generate a graph $\mathbf{G}'$; $\Delta := \Delta \cup \{\mathbf{G}'\}$; $\Sigma := \Sigma \cup \{\mathbf{G} \prec \mathbf{G}'\}$
if y_j a nominal node, $\mathsf{Merge}(\mathsf{y_i}, \mathsf{y_j})$ in $\mathbf{G}'$,
else if y_i a nominal node or ancestor of y_j, $\mathsf{Merge}(\mathsf{y_j}, \mathsf{y_i})$,
else $\mathsf{Merge}(\mathsf{y_i}, \mathsf{y_j})$ in $\mathbf{G}'$
if y_i is merged into y_j, for each concept C_i in $\mathcal{L}(y_i)$,

O-rule: **if**, $\{o\} \in \mathcal{L}(x) \cap \mathcal{L}(y)$ and not $x \dot{\neq} y$, **then** $\mathsf{Merge}(\mathsf{x}, \mathsf{y})$.

NN-rule: **if** $(\leq nS) \in \mathcal{L}(x)$, x nominal node, y blockable S-predecessor of x and there is no m s.t. $1 \leq m \leq n, (\leq mS) \in \mathcal{L}(x)$ and there exist m nominal S-neighbors $z_1, ...z_m$ of x s.t. $z_i \neq z_j$, $1 \leq i \leq j \leq m$, then
Generate new $\mathbf{G_m}$ for each m, $1 \leq m \leq n$, add $\Delta := \Delta \cup \{\mathbf{G_m}\}$; $\Sigma := \Sigma \cup \{\mathbf{G} \prec \mathbf{G_m}\}$ and do the following in each $\mathbf{G_m}$:
$\mathsf{Add}(\leq \mathsf{mS}, \mathsf{x})$
create $y_1, ...y_m$; Add $y_i \neq y_j$ for $1 \leq i \leq j \leq m$.
$\mathsf{Add}(\mathsf{S}, \langle \mathsf{x}, \mathsf{y_i} \rangle)$; $\mathsf{Add}(\{\mathsf{o_i}\}, \mathsf{y_i})$:

Table 2.3: Tableau Expansion Rules for $\mathcal{SHOIQ}$

concept can be converted to an equivalent one in NNF by pushing negations inwards using a combination of DeMorgan's Laws. Normalization helps discover contradictions easily, by syntactically comparing expressions in their normal form, e.g., $C \sqcap \neg(C \sqcup D) \rightarrow (C \sqcap \neg C) \sqcap \neg D$.

Sometimes, normalization can also include a range of simplifications so that obvious contradictions and tautologies are detected; for example, $(C \sqcap \bot)$ could be simplified to $\bot$.

– *Absorption*: Absorption is the process of eliminating certain kinds of General Concept Inclusion axioms (GCI's) by embedding them in primitive concept definitions. The basic idea is to manipulate the GCI so that it has the form of a primitive definition $A \sqsubseteq D_0$, where A is an atomic concept name. This axiom can then be merged into an existing primitive definition $A \sqsubseteq C_0$ to give $A \sqsubseteq C_0 \sqcap D_0$ which then replaces the GCI in the KB.

The significance of absorption is the following: From a reasoning standpoint, the primitive definition axiom $C \sqsubseteq D$ can be used as a *macro* to expand the label of a node which contains C – by directly adding D, while the same does not hold for General Concept Inclusion Axioms (GCIs). Instead, a GCI needs to be converted to the disjunction $D \sqcup \neg C$ that must be added to *every* node label in the completion graph, which leads to a non-deterministic search, and is thus very expensive. Hence, the use of absorption can greatly reduce reasoning times for KBs containing numerous (absorbable) GCIs (e.g. the Galen medical ontology[2]).

• Optimizations during Tableau Expansion:

– *Lazy Unfolding*: Given an unfoldable KB $\mathcal{T}$ (consisting of unique, acyclic concept definitions), and a concept C whose satisfiability is to be tested with respect to $\mathcal{T}$, it is possible to eliminate from C all concept names occurring in $\mathcal{T}$ using a recursive substitution procedure called *unfolding*. The satisfiability of the resulting concept is independent of the axioms in $\mathcal{T}$ and can therefore be tested using a decision procedure that is only capable of determining the satisfiability of a single concept.

An optimization usually enforced in reasoners is *lazy unfolding*, i.e., unfolding on the fly, using pointers to refer to complex concepts, and detecting clashes between lexically equivalent concepts as early as possible, e.g., detecting a clash between the complex concepts $(C \sqcap D)$ and $\neg(C \sqcap D)$ before unfolding them.

– *Dependency Directed Backjumping*: Dependency directed backjumping is an optimization technique that adds an extra label to the type and property assertions so that the branch numbers that caused the tableau algorithm to add those assertions are tracked. Obviously, assertions that exist in the original ontology and the assertions that were added as a result of only deterministic rule applications will not depend on any branch. This means these assertions

[2]http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/galen.owl

are direct consequence of the axioms in the ontology and affect every interpretation. If a clash found during tableau expansion does not depend on any non-deterministic branch, the reasoner will stop applying the rule as it is obvious that there is no way to fix the problem by trying different branches.

Chapter 3
Related Work

Diagnosis has been widely regarded as an integral component of (deductive) reasoning systems for many years. Logic programming systems, rule-based expert systems, deductive databases and automated theorem provers (ATP) have all incorporated debugging and explanation facilities of some sort.

In this chapter, we review other related approaches. In particular, in section 3.1, we discuss the various types of debugging support found in existing logic-based systems; and in section 3.2, we look at two classical theories of diagnosis and revision, and describe the relation between these generic theories and the debugging/repair services devised in this thesis.

3.1 Diagnosis in Reasoning Systems

We first discuss debugging support found in non-Description Logic (DL) based deduction systems, and compare and contrast it to the DL case. We then enumerate recent trends for explanation and debugging in DL systems.

3.1.1 Debugging of Logic Programs

Logic Programming (LP) is a well-known programming paradigm based on a subset of First Order Logic–named Horn Clause Logic. LP has been extended with explicit negation (extended logic programming XLP [83]) and defaults giving rise to non-monotonic reasoning. These programming languages have both, a proof-theoretic and a model-theoretic semantics, with resolution-based algorithms for reasoning. Hence, the debugging of LP and XLP programs is a related field we need to explore.

We discuss two different debugging paradigms for LPs, *operational* and *declarative* debugging.

The naive approach to interactive debugging (a.k.a. *operational* debugging) involves instrumenting the program and exploring its execution trace [20], i.e., the user inserts appropriate break points in the program (e.g., between the expand and branch steps of the algorithm, or after each step of the inference function) and is given control of how many and what type of steps can be taken (e.g., `trace` and `spy` commands in Prolog work in this manner). Commands to these systems are typically broken into two categories, *control* commands that allow the computation to continue until a specified point is reached or condition occurred, and *display* commands that allow the user to query the status of atoms and rules within the current context. Numerous debugging systems work on this methodology. However, debugging of this kind can be painstakingly difficult placing a huge cognitive load on the user.

The analog from a DL debugging point of view is interesting to consider. Explaining the trace of the tableau reasoner amounts to iterating through the expansion process of

generating the completion graph, and displaying the sequence of expansion rules that are fired. However, there are several complications that need to be dealt with here. Firstly, the reasoner heavily modifies the original axioms of the KB internally (using techniques such as normalization, absorption etc.) and the labels of the nodes and edges in the graph barely resemble the original terms. Though it is possible to extend the algorithm to keep track of the axioms in the KB responsible for various tableau events (as done in Chapter 4), it places an additional burden on the user to correlate between the internal terms and the asserted axioms. Secondly, the application of expansion rules modify the graph in a variety of different ways, e.g., some rules cause a node merge, whereas others introduce successors to anonymous nodes, and explaining such graph changes to the user can be difficult, possibly requiring a flexible and scalable visual interface. Thirdly, even for simple inferences, the number of steps in the reasoning process can be very large due to many trivial steps, and thus isolating and identifying *critical* steps is important (besides allowing the user to systematically skip steps). For example, the point where a non-deterministic choice is introduced in the algorithm, or where the algorithm backtracks to a previous choice point can be considered as key steps, besides the obvious critical step when a contradiction is detected. Fourthly, it is not easy to retrace steps without caching a large amount of data. Also, memory management is an important issue in general, given that the size of the completion graph can blow up for complex inferences in large KBs. Besides the above factors, it is assumed that the user is aware of the basics of tableau-based reasoning. For all these reasons, to our knowledge, no effort has been made yet to visualize the trace of the tableau reasoner in a meaningful and effective manner.

On the other hand, *algorithmic* or *declarative* debugging introduced by Shapiro [94] introduces a theoretical framework for debugging. The process briefly works as follows: the debugging system builds an abstract model representing the execution trace of the program and elicits feedback from an *oracle* (could be the user) to navigate the model in a top-down manner till the faulty or erroneous component is reached. The declarative notion comes from the fact that the semantics of the program are encoded in the oracle, which needs to be able to differentiate between expected and unexpected behavior. The underlying principle is that a correct and complete oracle will always find the error using this algorithmic debugging procedure. The technique has been extended over the years. For example, while the oracle in [94] could only give yes/no answers, later work [31] allowed the oracle to provide assertions about the intended program behavior. Also, more recently, techniques have been developed to improve the quality of the queries posed to the oracle/user in debugging programs with *Answer Set* semantics [15] (also known as *query-based* debugging).

We now discuss the possibility of building an analogous system to deal with expressive DLs. The basic procedure would be to have the user start with the root inconsistency condition and investigate its dependencies in a top-down manner until the source of the problem is reached. Taking a simple example, if the contradiction or clash in the tableau reasoner was because a concept C and its negation were present in the label of some node x, we would start by displaying this root clash information to the user in a sensible manner (as done in Chapter 7). Suppose the user felt that $\neg C$ was mistakenly present, i.e., the individual represented by the node x should not have been of type $\neg C$, the next step would be to display to the user, the conditions that caused $\neg C$ to occur in $\mathcal{L}(x)$. The process

would recursively continue until the user discovered a faulty premise (axiom). The main challenge in this case lies in hiding the underlying details of the tableau reasoner and presenting the conditions and its premises in a useful manner, while dealing with the fact that a large number of inference steps may be present. In addition, there could be numerous clashes in the completion graph generated by the reasoner and we need to focus on only those clashes responsible for the inconsistency.

Finally, we also discuss a technique to diagnose and remove contradictions in XLP programs. The common theme, described initially in [84] (and later extended in [25] etc.), is to revise a contradictory program by changing the values of one or more *default* literals, which otherwise due to Closed World Assumption (CWA) is assumed to be *true* leading to the contradiction. The revision changes the value of the defaults to either *false* or *undefined* in order to regain consistency. The algorithm first determines *revisables*, i.e., literals whose values can be changed. It then exhaustively computes all consequences of the program containing contradictions and finds the sets of support (SOS) for each contradiction. Finally, it uses Rieter's Hitting Set [88] approach to arrive at minimal repair solutions involving the revisable literals in the computed SOS. An advantage of this repair technique is that it focuses on defaults, which provide an easy point for alteration. However, the technique is not directly applicable to OWL-DL, since OWL is based on a monotonic description logic $\mathcal{SHOIN}$ and hence lacks support for defaults. An interesting notion to take from here is the possibility of ontology modelers providing a list of revisable axioms or terms beforehand, which would act as a useful pointer to the debugging tool while generating repair solutions.

3.1.2 Expert System Debugging and Maintenance

Rule-base verification (or validation) has been an important area of research in the expert-system community. Verification criteria range from semantic checks for consistency and completeness, to structural checks for redundancy, relevance and reachability. Recent surveys can be found in [85], [2], [86].

Rule-based debuggers differ from programming language debuggers in that the former focus more on high-level details such as the interaction of rules with the underlying facts in the knowledge base, the interaction among rules, and the rule-event interaction. Early systems such as TEIRESIAS [26] (designed to work in conjunction with a completed MYCIN [95] rule base), and ONCOCIN [99] would generate a rule model showing the conditions used to reach certain conclusions, and test the model for conflicts, redundancy, subsumption, and missing rules or conditions. The significant problem with this approach is the combinatorial explosion, in which an impossibly large number of combinations exist. To deal with this problem, heuristic approaches have been suggested in Nyugen's CHECK system [77] and Stachowitz's EVA system [18]. CHECK builds relational tables to represent rule-dependencies (determined by matching clauses in rules) and generates a DAG from the generated tables. It inspects the DAG to find errors such as circular rules and unreachable conclusions. Similar techniques to detect structural (or *style* defects as defined in Chapter 1) in description logic KBs can be seen in tools such as Chimaera [70]. From a semantic point of view in DLs, heuristic approaches to detect simple conflicts in axioms based on structural dependency analysis can be seen in Chapter

5 (Structural Analysis) and [105]. More details on these follow in Section 3.1.5.

In some expert systems, the user can enter into an interactive dialogue with the system, and choose to focus on a specific executed part of the expert system so as to better understand its working. The explanations are provided using natural language paraphrases (e.g., MYCIN [95], XPLAIN [100], ESS [101]) or using an appropriate visualization scheme (e.g., Vizar [23]). In some systems, where there are a large number of low-level rules, or complex problem-solving strategies, knowledge engineers are allowed to provide for higher-level *meta-rules*, or abstract representations of the strategies, which are then used by the system to generate more concise explanations. Similarly, to help the user understand the rationale behind some of the rules, the implicit domain knowledge underlying the rules such as preferences for certain rules, tradeoff conditions etc. can be explicitly encoded by the system designer, which is then available in the debugging phase.

From a repair point of view for DL-based ontologies, the analog of annotating rules in the expert system could be useful. This would mean annotating axioms in the ontology, or explaining the modeling philosophy behind a particular set of concept/role definitions (e.g., by following the OntoClean [38] philosophy).Besides being used to explain the rationale for the presence of a certain set of axioms, the annotations can be used to rank axioms in the repair phase (see Chapter 6) and/or suggest revisions to the ontology which are in keeping with the modeling methodology.

Finally, we also discuss recent trends involving the use of machine learning to detect and resolve errors in rule-based expert systems. This has been seen in systems such as KR-FOCL [82] and more recently in [106]. The idea here is to investigate the execution trace of the system when used to learn a set of training cases containing positive and negative tests in order to expose faulty clauses in rules, e.g., clauses with extraneous or missing literals (similar form of diagnosis has also been proposed for logic programming systems [22]). Revisions are based on various heuristics that check which clauses are *operationalized* (come into effect) during the execution. If a clause is not operationalized at all during the learning phase, it is treated as a redundant clause and is removed from the system.

The analog in the DL case would be to devise a test suite for the ontology containing desired and undesired entailment tests and running the reasoner to see which tests pass/fail. Then, knowing the justification sets for the desired (positive) entailment tests that pass (using the Axiom Pinpointing service seen in Chapter 4), we can determine the *unused* axioms which are not responsible for any entailment and flag them to the user. Also, for the undesired (negative) entailment tests that pass, we can look at the corresponding justification sets and consider appropriate revisions to the ontology (on the lines of the repair strategies seen in Chapter 6). However, a more difficult problem is dealing with the desired entailment tests that *fail*. In this case, a trivial solution is to *simply* add the entailment as an axiom to the ontology, but this is probably not what the user expects. The problem is compounded by the fact that explaining the cause of the non-entailment to the user is hard, since in terms of the tableau-based refutation technique, it implies that the reasoner is able to construct *any one* model representing the counter-example. Explanations using counter-examples have been investigated in [67], where the author deals with a much weaker description logic for which non-tableau based reasoning algorithms are used. Extending this technique to tableaux calculus is, however, an open issue.

3.1.3 Repairing Integrity Constraint Violations in Deductive Databases

In this subsection, we briefly discuss automated repair strategies when dealing with Integrity Constraint (IC) violations in deductive databases. ICs are certain rules (usually specified at database design time) that must be satisfied by the database under all transactions to maintain integrity. In [75], an approach is presented where the designer of the consistency constraints specifies a set of repair actions to be taken for each constraint. Once a consistency violation is detected, the system automatically selects one of the repair actions for one of the violated constraints (possibly prioritized), performs it, and restarts the consistency check.

While there exists no analogous technique for logic-based KBs, a similar theme has been discussed in [11], where inconsistency resolution is considered in the context of *stratified* propositional KBs. In the DL case, the stratified KB, as carefully modeled by the ontology designer, would contain alternate versions for each of axioms (each at a different *strata* or layer), with the idea that when a particular erroneous axiom is found, it is automatically replaced by the corresponding axiom in a lower strata until the consistency of the KB is restored. Obviously, designing such a KB requires a lot of skill and effort on the ontology modelers' part.

An alternate approach to automated repair in deductive databases is presented in [72], where the database consistency check is traced to obtain symptoms that violate the constraints, and dependency analysis is done to identify potential and definite causes. The causes are transformed into repair transactions and presented to the user. In order to "clean up" repairs, various heuristics are used to eliminate unwanted solutions (e.g., facts that derive an existing inference are not added) and sort due to *plausibility* (e.g., more importance is given to shorter transactions, or those that minimally change the database). Similar heuristics can be seen in our Ontology Repair service (Chapter 6), where we determine the importance of a repair solution based on its size and impact on the ontology.

3.1.4 Proof Explanation in ATPs

We briefly review the proof-explanation literature to compare and contrast our explanation support described in Chapter 4 (Axiom Pinpointing service implementation).

Most proof explanation facilities for ATPs are based on the following fundamental principles described in [37]: *"**(a)** The exact way in which the knowledge is coded and structured in the system is irrelevant to the user; **(b)** All information accidental to the proof process should be omitted from the explanation; **(c)** The user himself must be able to achieve the deduction steps in a simple and direct inferential process as long as he knows the premises, in their correct order, and the conclusion; **(d)** The amount of information contained in any explanation step should be limited to the amount that can be simultaneously visualized and processed by a human being without great effort."*

The above principles translate into a set of transformations that need to be applied to the proof to convert it into a human-oriented form. One common example (as seen in [32], [53], [33] etc.) is the use of *proof trees* as a flexible structure for the argument, where the root of the tree is the main theorem, and every inference rule that proves the theorem becomes a child of the root. The tree is recursively expanded by considering

the premises of each child inference rule. The use of the tree structure allows the user to direct his attention to a particular fragment of the proof, focus solely on the relevant conditions necessary to derive that fragment, and use the chain of inferences to understand the broader reasoning step.

In our case, the main explanation generation component closely resembles the approach in [79], which generates arguments in FOL-based KBs based on the above principles. Common ideas here include using an appropriate tree-style layout (indentation) to construct an inference chain, suppressing irrelevant parts of the axioms that do not contribute to the entailment, and the use of hypertext to support navigation across different axioms (parts of the argument). In this manner, our system adheres to the principles above, however, the main challenge for expressive DLs is due to the complex interaction between the inference rules leading to the final conclusion, which makes it difficult to order the steps of the deduction properly. We have explored workarounds as discussed in Chapter 8 (e.g., by inserting intermediate inference steps in the proof), though generating an easy-to-understand explanation chain for all cases remains an open issue.

3.1.5 Description Logic (DL) Explanation and Debugging

We divide this discussion into two parts – first we enumerate generic explanation support for DLs, and then we focus specifically on the debugging and repair facilities developed for DL KBs in recent years.

Explanations for DL, 1995 - present

One of the earliest works in the field of explanations for description logic (DL) systems is [69], where a deductive framework based on natural deduction style proof rules is used to explain inferences in the CLASSIC [13]. CLASSIC is a family of knowledge representation languages based on DLs and it allows universal quantification, conjunction and restricted number restrictions. In [69], the authors argue that the standard implementation for reasoning in CLASSIC based on structural subsumption algorithms involves steps such as normalization, graph construction and traversal etc., where the asserted information is modified to such an extent that explaining the inference by mirroring the implementation and tracing the code directly is difficult for users to follow. Hence, they propose a proof-theoretic form of explanation, whereby the reasoning procedures of the system are encoded as natural deduction style inference rules (e.g. modus ponens). In order to simplify explanations, they define the notion of atomic descriptions, atomic explanations and explanation chains, and also decompose lengthy explanations into smaller steps. However, there exist some drawbacks of this approach. Firstly, the authors acknowledge that the definition of atomic descriptions is sufficient for CLASSIC, however, it breaks for more expressive DLs (e.g. including role composition). Secondly, the relative simplicity of the inference rules results from the fact that the reasoning algorithms are based on structural subsumption. However, structural subsumption is known to be incomplete for expressive DLs, where tableaux algorithms are typically used. In such cases (i.e., for more expressive DLs), explanation generation needs to be modified and natural-semantics style inference rules corresponding to the tableaux expansion procedure need

to be derived, which adds a new level of complexity.

The authors take an alternate approach in [14] by introducing a sequent calculus to explain $\mathcal{ALC}$ subsumption. The motivation here is to use modified sequent rules to imitate the behavior of tableau calculus and that of human reasoning, and additionally use quasi-natural-language paraphrases to explain the rule application. An advantage of sequent rules is that the original structure of the concepts is preserved and the concepts are not shifted between the subsumer and subsumee positions in the proof. This principle has been extended to definitorial $\mathcal{ALEHF}_{\mathcal{R}^+}$ TBoxes (with global domain/range restrictions) in [62] and implemented in the ontology editor OntoTrack [61]. While this explanation technique is tied to the tableau algorithm, its main disadvantage is that most of the common tableau optimizations (except lazy unfolding) cannot be applied as they modify the structure of the asserted axioms, which the explanation component is very sensitive to. Hence, the performance penalty on the explanation generation is huge. In addition, the authors of [62] acknowledge that extending the technique to say, generalized cardinality, could blow up the explanation because of the potentially huge set of cardinality enforced combinatorial changes. Finally, another drawback we see with this approach is that the quality of the quasi-NL explanations is severely hampered by complex concept descriptions, and it is an open question of how effective the NL can be for *understanding* the cause of the entailment. The lack of a user study in [62] is a concern in this respect.

In contrast to the earlier works, [6] describes a technique to find minimal sets of axioms responsible for an entailment (in this case, minimal inconsistent ABoxes) by labeling assertions, tracking labels through the tableau expansion process and using the labels of the clashes to arrive at a solution. The technique is applicable to the logic $\mathcal{ALCF}$. Similar ideas can be seen in [91], where the motivation is debugging unsatisfiable concepts in the DICE terminology. The main contribution of the paper is a formalization of the problem including the specification of terms such as MUPS and MIPS, which are essentially minimal fragments of a KB responsible for a particular set of error(s) in it. We have extended this work in [58] to the more expressive logic $\mathcal{SHIF}$, which corresponds to OWL-$Lite$, where we have presented a computationally more efficient algorithm to find the MUPS by avoiding tableau saturation (which [91] proposes). Also, we show through a usability evaluation that various UI enhancements to the display of the MUPS, such as highlighting key entities, ordering/indenting axioms etc. (see Chapter 8) are useful for understanding and debugging unsatisfiable concepts in OWL ontologies. We have further extended this technique to explain arbitrary entailments in OWL-DL as discussed in Chapters 4, 7.

Finally, there has been recent work done on explaining DL reasoning in $\mathcal{ALC}$ using an FOL-resolution based approach [28]. The idea here is to translate the DL axioms into FOL formulae or clauses, use a resolution-based theorem prover to derive the contradiction (which is expected beforehand), and transform the resolution proof into a *refutation graph*. The refutation graph being a more abstract representation of the proof is more useful for explanation purposes, and traversing the graph in an appropriate manner helps understand the cause of the various intermediate resolution steps leading to the ultimate goal. The work is still in its infancy, with the authors presenting two simple examples to demonstrate their technique. It is interesting to see whether the technique scales to more complex examples containing many steps of resolution in a large proof. Challenges include dealing with the problem of skolemization due to existential restrictions (which

blurs the gap between the asserted axioms and the FOL clauses), deriving a traversal of the graph that is easy to follow/understand (since there could be many traversal options), and determining through a usability evaluation, whether users find this technique of explanation helpful.

We also note that having generated an explanation, the *Inference Web* (IW) Infrastructure [68] can be used to exchange them across reasoning systems and users. IW comprises of a web-based registry for information sources, reasoners, etc., a portable proof specification language (PML [24]) for sharing explanations, and a browser to view and interact with the proof explanation in different formats.

In summary, though there exists various forms of explanation for inferences in DLs, there is no generic solution. The success of the explanation depends on factors such as skill, expertise and background knowledge of the user, and preference for a particular kind of reasoning algorithm. For example, users exposed to resolution would prefer the last approach as opposed to those more comfortable with tableaux-based reasoning. Also, most of the explanation techniques have only been recently applied to DLs, which is not surprising given that OWL became a W3C recommendation in 2004, and it is interesting to see how the techniques evolve to cater to the needs to the OWL user community as it gets more exposed to DL-based knowledge representation.

Recent Developments in Debugging/Repair of DL KBs

In this subsection, we review specialized techniques for debugging and repairing errors in DL knowledge bases. We note that with OWL reaching recommendation status only recently, the area of debugging OWL ontologies, in particular, is a largely unexplored field.

In [70], the authors present a tool, Chimaera, which apart from supporting ontology merging, allows users to run a diagnostic suite of tests across an ontology. The tests include incompleteness tests, syntactic checks and taxonomic analysis, and the results are displayed as an interactive log, which the users can study and explore. The focus here is clearly on detecting style defects, whereas explanation support for semantic defects is fairly weak.

Work has been done on a 'Symptom Ontology' [7] for representing errors and warnings resulting from defects in OWL ontologies, and an implementation is provided in the tool, ConVisor. The authors here do a good job of categorizing commonly occurring symptoms and motivate the significance of creating and exchanging standardized bug reports using a symptom ontology. However, just as in the previous case, their work does not deal with pinpointing the cause of logical inconsistency.

For dealing with inconsistency in DL KBs, broadly two different approaches have been taken. The first is the solution in [6], [91], as seen in the previous section, which involves identifying the source of the inconsistency (MUPS) in the ontology and correcting it manually. This technique has been extended in [93], [35] where the authors use Reiter's Hitting Set algorithm [88] (and subsequently a faster algorithm in [92]) to find a *diagnosis* set, i.e., minimal set of axioms in the ontology whose removal turns it consistent. However, the main drawback here is that the solution focuses simply on turning the ontology coherent without considering the quality of the solution. Also, as noted earlier,

the tableaux-based technique to find the MUPS is limited to unfoldable $\mathcal{ALCF}$ TBoxes.

The second approach is based on phrasing the problem as a belief revision as done in [74]. [71] uses this idea to propose revising the knowledge base to get rid of the inconsistency by rewriting the axioms to preserve semantics, e.g., introducing disjunctions. On a similar note, [54] proposes tolerating inconsistent theories and using a non-classical form of inference to derive meaningful results from a consistent sub-theory.

We propose a hybrid of both approaches, by developing techniques to identify all sources of inconsistency and using metrics based on belief revision such as *minimal impact* to arrive at meaningful repair solutions.

Finally, [105] describes a black-box heuristic approach for debugging OWL, which is similar in principle to the structural analysis algorithms described in [58]. The idea here is to use a pre-defined set of rules to detect commonly occurring error patterns in ontologies based on extensive use-case data (for example, as enumerated in [87]). While such a rule-based heuristic can be fast, it is clearly incomplete.

3.2 Key Theories of Diagnosis and Revision

In this section, we briefly look at two mature and widely accepted theories of diagnosis and revision that relate to the work described in this thesis – Reiter's theory of model-based diagnosis, and the AGM Belief Revision theory.

3.2.1 Reiter's Theory of Diagnosis based on First Principles

In [88], Reiter developed a general theory of diagnosis based on the "first principles" approach, i.e., using a representation language based on first-order logic. A system to be diagnosed is defined by a set of $COMPONENTS$, a system description SD, and a set of observations, OBS. A diagnosis for ($SD, COMPONENTS, OBS$) is defined to be a minimal set $\Delta \subseteq COMPONENTS$ such that

$$SD \cup OBS \cup \{\neg AB(c) | c \in COMPONENTS - \Delta\} \cup \{AB(c) | c \in \Delta\}.$$

is consistent, where AB is a predicate indicating that a component is abnormal. Reiter proposes a characterization of a diagnosis which uses the concept of a *conflict set*. A conflict set for ($SD, COMPONENTS, OBS$) is a set $\{c_1, ..c_k\} \subseteq COMPONENTS$ such that $SD \cup OBS \cup \{\neg AB(c_i) \cup .. \cup \neg AB(c_k)\}$ is inconsistent. A conflict set is minimal iff no proper subset of it is a conflict set. Finally, Reiter uses the notion of hitting sets. A hitting set for a collection of sets C is a set $H \subseteq \bigcup_{S \in C} S$ s.t. $H \cap S \neq \emptyset$ for each $S \in C$. A hitting set for C is minimal, iff no proper subset of it is a hitting set for C.

Two of the main results of Reiter's work are: a theorem which states that the diagnosis for ($SD, COMPONENTS, OBS$) is a minimal hitting set for the collection of conflict sets for ($SD, COMPONENTS, OBS$); and a technique to generate minimal hitting sets using the notion of a *Hitting Set Tree* (HST) that does not require the conflict sets to be minimal.

We have used Reiter's theory of diagnosis in the context of the Axiom Pinpointing Service (Chapter 4), where we employ the HST concept to obtain all the justifications

for an arbitrary entailment of a DL KB. The idea here is that the justifications for the unsatisfiability entailment correspond to minimal conflict sets in the general case, and an algorithm that generates minimal hitting sets can also be used to find all minimal conflict sets (by duality, see proof in Chapter 4, Theorem 4).

3.2.2 AGM Belief Revision Postulates

There has been a body of work on belief revision with roots at least as far back as [36] and subsequently formulated in [1].

The AGM belief revision theory is concerned with formulating postulates to characterize three operations of belief revision: adding a new assertion to a knowledge base ("expansion"); removing an assertion from a knowledge base ("contraction"); adding a new assertion to knowledge base that makes it inconsistent, and adjusting the result to restore consistency ("revision"). Revision can be viewed as a contraction followed by an expansion. The authors express these postulates in a very high-level way.

Two key revision postulates are (Gardenfors and Rott, 1995, p.38):

"(i) The amount of information lost in a belief change should be kept minimal.
(ii) In so far as some beliefs are considered more important or entrenched than others, one should retract the least important ones".

These two postulates are satisfied by our Ontology Repair Service (Chapter 6), i.e., *(i)* translates in our case to removing axioms which drop the least number of entailments from the KB (minimal change), and *(ii)* translates to removing axioms that are of the least *rank* (or importance), based on some manual or automated ranking criteria.

Note that an in-depth analysis of the applicability of the AGM theorem to DLs is beyond the scope of this thesis. For more details, we refer the reader to [34], [60].

Chapter 4
Core Debugging Service: Axiom Pinpointing
4.1 Introduction and Background

As noted in Chapter 1, OWL-DL is a World Wide Web Consortium standard for representing ontologies on the Semantic Web [27]. It is a syntactic variant of the Description Logic $\mathcal{SHOIN}(\mathcal{D})$ [52], with an OWL-DL ontology corresponding to a $\mathcal{SHOIN}(\mathcal{D})$ knowledge base.

DL systems typically offer a set of basic inference services, such as concept classification, concept satisfiability and knowledge base (KB) consistency checking, among others. However, in order to be useful for real-world applications, a DL-based Knowledge Representation (KR) system must expose to the user additional more-sophisticated services. A typical example is the generation of *explanations* for the inferences performed by the reasoner, such as inconsistencies in the KB and entailed subsumption relations in the concept hierarchy. These services are critical, especially with the advent of the Semantic Web, which has exposed Ontology Engineering to a broader audience of users and developers.

A natural question is whether these services can be formalized as *reasoning* services in a way that is both useful and understandable to modelers. In this chapter, we present a novel DL inference service, *Axiom Pinpointing*, that, given a KB and any of its logical consequences, provides the set of all the *justifications* for the entailment to hold in the KB. In this context, we provide a formal notion of justification and propose a set of decision procedures for the axiom pinpointing problem.

Roughly, given a $\mathcal{SHOIN}$ axiom (or assertion) α entailed by a knowledge base $\mathcal{K}$, a justification for α in $\mathcal{K}$ is a minimal fragment $\mathcal{K}' \subseteq \mathcal{K}$ responsible for α to occur. The justification $\mathcal{K}'$ is minimal in the sense that α is a logical consequence of $\mathcal{K}'$, on the one hand, and any proper subset of $\mathcal{K}'$ does not entail α, on the other hand. In general, there may exist various justifications for α in $\mathcal{K}$.

We use a simple example to illustrate this notion. Consider a KB $\mathcal{K}$ composed of the following axioms:

1. $A \sqsubseteq B \sqcap C$
2. $B \sqsubseteq \neg E$
3. $A \sqsubseteq D \sqcap \exists R.E$
4. $D \sqsubseteq C \sqcap \forall R.B$

In the KB above, A, B, C, D, E represent atomic concepts, and R represents an atomic role. In what follows, we will use natural numbers to denote each of these axioms.

We find that $\mathcal{K} \models (A \sqsubseteq C)$. However, the minimal fragments of $\mathcal{K}$ that entail the same subsumption relationship are $\mathcal{K}_1 = \{1\}$ and $\mathcal{K}_2 = \{3, 4\}$. We refer to $\mathcal{K}_1$ and $\mathcal{K}_2$ as the justifications for the subsumption entailment $A \sqsubseteq C$.

Now, while the sample KB considered above is rather small, it is easy to see the significance of the axiom pinpointing service when dealing with large KBs consisting of hundreds or thousands of axioms. By specifying the minimal asserted axiom sets responsible for an entailment, the service can be used to isolate, highlight and explain the cause or basis of the entailment. This is crucial from a debugging standpoint, e.g., given an unsatisfiable concept, the service exposes all and only the axioms that are responsible for the error. In this case, obtaining all the justifications becomes necessary for resolving the error, since at least one erroneous axiom in each of the justification sets needs to be *fixed* in order to make the concept satisfiable.

However, the axiom pinpointing service discussed so far has an inherent *granularity* limitation: it works at the axiom level and does not distinguish the specific *parts of the axiom* responsible for the entailment. Taking our earlier example of the sample KB $\mathcal{K}$, the concept B in the conjunct $B \sqcap C$ in axiom 1 is, in some sense, *irrelevant* for the subsumption $A \sqsubseteq C$, i.e., if the axiom was modified such that only the concept B (in the conjunct) was removed or replaced with another concept, say E, the subsumption $A \sqsubseteq C$ would still hold. Similarly, the concept $\exists R.E$ and the concept $\forall R.B$ in axioms 3 and 4 respectively, are both irrelevant for the entailment $A \sqsubseteq C$. It is important to consider parts of axioms that contribute to an entailment since in a lot of cases, repairing an error involves editing axioms instead of simply removing them.

For this purpose, we introduce the notion of a *KB splitting function*. The idea is to rewrite the axioms in the KB in a convenient normal form and split across conjunctions in the normalized version, e.g., rewriting $A \sqsubseteq C \sqcap D$ as $A \sqsubseteq C, A \sqsubseteq D$. We then extend the axiom pinpointing service to capture (*precise*) justifications in this split version of the KB, which is equivalent to the original KB, though contains "smaller" axioms. In the earlier case, the output of the extended service for the entailment $A \sqsubseteq C$ becomes $\mathcal{K}'_1 = \{A \sqsubseteq C^1\}$ and $\mathcal{K}'_2 = \{A \sqsubseteq D^3, D \sqsubseteq C^4\}$, where the superscripts denote the asserted axiom that each of the split axioms has been derived from.

We devise a set of algorithms for axiom pinpointing and provide proofs of correctness and completeness. The algorithms are mainly of two types:

1. *Reasoner dependent (or Glass-box)* algorithms are built on existing tableau-based decision procedures for expressive Description Logics. Their implementation requires a thorough and non-trivial modification of the internals of the reasoner.
2. *Reasoner independent (or Black-box)* algorithms use the DL reasoner solely as a *subroutine* and the internals of the reasoner do not need to be modified. The reasoner behaves as a "Black-box" that accepts, as usual, a concept and a KB as input and returns an affirmative or a negative answer, depending on whether the concept is satisfiable or not w.r.t. the KB. In order to obtain the justifications, the axiom pinpointing algorithm selects the appropriate inputs to the DL reasoner and interprets its output accordingly.

Glass-box algorithms typically affect many aspects of the internals of the reasoner and strongly depend on the DL under consideration.

Black-box algorithms typically require many satisfiability tests, but they can be easily and robustly implemented, since they only rely on the availability of a sound and

complete reasoner for such a DL. Consequently, using a Black-box approach, the service can also be implemented on reasoners that are based on techniques other than tableaux, such as resolution.

Finally, we also investigate *hybrid* algorithms, which combine Glass-box and Black-box approaches to obtain sound and complete solutions relatively easily, i.e., without dealing with complicated implementation issues. The idea here is to use one of the approaches to reduce the problem space significantly and the other as a post-processing step to obtain the correct solution.

The remainder of this chapter is organized as follows: in Section 4.1.1, we formally define justification of entailments and show how it is closely related to the notion of MUPS as described in [91]. We then present two versions (Black-box / Hybrid) of an algorithm to compute a single justification (Section 4.2) and extend it to find all justifications (Section 4.3). In Section 4.4, we formally define precise justifications based on the notion of splitting KBs and show how the algorithms described in the earlier sections can be modified to enhance the output granularity.

4.1.1 Justification of Entailments and MUPS

In this section, we provide a formal definition of justifications and introduce the notion of a MUPS, as described in [91]. Finally, we show how justifications and MUPS relate to each other for the description logic $\mathcal{SHOIN}$.

We start with the definition of justifications.

Definition 6 *(JUSTIFICATION)*
Let $\mathcal{K} \models \alpha$ where α is a sentence. A fragment $\mathcal{K}' \subseteq \mathcal{K}$ is a justification for α in $\mathcal{K}$ if $\mathcal{K}' \models \alpha$, and $\mathcal{K}'' \not\models \alpha$ for every $\mathcal{K}'' \subset \mathcal{K}'$.

We denote by $JUST(\alpha, \mathcal{K})$ the set of all the justifications for α in $\mathcal{K}$. Given α and $\mathcal{K}$, the *Axiom Pinpointing* inferential service is the problem of computing $JUST(\alpha, \mathcal{K})$

MUPS are formally defined as follows:

Definition 7 *(MUPS) Let C be a concept, which is unsatisfiable w.r.t. a knowledge base $\mathcal{K}$. A fragment $\mathcal{K}' \subseteq \mathcal{K}$ is a MUPS of C in $\mathcal{K}$ if C is unsatisfiable in $\mathcal{K}'$, and C is satisfiable in every $\mathcal{K}'' \subset \mathcal{K}'$.*

We denote by $MUPS(C, \mathcal{K})$ the set of all the MUPS for C in $\mathcal{K}$. When the KB we are referring to is clear from the context, we will relax the notation and use $MUPS(C)$ instead.

The relationship between MUPS and justifications is established by Theorem 1. The simple theorem is based on the following result [47]: given a $\mathcal{SHOIN}$ knowledge base $\mathcal{K}$, for every sentence (axiom or assertion) α entailed by $\mathcal{K}$, there is always a concept C_α that is unsatisfiable w.r.t $\mathcal{K}$. Conversely, given any concept C that is unsatisfiable w.r.t. $\mathcal{K}$, there is always a sentence α_C that is entailed by $\mathcal{K}$. Consequently, given a $\mathcal{SHOIN}$ KB, the problem of finding all the MUPS for an unsatisfiable concept and the problem of finding all the justifications for a given entailment can be reduced to each other.

Theorem 1 *Let $\mathcal{K}$ be a knowledge base, α be a sentence and let C_α be a concept s.t.:*

For every KB $\mathcal{K}' \subseteq \mathcal{K}$, $\mathcal{K}' \models \alpha \Leftrightarrow C_\alpha$ is unsatisfiable w.r.t. $\mathcal{K}'$

Then, $JUST(\alpha, \mathcal{K}) = MUPS(C_\alpha, \mathcal{K})$

Proof

Let $\mathcal{K}' \in JUST(\alpha, \mathcal{K})$, then $\mathcal{K}' \models \alpha$ and $\mathcal{K}'' \not\models \alpha$ for every $\mathcal{K}'' \subset \mathcal{K}'$. From the relationship between C_α and α, we have that C_α is unsatisfiable w.r.t. $\mathcal{K}'$ and it is satisfiable w.r.t. every $\mathcal{K}'' \subset \mathcal{K}'$ then, by definition of MUPS, $\mathcal{K}' \in MUPS(C_\alpha, \mathcal{K})$.

Conversely, let $\mathcal{K}' \in MUPS(C_\alpha, \mathcal{K})$, then C_α is unsatisfiable in $\mathcal{K}'$ and C_α is satisfiable in every $\mathcal{K}'' \subset \mathcal{K}'$. From the relationship between C_α and α, we have that $\mathcal{K}' \models \alpha$ and $\mathcal{K}'' \not\models \alpha$ for every $\mathcal{K}'' \subset \mathcal{K}'$ and thus $\mathcal{K}' \in JUST(\mathcal{K}, \alpha)$

□

In the remainder of this chapter, we shall restrict our attention, without loss of generality, to the problem of finding all the MUPS for an unsatisfiable concept w.r.t to a $\mathcal{SHOIN}$ KB.

Note: The notion of justifications can be easily extended to include justifications for an *inconsistent* KB, i.e., minimal sets of axioms responsible for making a KB logically inconsistent. Also, all the ensuing algorithms for finding justifications for unsatisfiability entailments are directly applicable to finding justifications for inconsistency. This should be no surprise as unsatisfiability detection is performed by attempting to generate an inconsistent ontology.

4.2 Computing a Single Justification

4.2.1 Black Box: Simple Expand-Shrink Strategy

In this section, we describe a Black-box solution to the problem of finding a single MUPS of an unsatisfiable concept. The algorithm we describe is reasoner-independent, in the sense that the DL reasoner is solely used as an oracle to determine concept satisfiability w.r.t. a knowledge base.

This algorithm, which we refer to as SINGLE_MUPS$_{Black-Box}$(C, $\mathcal{K}$), shown in Table 4.1, is composed of two main parts: in the first loop, the algorithm generates an empty KB $\mathcal{K}'$ and inserts into it axioms from $\mathcal{K}$ in each iteration, until the input concept C becomes unsatisfiable w.r.t $\mathcal{K}'$. In the second loop, the algorithm removes an axiom from $\mathcal{K}'$ in each iteration and checks whether the concept C turns satisfiable w.r.t. $\mathcal{K}'$, in which case the axiom is reinserted into $\mathcal{K}'$. The process continues until all axioms in $\mathcal{K}'$ have been tested.

Obviously, a key component of the algorithm above is selecting *which* axioms to add into $\mathcal{K}'$ in the first segment of the algorithm. In our implementation, we start by inserting the concept definition axioms into $\mathcal{K}'$ and slowly expand it to include axioms

Algorithm: SINGLE_MUPS$_{Black-Box}$
Input: KB $\mathcal{K}$, Unsatisfiable concept C
Output: KB $\mathcal{K}$'
$\mathcal{K}' \leftarrow \emptyset$
while (C is satisfiable w.r.t $\mathcal{K}$') **do**
select a set of axioms $s \subseteq \mathcal{K}/\mathcal{K}'$
$\mathcal{K}' \leftarrow \mathcal{K}' \cup s$
for each axiom $k' \in \mathcal{K}'$, **do**
$\mathcal{K}' \leftarrow \mathcal{K}' - \{k'\}$
if (C is satisfiable w.r.t. $\mathcal{K}'$), **then**
$\mathcal{K}' \leftarrow \mathcal{K}' \cup \{k'\}$

Table 4.1: Singe MUPS (Black Box)

of structurally related concepts, roles and individuals[1]. Moreover, while expanding the fragment $\mathcal{K}'$ by iteratively considering a set of axioms $s \subseteq \mathcal{K}$, we establish a small initial limit on the size of s and slowly increase this limit with each iteration.

Also, we have implemented an additional optimization that has proved effective: after the first stage, we perform a fast pruning of $\mathcal{K}'$ before proceeding to the second stage. The goal is to reduce the size of the input to the second stage. For this purpose, we use a window of n axioms, slide this window across the axioms in $\mathcal{K}'$, remove axioms from $\mathcal{K}'$ that lie within the window and determine if the concept is still unsatisfiable in the new $\mathcal{K}'$. If the concept turns satisfiable, we can conclude that at least one of the n axioms removed from $\mathcal{K}'$ is responsible for the unsatisfiability and hence we insert the n axioms back into $\mathcal{K}'$. However, if the concept still remains unsatisfiable, we can conclude that all n axioms are irrelevant and we remove them from $\mathcal{K}'$.

4.2.2 Hybrid: Tableau-based Decision Procedure (Tableau-Tracing)

As seen in the previous section, the Black-box approach to find a single element of $\mathsf{MUPS}(\mathsf{C}, \mathcal{K})$ works by expanding an empty KB using axioms from the original KB, till the concept is unsatisfiable in it, and then shrinking or pruning this KB to arrive at a minimal set of axioms responsible for the unsatisfiability. Note that the second stage (pruning) can be directly applied to the original KB itself, except that the approach may be practically unfeasible for large KBs with thousands of axioms.

In this section, we present a Glass-box algorithm for obtaining a much smaller set of axioms than the original KB in which the concept is unsatisfiable. This algorithm can be used in place of the first step in the Black-box technique seen earlier to obtain a single justification relatively quickly, thus making the complete solution an hybrid one.

The algorithm is based on the tableau decision procedure for concept satisfiability in $\mathcal{SHOIN}$ recently presented in [52]. DL tableau-based algorithms decide the satisfiability of a (possibly complex) concept C w.r.t a KB $\mathcal{K}$ by trying to construct (an abstraction of) a common model for C and $\mathcal{K}$, called a *completion graph*, which is constructed by re-

[1]In the case of an inconsistent ontology, we start by inserting individual assertions, especially considering axioms which assert distinctness of individuals. Note that in this case, there is no unsatisfiable concept C input to the algorithm.

peatedly applying a set of *expansion rules*. DL tableau algorithms are non-deterministic. Whenever a contradiction is encountered, a DL reasoner will either backtrack and select a different non-deterministic choice, or report the inconsistency and terminate, if no choice remains to be explored.

Obviously, in our problem, the goal is no longer constructing a model for the input, but identifying which axioms in the input ontology are responsible for the contradictions that prevent the model from being built.

Before we proceed to the formal description of the algorithm, we provide an example to illustrate the main intuitions. We assume some familiarity of the reader with the logic $\mathcal{SHOIN}$ as well as with tableaux-based reasoning algorithms for expressive DLs as presented in Chapter 2.

An Example

Let us consider a KB $\mathcal{K}$ composed of the 10 axioms, denoted with natural numbers:

1. $A \sqsubseteq \exists R.D \sqcap B$	6. $C \sqsubseteq \neg E$
2. $B \sqsubseteq \geq 1.R$	7. $D \sqsubseteq F$
3. $B \sqsubseteq F$	8. $C \sqsubseteq \forall R.\neg D$
4. $F \sqsubseteq \neg E$	9. $D \sqsubseteq \neg B$
5. $A \sqsubseteq C \sqcup D$	10. $E \sqsubseteq \forall R.F$

The concept A is unsatisfiable w.r.t $\mathcal{K}$, and $\mathsf{MUPS}(\mathsf{A}, \mathcal{K}) = \{\{1, 5, 8, 9\}\}$. Our strategy is to keep track of the axioms from the KB responsible for each change in the completion graph, namely, the addition of a particular concept (or role) to the label of a specific node (or edge), or the detection of a contradiction (clash) in the label of a node. In the Figure, this is denoted by the superscript of each concept in the node labels. We generically refer to *tracing* as the process of tagging concepts, roles and clashes in the completion graph with sets of axioms in the KB.

The algorithm works on a tree $\mathbf{T}$ of completion graphs. Given the input $A, \mathcal{K}$, the tree is initialized with a single completion graph $\mathbf{G_0}$ containing a node x with A in its label. This initial graph is incrementally built using the set of available *expansion rules*. [2]

The application of non-deterministic rules results in the addition of new completion graphs as leaves of $\mathbf{T}$, one for each different non-deterministic choice. The algorithm terminates when all the leaves of the tree contain a clash. Upon termination, the trace of the detected clashes in the leaves of $\mathbf{T}$ yield a smaller set of axioms that contain at least one element of $\mathsf{MUPS}(\mathsf{A}, \mathcal{K})$.

In our example, the algorithm starts with graph $\mathbf{G_0}$ and applies the $unfolding$, $\sqcap$ rules to axiom 1 which adds concepts $\exists R.D$, B to $\mathcal{L}(x)$; then, it applies the $\exists$ rule which generates an R-successor y of x, and adds concept D to the label of y.

The algorithm now applies the $unfolding$ rule to axioms $2, 3, 4, 5$, the last of which adds the disjunction $C \sqcup D$ to $\mathcal{L}(x)$. It is forced to make a non-deterministic choice due to the application of the $\sqcup$ rule. This creates two new completion graphs $\mathbf{G_1}, \mathbf{G_2}$ (shown in Figure 4.1) each containing a separate choice of the disjunction $C \sqcup D$ in axiom 3.

[2] For a full specification of the expansion rules, we refer the reader to the next Section.

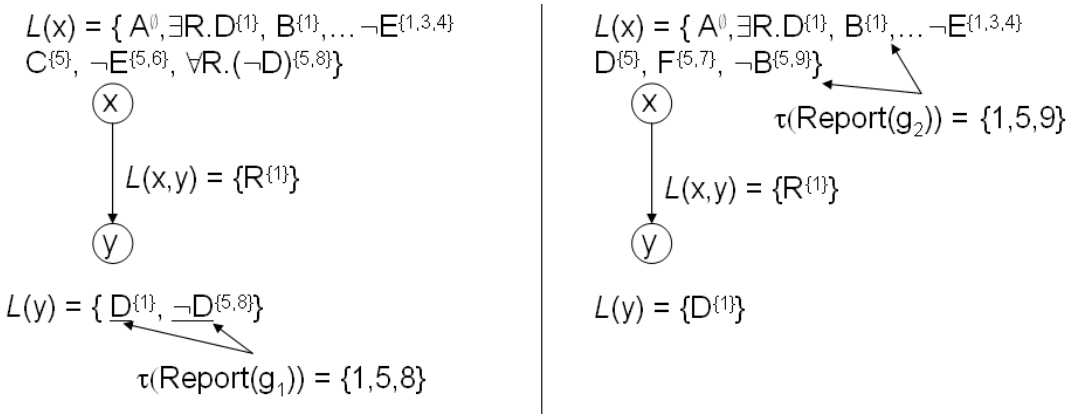

Figure 4.1: **Tableau Tracing:** Completion Graphs $\mathbf{G_1}, \mathbf{G_2}$ created after applying non-deterministic rules and added as leaves of $\mathbf{T}$.

Both graphs are then added as leaves of $\mathbf{T}$. Since no more rules are applicable in $\mathbf{G_0}$ the algorithm now starts expanding $\mathbf{G_1}$.

In $\mathbf{G_1}$, the presence of $C \in \mathcal{L}(x)$ causes the application of the $unfolding$ rule to axioms $6, 8$, the latter yields a clash since both D and its negation are present in the label of node y. The trace of this clash is computed by considering the axiom sets responsible for adding both $D, \neg D \in \mathcal{L}(y)$, in this case the set $\{1, 5, 8\}$.

Since a clash is found in $\mathbf{G_1}$, the algorithm moves to $\mathbf{G_2}$ and starts expanding it. It finds a new clash in $\mathbf{G_2}$ after applying the $unfolding$ rule to axioms $7, 9$, as both B and its negation are present in $\mathcal{L}(x)$, and the trace of this clash is $\{1, 5, 9\}$. The algorithm now concludes that A is unsatisfiable since all the leaves of the tree contain a clash. The output is computed by taking the union of the traces of all clashes present in the leaves of $\mathbf{T}$, i.e., $\{1, 5, 8, 9\}$. In this case, the output corresponds to a MUPS directly.

In order to ensure a smaller yet correct output, we impose an ordering among the deterministic rules, i.e., $unfolding$ and CE rules are only applied when *no other* deterministic rule is applicable. The rationale for this strategy relies on the definition of justification that establishes minimality w.r.t. the number of axioms considered; the new rules cause new axioms to be considered in the tracing process and additional axioms should only be considered if strictly necessary.

Thus, we have introduced two main variations to the standard algorithm for concept satisfiability: first, we *keep track* of axiom sets responsible for various changes on the completion graphs and compute the output of the algorithm from the trace of each clash found in the leaves of the tree; second, we establish additional conditions in the order of rule application for ensuring the output is as small as possible.

Definition of the Algorithm

In this section, we provide a formal description of the tableau algorithm for computing a single MUPS. The algorithm runs on a tree $\mathbf{T} = (W, \prec)$ of completion graphs and returns a set $S \in \mathsf{MUPS}(\mathsf{C}, \mathcal{K})$.

A completion graph for a concept C with respect to $\mathcal{K}$ is a directed graph $\mathbf{G} =$

$(V, E, \mathcal{L}, \dot{\neq})$. Each node $x \in V$ is labeled with a set of concepts $\mathcal{L}(x)$ and each edge $e = \langle x, y \rangle$ with a set $\mathcal{L}(e)$ of role names. The binary predicate $\dot{\neq}$ is used for recording inequalities between nodes. If $\langle x, y \rangle \in E$, then y is called a *successor* of x and x a *predecessor* of y. *Ancestor* is the transitive closure of predecessor and *descendant* the transitive closure of successor. A node y is an R-successor of x as given in [52].

The set S is initially empty and the initial tree contains a single graph $\mathbf{G} = (\{v_0, ..., v_l\}, \emptyset, \mathcal{L}, \emptyset)$, where $\mathcal{L}(v_i) = \{o_i\}$ for $1 \leq i \leq l$ and $o_1, ..., o_l$ the individual names occurring in $\mathcal{K}$ and C. The graph $\mathbf{G}$ is then expanded by repeatedly applying the rules in Table 4.2.

We keep a set Δ of completion graphs and a set Σ of edges to be added at the next level of the tree. The application of a non-deterministic rule results in the creation of a new completion graph, added to Δ, for each possible non-deterministic choice. When all the graphs in the current level of $\mathbf{T}$ have been expanded, the algorithm determines which graphs in Δ need to be added as leaves of $\mathbf{T}$, as follows: for each $\mathbf{G}$ in the current level of $\mathbf{T}$ that contains a clash and each edge $\mathbf{G} \prec \mathbf{G}' \in \Sigma$, remove $\mathbf{G}'$ from Δ and $\mathbf{G} \prec \mathbf{G}'$ from Σ; at the end of this process, if $\Delta = \emptyset$, then the algorithm terminates; otherwise, the algorithm adds the remaining graphs in Δ and edges in Σ to the tree, i.e. $W := W \cup \Delta$ and $\prec := \prec \cup \Sigma$ and initializes again Δ and Σ to the empty set before starting the expansion of the next level of $\mathbf{T}$. Since the input concept is unsatisfiable w.r.t. the input KB, the set Δ will become empty after exploring a finite number of levels in $\mathbf{T}$ and, thus, the algorithm will terminate.

We have introduced two additional rules with respect to the ones presented in Chapter 2: The *unfolding* rule adds the definition of a concept C to the label $\mathcal{L}(x)$ of a node x whenever C is contained in $\mathcal{L}(x)$. The GCI rule (CE) adds the disjunction $\neg C \sqcup D$ to the label of a node x if the GCI $C \sqsubseteq D$ is contained in $\mathcal{K}$. These rules are required in order to identify which axioms in $\mathcal{K}$ are influencing the expansion of $\mathbf{G}$. The remaining rules remain unaltered w.r.t. [52], except for the additional conditions to compute the tracing functions. For ensuring termination, we establish the same priorities for rule application as in [52]; concerning the additional rules, we enforce that the $unfolding$ and CE rules are only applied when *no other* non-deterministic rule is applicable, as seen in the example of the previous Section. Finally, we adopt the same mechanism for cycle detection in the graph expansion as in [52], namely *pair-wise blocking*.

The application of the expansion rules triggers a set of *events* that change the state of the completion graph, or the flow of the algorithm: **1**: $\mathsf{Add}(\mathsf{C}, \mathsf{x})$ is the action of adding a concept C to $\mathcal{L}(x)$; **2)** $\mathsf{Add}(\mathsf{R}, \langle \mathsf{x}, \mathsf{y} \rangle)$ inserts a role R into $\mathcal{L}(\langle x, y \rangle)$; **3)** $\mathsf{Merge}(\mathsf{x}, \mathsf{y})$ is the action of *merging* the nodes x, y; **4)** $\dot{\neq}(\mathsf{x}, \mathsf{y})$ adds the inequality $x \dot{\neq} y$; **5)** $\mathsf{Report}(\mathsf{g})$ represents the detection of a clash g. We denote by $\mathcal{E}$ the events recorded during the execution of the algorithm.

The graph $\mathbf{G}$ contains a *clash* if either $\{C, \neg C\} \subseteq L(x)$ for some concept C and node x, or the events $\mathsf{Merge}(\mathsf{x}, \mathsf{y})$ and $\dot{\neq}(\mathsf{x}, \mathsf{y})$ belong to $\mathcal{E}$.

We introduce a *tracing function*, which keeps track of the axioms responsible for the changes in the graph to occur. The *tracing function* τ maps each event $e \in \mathcal{E}$ to a fragment of $\mathcal{K}$. The function τ is initialized as empty and defined by construction using the expansion rules[3]. For a clash g of the form $\{C, \neg C\} \subseteq \mathcal{L}(x)$, $\tau(\mathsf{Report}(\mathsf{g})) =$

[3]In the rules shown in Table 4.2, we have abbreviated $\tau(\mathsf{Add}(\mathsf{C}, \mathsf{x})$ and $\tau(\mathsf{Add}(\mathsf{R}, \langle \mathsf{x}, \mathsf{y} \rangle))$ by $\tau(C, x)$

unfold-rule: **if** $A \in \mathcal{L}(x)$, A atomic, $(A \sqsubseteq D) \in \mathcal{K}$:
if $D \notin \mathcal{L}(x)$, $\mathsf{Add}(\mathsf{D}, \mathcal{L}(\mathsf{x}))$
$\tau(D, x) := (\tau(A, x) \cup \{A \sqsubseteq D\})$
CE-rule: **if** $(C \sqsubseteq D) \in \mathcal{K}$, C not atomic, x not blocked,
if $(\neg C \sqcup D) \notin \mathcal{L}(x)$, $\mathsf{Add}(\neg C \sqcup D, x))$, $\tau((\neg C \sqcup D), x) := \{C \sqsubseteq D\}$
$\sqcap$-rule: **if** $(C_1 \sqcap C_2) \in \mathcal{L}(x)$, x not indirectly blocked,
if $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$, $\mathsf{Add}(\{\mathsf{C_1}, \mathsf{C_2}\}, \mathsf{x}))$.
$\tau(C_i, x) := \tau((C_1 \sqcap C_2), x)$
$\sqcup$-rule: **if** $(C_1 \sqcup C_2) \in \mathcal{L}(x)$, x not ind. blocked,
if $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$, generate graphs $\mathbf{G}_i := \mathbf{G}$ for each $i \in \{1, 2\}$
$\Delta := \Delta \cup \{\mathbf{G}_1, \mathbf{G}_2\}$; $\Sigma := \Sigma \cup \{\mathbf{G} \prec \mathbf{G}_1, \mathbf{G} \prec \mathbf{G}_2\}$
$\mathsf{Add}(\mathsf{C_i}, \mathsf{x})$ in $\mathbf{G}_i$ for each $i \in \{1, 2\}$
$\tau(C_i, x) := \tau((C_1 \sqcup C_2), x)$
$\exists$-rule: **if** $\exists S.C \in \mathcal{L}(x)$, x not blocked,
if no S-neighbor y with $C \in \mathcal{L}(y)$, create y, $\mathsf{Add}(\mathsf{S}, \langle \mathsf{x}, \mathsf{y} \rangle)$, $\mathsf{Add}(\mathsf{C}, \mathsf{y})$
$\tau(S, \langle x, y \rangle) := \tau((\exists S.C), x)$
$\tau(C, y) := \tau((\exists S.C), x)$
$\forall$-rule: **if** $\forall S.C \in \mathcal{L}(x)$, x not ind. blocked, y S-neighbor of x:
if $C \notin \mathcal{L}(y)$, $\mathsf{Add}(\mathsf{C}, \mathsf{y})$
$\tau(C, y) := (\tau((\forall S.C), x) \cup \tau(S, \langle x, y \rangle))$
$\forall^+$-rule: **if** $\forall S.C \in \mathcal{L}(x)$, x not ind. blocked, y R-neighbor of x with $\mathsf{Trans}(\mathsf{R})$ and $R \sqsubseteq S$:
if $\forall S.C \notin \mathcal{L}(y)$, $\mathsf{Add}(\forall \mathsf{S.C}, \mathsf{y})$
$\tau((\forall S.C), y) := \tau((\forall S.C), x) \cup (\tau(R, \langle x, y \rangle) \cup \{\mathsf{Trans(R)}\} \cup \{\mathsf{R} \sqsubseteq \mathsf{S}\})$
$\geq$-rule: **if** $(\geq nS) \in \mathcal{L}(x)$, x not blocked:
if no safe S-neighbors $y_1, .., y_n$ of x with $y_i \neq y_j$, create $y_1, .., y_n$; $\mathsf{Add}(\mathsf{S}, \langle \mathsf{x}, \mathsf{y_i} \rangle)$; $\neq(\mathsf{y_i}, \mathsf{y_j})$
$\tau(S, \langle x, y_i \rangle) := \tau((\geq nS), x)$
$\tau(\neq(\mathsf{y_i}, \mathsf{y_j})) := \tau((\geq \mathsf{nS}), \mathsf{x})$
$\leq$-rule: **if** $(\leq nS) \in \mathcal{L}(x)$, x not ind. blocked, $y_1, .., y_m$ S-neighbors of x, $m > n$:
For each possible pair y_i, y_j, $1 \leq i, j \leq m; i \neq j$:
Generate a graph $\mathbf{G}'$; $\Delta := \Delta \cup \{\mathbf{G}'\}$; $\Sigma := \Sigma \cup \{\mathbf{G} \prec \mathbf{G}'\}$
$\tau(\mathsf{Merge}(\mathsf{y_i}, \mathsf{y_j})) := (\tau((\leq \mathsf{nS}), \mathsf{x}) \cup \tau(\mathsf{S}, \langle \mathsf{x}, \mathsf{y_1} \rangle) .. \cup \tau(\mathsf{S}, \langle \mathsf{x}, \mathsf{y_m} \rangle))$
if y_j a nominal node, $\mathsf{Merge}(\mathsf{y_i}, \mathsf{y_j})$ in $\mathbf{G}'$,
else if y_i a nominal node or ancestor of y_j, $\mathsf{Merge}(\mathsf{y_j}, \mathsf{y_i})$,
else $\mathsf{Merge}(\mathsf{y_i}, \mathsf{y_j})$ in $\mathbf{G}'$
if y_i is merged into y_j, for each concept C_i in $\mathcal{L}(y_i)$,
$\tau(\mathsf{Add}(\mathsf{C_i}, \mathcal{L}(\mathsf{y_j})) := \tau(\mathsf{Add}(\mathsf{C_i}, \mathcal{L}(\mathsf{y_i})) \cup \tau(\mathsf{Merge}(\mathsf{y_i}, \mathsf{y_j}))$
(similarly for roles merged, and correspondingly for concepts in y_j if merged into y_i)
O-rule: **if**, $\{o\} \in \mathcal{L}(x) \cap \mathcal{L}(y)$ and not $x \doteq y$, **then** $\mathsf{Merge}(\mathsf{x}, \mathsf{y})$.
$\tau(\mathsf{Merge}(\mathsf{x}, \mathsf{y})) := \tau(\{\mathsf{o}\}, \mathsf{x}) \cup \tau(\{\mathsf{o}\}, \mathsf{y})$
For each concept C_i in $\mathcal{L}(x)$, $\tau(\mathsf{Add}(\mathsf{C_i}, \mathcal{L}(\mathsf{y})) := \tau(\mathsf{Add}(\mathsf{C_i}, \mathcal{L}(\mathsf{x})) \cup \tau(\mathsf{Merge}(\mathsf{x}, \mathsf{y}))$
(similarly for roles merged, and correspondingly for concepts in $\mathcal{L}(y)$)
NN-rule: **if** $(\leq nS) \in \mathcal{L}(x)$, x nominal node, y blockable S-predecessor of x and there is no m
s.t. $1 \leq m \leq n, (\leq mS) \in \mathcal{L}(x)$ and there exist m nominal S-neighbors $z_1, ...z_m$ of x s.t. $z_i \neq z_j$, $1 \leq i \leq j \leq m$,
then generate new $\mathbf{G_m}$ for each m, $1 \leq m \leq n$, add $\Delta := \Delta \cup \{\mathbf{G_m}\}$; $\Sigma := \Sigma \cup \{\mathbf{G} \prec \mathbf{G_m}\}$
and do the following in each $\mathbf{G_m}$:
$\mathsf{Add}(\leq \mathsf{mS}, \mathsf{x})$, $\tau((\leq mS), x) := \tau((\leq nS), x) \cup \tau(S, \langle y, x \rangle)$
create $y_1, ...y_m$; Add $y_i \neq y_j$ for $1 \leq i \leq j \leq m$. $\tau(\neq(\mathsf{y_i}, \mathsf{y_j})) := \tau((\leq \mathsf{nS}), \mathsf{x}) \cup \tau(\mathsf{S}, \langle \mathsf{y}, \mathsf{x} \rangle)$
$\mathsf{Add}(\mathsf{S}, \langle \mathsf{x}, \mathsf{y_i} \rangle)$; $\mathsf{Add}(\{\mathsf{o_i}\}, \mathsf{y_i})$:
$\tau(S, \langle x, y_i \rangle) := \tau((\leq nS), x) \cup \tau(S, \langle y, x \rangle)$ $\tau(\{o_i\}, y_i) := \tau((\leq nS), x) \cup \tau(S, \langle y, x \rangle$

Table 4.2: Modified Tableau Expansion Rules with Tracing

$\tau(\mathsf{Add}(\mathsf{C},\mathsf{x})) \cup \tau(\mathsf{Add}(\neg\mathsf{C},\mathsf{x}))$. The trace for a clash of the form $\mathsf{Merge}(\mathsf{x},\mathsf{y}), \neq(\mathsf{x},\mathsf{y}) \in \mathcal{E}$ is defined identically.

The algorithm terminates when all the leaves of the tree contain a clash and there is no way to apply the non-deterministic rules to generate new leaves. If $g_1, ..g_n$ are the clashes in each of the leaves of the tree and $\tau(\mathsf{Report}(\mathsf{g_i})) = \{\mathsf{s_{g_i}}\}$, the output of the algorithm is $S' = \bigcup_{i\in\{1,\ldots,n\}} s_{g_i}$, which is then pruned to give a final set S using the Black-box approach seen in Table 4.1.

The output of the complete hybrid algorithm is guaranteed to be a MUPS($C, \mathcal{K}$), as established by the following theorem:

Theorem 2 *Let C be an unsatisfiable concept w.r.t. $\mathcal{K}$ and let S be the output of the hybrid algorithm with input C, $\mathcal{K}$, then $S \in \mathsf{MUPS}(\mathsf{C},\mathcal{K})$*

Proof (Sketch) We need to prove that the output of the tableau algorithm S' (before it is pruned) includes at least one MUPS(C, $\mathcal{K}$), i.e., C is unsatisfiable w.r.t S'.

Let $\mathcal{E}$ be the sequence of events generated by the tableau algorithm with inputs $C, \mathcal{K}$. Now suppose (C, S') are inputs to the tableau algorithm and Δ', $\mathcal{E}'$ be the corresponding sets of completion graph and events generated. For each event $e_i \in \mathcal{E}$, it is possible to perform e_i in the same sequence as before in $\mathcal{E}'$. This is because for each event e_i, the set of axioms in $\mathcal{K}$ responsible for e_i have been included in the output S' by construction of the tracing function τ in Table 4.2. (Note that there are cases where additional axioms are also included in τ, e.g., during the $\leq n.R$ rule, where axioms responsible each of the R successor edges are considered). Thus, given $\mathcal{E}' = \mathcal{E}$, a clash occurs in each of the completion graphs in Δ' and the algorithm finds C unsatisfiable w.r.t S'.

□

The complexity of concept satisfiability checking in $\mathcal{SHOIN}$ is 2NExpTime [103]. The changes we have introduced for axiom tracing occur in either constant or linear time. Thus, the complexity of the tableau tracing algorithm minus the final pruning stage remains the same.

4.3 Computing All Justifications

In this section, we describe a technique based on Reiter's Hitting Set Tree Algorithm that is used to compute all the MUPS of an unsatisfiable concept, assuming we have a procedure to compute any one arbitrary MUPS.

In what follows, we briefly introduce Hitting Sets and Reiter's algorithm and show their applicability to our problem.

4.3.1 The Hitting Set Problem and Reiter's Algorithm

Let us consider a set U, the *universal set*, and a set $S \subseteq \mathcal{P}U$ of *conflict sets*, where $\mathcal{P}$ denotes the powerset operator. The set $T \subseteq U$ is a *hitting set* for S if each $s_i \in S$ contains at least one element of T, i.e. if $s_i \cap T \neq \emptyset$ for all $1 \leq i \leq n$. We say that T is a *minimal hitting set* for S if T is a hitting set for S no $T' \subset T$ is a hitting set for

and $\tau(R, \langle x, y\rangle)$ respectively.

S. The *Hitting Set Problem* with input S, U is to compute all the minimal hitting sets for S. The problem is of interest to many kinds of *diagnosis* tasks and has found numerous applications.

Given a collection S of conflict sets, Reiter's algorithm constructs a labeled tree called *Hitting Set Tree* (HST). Nodes in an HST are labeled with a set $s \in S$. If $H(v)$ is the set of edge labels on the path from the root of the HST to the node v, then the label for v is any $s \in S$ such that $s \cap H(v) = \emptyset$, if such a set exists. If s is the label of v, then for each element $\sigma \in s$, v has a successor w connected to v by an edge with σ in its label. If the label of v is the empty set, then $H(v)$ is a hitting set for S.

4.3.2 Hitting Sets and Axiom Pinpointing

In this section, we establish the relationship between the Hitting Set and the Axiom Pinpointing problems.

Our approach is based on the following result:

Theorem 3 *Let C be unsatisfiable w.r.t $\mathcal{K}$ and let $\mathcal{K}' \subset \mathcal{K}$, with $\mathcal{K}' = \mathcal{K} - \mathcal{H}$, then:*

1. *C is satisfiable w.r.t. $\mathcal{K}'$ if and only if $\mathcal{H}$ is a Hitting Set for $MUPS(C, \mathcal{K})$ w.r.t. $\mathcal{K}$*
2. *$\mathcal{H}$ is a minimal Hitting Set for $MUPS(C, \mathcal{K})$ w.r.t. $\mathcal{K}$, if and only if there is no $\mathcal{H}' \subset \mathcal{H}$ such that C is satisfiable w.r.t. $\mathcal{K} - \mathcal{H}'$.*

Proof

1. Suppose that C is satisfiable w.r.t. $\mathcal{K}'$ but $\mathcal{H}$ is not a hitting set for $MUPS(\mathcal{K}, C)$ w.r.t. $\mathcal{K}$. Then, by definition of hitting set, there is a set $S \in MUPS(C, \mathcal{K})$ s.t. $S \cap \mathcal{H} = \emptyset$. Thus, $S \subseteq \mathcal{K}'$ and, by definition of MUPS, C is unsatisfiable w.r.t. S. By monotonicity, C is also unsatisfiable w.r.t. $\mathcal{K}'$, which yields a contradiction. Assume now that $\mathcal{H}$ is a hitting set for $MUPS(\mathcal{K}, C)$, but C is unsatisfiable w.r.t. $\mathcal{K}'$. By definition of Hitting Set, for every $S \in MUPS(C, \mathcal{K})$, $S \cap \mathcal{H} \neq \emptyset$. Thus, there is no $S \in MUPS(C, \mathcal{K})$ s.t. $S \subseteq \mathcal{K}'$ which implies that C is indeed satisfiable w.r.t. $\mathcal{K}'$.
2. Suppose $\mathcal{H}$ is a minimal Hitting Set for $MUPS(C, \mathcal{K})$ w.r.t. $\mathcal{K}$. Then, by definition of minimal hitting set, no $\mathcal{H}' \subset \mathcal{H}$ is a Hitting Set. By 1) C is satisfiable w.r.t. $\mathcal{K} - \mathcal{H}'$ for every $\mathcal{H}' \subset \mathcal{H}$. The converse is also straightforward.

□

The intuition behind the theorem relies on the fact that, in order to make a concept C satisfiable w.r.t. a knowledge base $\mathcal{K}$, one needs to remove from $\mathcal{K}$ *at least* one axiom from each of the elements of $MUPS(C, \mathcal{K})$.

Our aim is to use Theorem 3 and Reiter's Hitting Set Trees to obtain $MUPS(C, \mathcal{K})$ out of a single set $S \in MUPS(C, \mathcal{K})$.

4.3.3 A Simple Example

In order to describe the main intuitions, let us consider a KB $\mathcal{K}_2$ with ten axioms and some unsatisfiable concept C. For the purpose of this example, we denote the axioms in $\mathcal{K}_2$ with natural numbers. Suppose that we are provided an algorithm SINGLE_MUPS(C,$\mathcal{K}$) that retrieves an arbitrary element of $MUPS(C, \mathcal{K})$; an example of

such a procedure could be the tableau algorithm presented in Section 4.2. We now show how to combine the use of Hitting Set Trees and $SINGLE-MUPS(C,\mathcal{K})$ to compute $MUPS(C,\mathcal{K})$. Figure 6.2 illustrates the whole process for our example. We anticipate that the expected outcome is the following:

$$MUPS(C,\mathcal{K}_2) = \{\{1,2,3\},\{1,5\},\{2,3,4\},\{4,7\},\{3,5,6\},\{2,7\}\}.$$

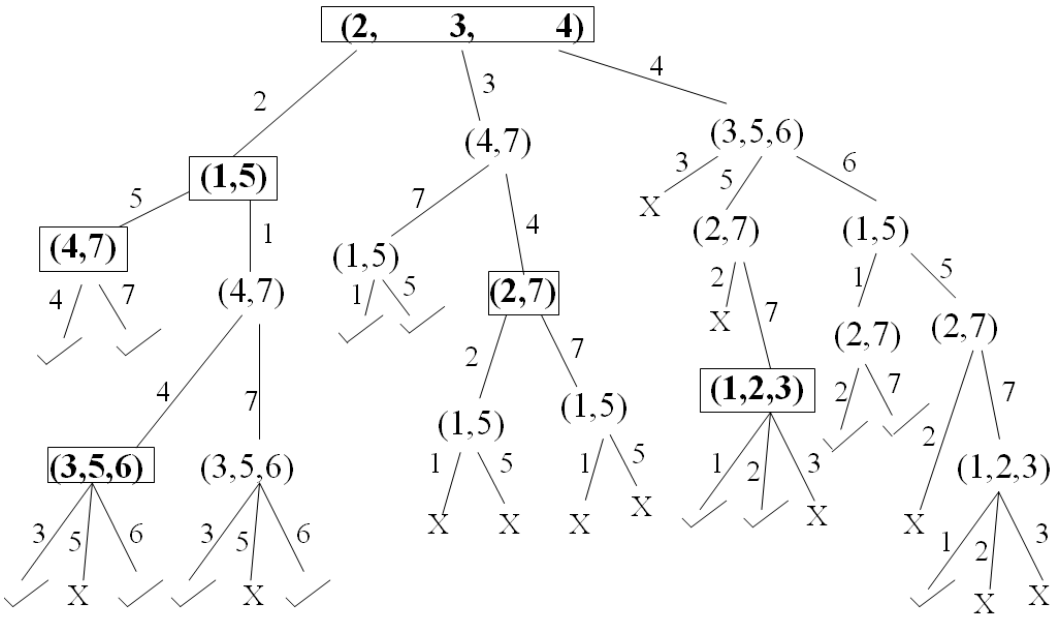

Figure 4.2: **Finding all MUPS using HST:** Each distinct node is outlined in a box and represents a set in $MUPS(C,\mathcal{K}_2)$. Total number of satisfiability tests is 31.

The algorithm starts by executing SINGLE_MUPS(C, $\mathcal{K}_2$) and let us assume that we obtain the set $S = \{2,3,4\}$ as an output. The next step is to initialize a Hitting Set Tree $\mathbf{T} = (V,E,\mathcal{L})$ with S in the label of its root, i.e. $V = \{v_0\}, E = \emptyset, \mathcal{L}(v_0) = S$. Then, it selects an arbitrary axiom in S, say 2, generates a new node w with an empty label in the tree and a new edge $\langle v_0, w\rangle$ with axiom 2 in its label. Finally, the algorithm tests the satisfiability of C w.r.t. $\mathcal{K}_2 - \{2\}$. If it is unsatisfiable, as in our case, we obtain a MUPS for C w.r.t. $\mathcal{K}_2 - \{2\}$, say $\{1,5\}$. We add this set to S and also insert it in the label of the new node w.

The algorithm repeats this process, namely removing an axiom, adding a node, checking satisfiability and executing the SINGLE_MUPS algorithm until the satisfiability test turns positive, in which case we mark the new node with a checkmark '$\surd$'.

The algorithm also eliminates extraneous satisfiability tests based on previous results, e.g., once a hitting set path is found, any superset of that path is guaranteed to be a hitting set as well, and thus no additional satisfiability tests are needed for that path, as indicated by a 'X' in the label of the node. For example, in Figure 6.2, the first path in the right-most branch of the root node is 4,3 and is terminated early since the algorithm has already considered all possible paths (hitting sets) containing axioms {3,4} in an earlier branch. Both '$\surd$' and 'X' labeled nodes constitute leaf nodes of $\mathbf{T}$.

When the HST is fully built, the distinct nodes of the tree collectively represent the complete set of MUPS of the unsatisfiable concept.

The correctness of this approach relies on the following key observations:

1. If a node is not a leaf of $\mathbf{T}$, then its label is an element of $MUPS(C, \mathcal{K})$
2. If one takes the union of the labels of the edges in any path from the root of $\mathbf{T}$ to a leaf node marked with a $\surd$, then a Hitting Set for $MUPS(C, \mathcal{K})$ w.r.t. $\mathcal{K}$ is obtained. In fact, all the the minimal Hitting Sets for $MUPS(C, \mathcal{K})$ w.r.t. $\mathcal{K}$ are obtained when all the paths from the root to a leaf in $\mathbf{T}$ are considered.

In what follows, we provide a formal specification of the algorithm and show that the above observations do hold in general.

4.3.4 Definition of the Algorithm

Algorithm: MUPS_HST(C, $\mathcal{K}$)
Input: C, $\mathcal{K}$, S, HS, w, α, p (default: all empty)
Output: S
if there exists a set $h \in HS$ s.t. $(\mathcal{L}(p) \cup \{\alpha\}) \subseteq h$, **then**
$\mathcal{L}(w) \leftarrow$ 'X'
return
else if C is unsatisfiable w.r.t. $\mathcal{K}$, **then**
$m \leftarrow$ SINGLE_MUPS$(C, \mathcal{K})$
$S \leftarrow S \cup m$
create a new node w' and set $\mathcal{L}(w') \leftarrow m$
if $w \neq null$, **then**
create an edge $e = \langle w, w' \rangle$ with $\mathcal{L}(e) \leftarrow \alpha$
$p \leftarrow p \cup e$
for each axiom $\beta \in \mathcal{L}(w')$ **do**
MUPS_HST(A, ($\mathcal{K} - \{\beta\}$), S, HS, w', β, p)
else
$\mathcal{L}(w) \leftarrow$ '$\surd$'
$HS \leftarrow HS \cup \mathcal{L}(p)$

Table 4.3: Finding all MUPS using Reiter's HST

The MUPS_HST algorithm is a recursive procedure that accepts as input a set S of conflict sets (initially containing a single MUPS), a set HS of Hitting Sets, the last node w added to the Hitting Set Tree, the last axiom α removed from $\mathcal{K}$ and the current edge path p. Initially, the Hitting Set Tree is empty.

The procedure incrementally builds a Hitting Set Tree while the input concept C is unsatisfiable w.r.t $\mathcal{K}$. The procedure works intuitively as sketched in the example of Section 4.3.3; the interested reader should find little difficulty in going through the algorithm using the example [4]

The correctness and completeness of this approach can be derived as a consequence of the above results and of Theorem 3 in Section 4.3.

[4] As a notation remark, we denote by $\mathcal{L}(p)$, for p a path in the tree, the union of the labels in all the edges in p.

Theorem 4 *(Correctness and Completeness)*
Let C be unsatisfiable w.r.t. $\mathcal{K}$, then:

MUPS_HST(C,$\mathcal{K}$) = MUPS(C,$\mathcal{K}$)

Proof

($\subseteq$)
Let $S \in$ MUPS_HST(C, $\mathcal{K}$), then S belongs to the label of some non-leaf node w in the Hitting Set Tree $\mathbf{T}$ generated by the algorithm. In this case, $\mathcal{L}(w) \in MUPS(C, \mathcal{K}')$, for some $\mathcal{K}' \subseteq \mathcal{K}$. Therefore, $S \in \mathsf{MUPS}(\mathsf{C}, \mathcal{K})$.

($\supseteq$)
We prove by contradiction. Suppose there exists a set $M \in MUPS(C, \mathcal{K})$, but $M \notin$ MUPS_HST(C, $\mathcal{K}$). In this case, M does not coincide with the label of any node in $\mathbf{T}$. Let v_0 be the root of $\mathbf{T}$, with $\mathcal{L}(v_o) = \{\alpha_1, ..., \alpha_n\}$. As a direct consequence of the completeness of Reiter's search strategy, the algorithm generates all the minimal Hitting Sets containing α_i for each $i \in \{1, .., n\}$. By Theorem 3, every minimal hitting set U is s.t. $U \cap M \neq \emptyset$. This implies that $\alpha_i \in M$ for $1 \leq i \leq n$. Therefore, $M \subseteq \mathcal{L}(v_0)$, which implies that $M \notin MUPS(C, \mathcal{K})$, since $\mathcal{L}(v_0) \in MUPS(C, \mathcal{K})$ and $M \subseteq \mathcal{L}(v_0)$.

□

The worst case of the algorithm arises when all the sets in $MUPS(C, \mathcal{K})$ are mutually disjoint. In this case, if there are n disjoint $MUPS(C, \mathcal{K})$ each of size k, the number of calls to SINGLE_MUPS (i.e., satisfiability tests involved) is k^n.

4.3.5 HST Optimization

In addition to the optimizations that Reiter's HST algorithm provides such as early path termination, there is one definite area of improvement, namely, storing the completion graph generated by the tableau algorithm at every node of the tree and *incrementally modifying* the graph for every change made (axiom removed). This saves the time required in building the graphs from scratch for each new node of the HST.

In order to incrementally modify the graph, we make use of the tableau tracing idea seen in Section 4.2.2 as follows: we first extend the set of tableau events to include operations for the removal of nodes/edges and their labels in the completion graph. Secondly, we extend the tracing function to capture a set of axiom sets instead of a single axiom set responsible for the event. Finally, when removing an axiom from the ontology, we remove only those portions of the graph that have been necessarily 'introduced' by the axiom, i.e., whose trace includes the concerned axiom (in every set in the trace). We then re-apply the tableau expansion rules to the current graph. The work is still in progress [42], [43].

4.4 Beyond Axioms: Finer-Grained Justifications

As noted earlier, a main drawback of the justifications is that they work at the asserted axiom level, and hence fail to determine which *parts* of the asserted axioms are *irrelevant* for the particular entailment under consideration to hold. For instance, given an axiom $A \sqsubseteq B \sqcap \neg B \sqcap \exists R.E \sqcap D$ where A is unsatisfiable, the conjuncts $\exists R.E$ and D

are irrelevant for the unsatisfiability of A. Moreover, additional parts of axioms that could contribute to the entailment are *masked*, e.g., if we were to add the axiom $A \sqsubseteq \forall R.\neg E$ to the earlier one, there exists an additional condition which makes A unsatisfiable, namely, the finer-grained axioms $A \sqsubseteq \exists R.E$ and $A \sqsubseteq \forall R.\neg E$, and this cannot be captured by the current definition of MUPS or justifications.

In this section, we discuss an extension to the Axiom Pinpointing service that captures *precise* justifications, which are at a finer granularity level than the original asserted axioms. In this context, we provide a formal notion of precise justification and propose a decision procedure for the problem.

4.4.1 Splitting a KB

Since we aim at identifying relevant parts of axioms, we define a function that *splits* the axioms in a KB $\mathcal{K}$ into "smaller" axioms to obtain an equivalent KB $\mathcal{K}_s$ that contains as many axioms as possible.

The idea of the transformation is to rewrite the axioms in $\mathcal{K}$ in a convenient normal form and split across conjunctions in the normalized version, e.g., rewriting $A \sqsubseteq C \sqcap D$ as $A \sqsubseteq C, A \sqsubseteq D$. In some cases, we are forced to introduce new concept names, only for the purpose of splitting axioms into smaller sizes (which prevents any arbitrary introduction of new concepts); for example, since the axiom $A \sqsubseteq \exists R.(C \sqcap D)$ is not equivalent to $A \sqsubseteq \exists R.C$, $A \sqsubseteq \exists R.D$, we introduce a new concept name, say E, and transform the original axiom into the following set of "smaller" axioms: $A \sqsubseteq \exists R.E$, $E \sqsubseteq C$, $E \sqsubseteq D$, $C \sqcap D \sqsubseteq E$.

We now provide a definition of splitting.

Definition 8 *Given a concept C in negation normal form (NNF), the set* $\mathsf{split(C)}$ *is inductively defined as follows:*

- *If $C \in \mathsf{Sig}(\mathcal{K})$ (the signature of $\mathcal{K}$, i.e. the set of names used in $\mathcal{K}$), C is of the form $\neg A$ for $A \in \mathsf{Sig}(\mathcal{K})$ or C of the form $\geq nR$ or $\leq nR$, then* $\mathsf{split(C)} = \{\mathsf{C}\}$.
- *If C is of the form $C_1 \sqcap C_2$, then* $\mathsf{split(C)} = \mathsf{split(C_1)} \cup \mathsf{split(C_2)}$.
- *If C is of the form $C_1 \sqcup C_2$, then* $\mathsf{split(C)} = \bigcup_{\mathsf{C_1' \in split(C_1), C_2' \in split(C_2)}} \mathsf{C_1' \sqcup C_2'}$
- *If C of the form $\forall R.D$, then* $\mathsf{split(C)} = \bigcup_{\mathsf{D' \in split(D)}} \forall \mathsf{R.D'}$.
- *If C of the form $\exists R.D$, then:*
  - *if D of the form $D_1 \sqcap D_2$, then* $\mathsf{split(C)} = \{\exists \mathsf{R.E}, \mathsf{E}\} \cup \mathsf{split}(\neg \mathsf{E} \sqcup \mathsf{D_1}) \cup \mathsf{split}(\neg \mathsf{E} \sqcup \mathsf{D_2}) \cup \mathsf{split}(\neg \mathsf{D_1} \sqcup \neg \mathsf{D_2} \sqcup \mathsf{E})\}$, *with E a new name.*
  - *otherwise,* $\mathsf{split(C)} = \bigcup_{\mathsf{D' \in split(D)}} \exists \mathsf{R.D'}$.

For a set of GCIs $\mathcal{K} = \bigcup_i \alpha_i$ with α_i a of the form $C_i \sqsubseteq D_i$, we have:

$$\mathsf{split}(\mathcal{K}) = \bigcup_{\mathsf{i}} \top \sqsubseteq \sqcap(\mathsf{split}(\neg \mathsf{C_i} \sqcup \mathsf{D_i}))$$

The splitting transformation is *conservative*, i.e., if $\mathcal{K}' = \mathsf{split}(\mathcal{K})$ every model of $\mathcal{K}'$ is also a model of $\mathcal{K}$, and every model of $\mathcal{K}$ can be extended to a model of $\mathcal{K}'$ by appropriately choosing the interpretation of the additional concept names.

Proposition 1 *Given ontologies $\mathcal{K}, \mathcal{K}'$, with $\mathcal{K}' = \mathsf{split}(\mathcal{K})$, we have that $\mathcal{K}'$ is a conservative extension of $\mathcal{K}$.*

Proof

The fact that every model $\mathcal{I}$ of $\mathcal{K}$ can be extended to a model of $\mathcal{K}'$ is trivial. We show the other direction, namely that if $\mathcal{I} \models \mathcal{K}'$, then $\mathcal{I} \models \mathcal{K}$. This is a direct consequence of the following claim:

$$C \in \mathsf{sub}(\mathcal{K}) \; implies \; C^{\mathcal{I}} = (\sqcap \mathsf{split}(\mathsf{C}))^{\mathcal{I}} \tag{4.1}$$

We prove this claim by induction on the structure of C. The base of the induction is given by the first bullet in Definition 8 and is straightforward to verify. We proceed with the induction step:

- If C is of the form $C_1 \sqcap C_2$, then $\mathsf{split}(\mathsf{C}) = \mathsf{split}(\mathsf{C_1}) \cup \mathsf{split}(\mathsf{C_2})$ and $C^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$. By induction, $C_i^{\mathcal{I}} = (\sqcap \mathsf{split}(\mathsf{C_i}))^{\mathcal{I}}$ and thus $C^{\mathcal{I}} = (\sqcap \mathsf{split}(\mathsf{C_1}))^{\mathcal{I}} \sqcap (\sqcap \mathsf{split}(\mathsf{C_2}))^{\mathcal{I}}$, and the hypothesis holds.
- If C is of the form $C_1 \sqcup C_2$, then $\mathsf{split}(\mathsf{C}) = \bigcup_{\mathsf{C_1'} \in \mathsf{split}(\mathsf{C_1}), \mathsf{C_2'} \in \mathsf{split}(\mathsf{C_2})} \mathsf{C_1'} \sqcup \mathsf{C_2'}$ and $C^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$. By induction, $C_i^{\mathcal{I}} = (\sqcap \mathsf{split}(\mathsf{C_i}))^{\mathcal{I}}$ and thus $C^{\mathcal{I}} = (\sqcap \mathsf{split}(\mathsf{C_1}))^{\mathcal{I}} \cup (\sqcap \mathsf{split}(\mathsf{C_2}))^{\mathcal{I}}$, and the hypothesis holds.
- If C is of the form $\forall R.D$ then $\mathsf{split}(\mathsf{C}) = \bigcup_{\mathsf{D'} \in \mathsf{split}(\mathsf{D})} \forall \mathsf{R}.\mathsf{D'}$ and $C^{\mathcal{I}} = \{a \in \Delta | \forall b \in \Delta$, if $(a,b) \in R^{\mathcal{I}}$, then $b \in D^{\mathcal{I}}\}$ (where Δ is the domain of interpretation). By induction, $D^{\mathcal{I}} = (\sqcap \mathsf{split}(\mathsf{D}))^{\mathcal{I}}$ and thus $C^{\mathcal{I}} = \{a \in \Delta | \forall b \in \Delta$, if $(a,b) \in R^{\mathcal{I}}$, then $b \in \sqcap \mathsf{split}(\mathsf{D})^{\mathcal{I}}\}$, which implies $C^{\mathcal{I}} = \bigcup_{D'^{\mathcal{I}} \in \mathsf{split}(\mathsf{D})^{\mathcal{I}}} \{a \in \Delta | \forall b \in \Delta$, if $(a,b) \in R^{\mathcal{I}}$, then $b \in D'^{\mathcal{I}}\}$.
- If C is of the form $\exists R.D$, then $C^{\mathcal{I}} = \{a \in \Delta | \exists b \in \Delta$ with $(a,b) \in R^{\mathcal{I}}$ and $b \in D^{\mathcal{I}}\}$ (where Δ is the domain of interpretation).
  - if D is not of the form $(D_1 \sqcap D_2)$, then $\mathsf{split}(\mathsf{C}) = \bigcup_{\mathsf{D'} \in \mathsf{split}(\mathsf{D})} \exists \mathsf{R}.\mathsf{D'}$. By induction, $D^{\mathcal{I}} = (\sqcap \mathsf{split}(\mathsf{D}))^{\mathcal{I}}$ and thus $C^{\mathcal{I}} = \{a \in \Delta | \exists b \in \Delta$, s.t. $(a,b) \in R^{\mathcal{I}}$ and $b \in \sqcap \mathsf{split}(\mathsf{D})^{\mathcal{I}}\}$, which implies $C^{\mathcal{I}} = \bigcup_{D'^{\mathcal{I}} \in \mathsf{split}(\mathsf{D})^{\mathcal{I}}} \{a \in \Delta | \exists b \in \Delta$ with $(a,b) \in R^{\mathcal{I}}$ and $b \in D'^{\mathcal{I}}\}$.
  - if D is of the form $(D_1 \sqcap D_2)$ then $\mathsf{split}(\mathsf{C}) = \{\exists \mathsf{R}.\mathsf{E}, \mathsf{E}\} \cup \mathsf{split}(\neg \mathsf{E} \sqcup \mathsf{D_1}) \cup \mathsf{split}(\neg \mathsf{E} \sqcup \mathsf{D_2}) \cup \mathsf{split}(\neg \mathsf{D_1} \sqcup \mathsf{D_2})$, where E is a new concept. Now, $(\exists R.(D_1 \sqcap D_2))^{\mathcal{I}} = (\exists R.E \sqcap (\neg E \sqcup (D_1 \sqcap D_2)))^{\mathcal{I}}$ and thus the hypothesis holds.

□

This ensures that every entailment in $\mathcal{K}$ holds in its splitting $\mathcal{K}'$, and every entailment in $\mathcal{K}'$ concerning only symbols in the signature of $\mathcal{K}$ holds in $\mathcal{K}$ as well.

Table 4.4 shows an algorithm to split a KB based on the above definition. In this algorithm, we also keep track of the correspondence between the new axioms and the axioms in $\mathcal{K}$ by using a function σ.

Proposition 2 *Given $\mathcal{K}$, the algorithm in Table 4.4 computes $\mathsf{split}(\mathcal{K})$ in linear time.*

Proof For each subclass axiom $\alpha \in \mathcal{K}$, the algorithm generates the concept C_α corresponding to α, normalizes the concept into NNF, and generates new axioms in $\mathsf{split}(\mathcal{K})$, $\mathcal{K}'$, based on the occurrence

Algorithm: Split KB
Input: KB $\mathcal{K}$
Output: TBox $\mathcal{K}'$, Axiom Correspondence Function σ

$\mathcal{K}' \leftarrow \emptyset$
initialize axiom correspondence function σ
initialize substitution $cache$
for each subclass axiom $\alpha \in \mathcal{K}$
    from α := $C \sqsubseteq D$ generate $C_\alpha := \neg C \sqcup D$
    normalize C_α to NNF (pushing negation inwards)
    $\mathcal{K}' \leftarrow \mathcal{K}' \cup \{\top \sqsubseteq C_\alpha\}$
    $\sigma(\{\top \sqsubseteq C_\alpha\}) \leftarrow \alpha$
while there exists $\{\top \sqsubseteq C_\alpha\} \in \mathcal{K}'$ with $A \sqcap B$ occurring at position π in C_α
    $\mathcal{K}' \leftarrow \mathcal{K}' - \{\top \sqsubseteq C_\alpha\}$
    if $A \sqcap B$ is not qualified by an existential restriction, **then**
        $C_A \leftarrow C_\alpha[A]_\pi$; $\sigma(C_A) \leftarrow \sigma(C_A) \cup \sigma(C_\alpha)$; $\mathcal{K}' \leftarrow \mathcal{K}' \cup \{\top \sqsubseteq C_A\}$
        $C_B \leftarrow C_\alpha[B]_\pi$; $\sigma(C_B) \leftarrow \sigma(C_B) \cup \sigma(C_\alpha)$; $\mathcal{K}' \leftarrow \mathcal{K}' \cup \{\top \sqsubseteq C_B\}$
    else
        if $cache(A \sqcap B) = \emptyset$, **then**
            let E be a new concept not defined in $\mathcal{K}'$
            $\mathcal{K}' \leftarrow \mathcal{K}' \cup \{E \sqsubseteq A, E \sqsubseteq B, A \sqcap B \sqsubseteq E\}$
            $cache(A \sqcap B) \leftarrow E$
        else $E \leftarrow cache(A \sqcap B)$
        $C_E \leftarrow C_\alpha[E]_\pi$; $\sigma(C_E) \leftarrow \sigma(C_\alpha)$; $\mathcal{K}' \leftarrow \mathcal{K}' \cup \{\top \sqsubseteq C_E\}$

Table 4.4: Splitting a KB

of a conjunction in the concept C_α. For each conjunction that is not qualified by some existential role restriction $\exists R$, the algorithm generates two new axioms (obtained by substituting the conjunction by each of its conjuncts), whereas for a conjunction qualified by some $\exists R$, the algorithm generates four new axioms (obtained by introducing a new concept as seen in Definition 8). Thus, for each axiom, the algorithm adds new axioms based on some constant times the number of conjunctions. Since the total number of conjunctions is fixed and each conjunction is split only once, the algorithm takes linear time to compute the result. Also, the size of $\mathcal{K}'$ increases linearly in the size of $\mathcal{K}$. □

Finer-grained justifications can be defined as follows:

Definition 9 ***(Precise Justification)***
*Let $\mathcal{K} \models \alpha$. A KB $\mathcal{K}'$ is a **precise justification** for α in $\mathcal{K}$ if $\mathcal{K}' \in \mathsf{JUST}(\alpha, \mathsf{split\_KB}(\mathcal{K}))$.*

We denote by $\mathsf{JUST}_\mathsf{p}(\alpha, \mathcal{K})$ the set of all precise justifications for α in $\mathcal{K}$.

4.4.2 Finding Precise Justifications

The problem of finding all precise justifications for an entailment α of a KB $\mathcal{K}$ now reduces to the problem of finding all justifications for α in the $split\_KB(\mathcal{K})$. Thus, we can use the algorithm listed in Table 4.4 to split a KB, and then apply any decision procedure to find all justifications for the entailment in the split version of the KB, such as the one described earlier in the chapter.

4.4.3 Optional Post-Processing

Sometimes, from a user perspective, it may be more desirable to provide an explanation for the entailment only in terms of (parts of) the original asserted axioms, and suppress any new concept names introduced during the splitting process. In such cases, we can use the axiom correspondence function σ generated in Table 4.4 to replace the newly introduced terms by their original counterparts using the algorithm shown in Table 4.5.

Algorithm: Remove New Terms
Input: Collection of Axiom Sets $\mathcal{J}$, Original KB $\mathcal{K}$
Output: $\mathcal{J}$
for each axiom set $j \in J$ **do**,
while there exists a term $C' \in \mathsf{Sig(j)}$ s.t. $C' \notin \mathsf{Sig}(\mathcal{K})$, **do**
$S \leftarrow \emptyset$
for each axiom $C' \sqsubseteq C_i \in j$, **do**
$S \leftarrow S \cup C_i$
if $S \neq \emptyset$, **then**
$C \leftarrow C_1 \sqcap C_2 \sqcap ..C_n$ (for all $C_i \in S$)
else
$C \leftarrow \top$
substitute C' with C in j

Table 4.5: Post-Processing to Remove New Concept Names

4.4.4 Example

We now present a detailed example to demonstrate how the algorithm finds precise justifications.

Consider a KB $\mathcal{K}$ composed of the following axioms:

1. $A \sqcup B \sqsubseteq \exists R.(C \sqcap \neg C) \sqcap D \sqcap E$
2. $A \sqsubseteq \neg D \sqcap B \sqcap F \sqcap D \sqcap \forall R.\bot$
3. $E \sqsubseteq \forall R.(\neg C \sqcap G)$

Note, the signature of the KB $\mathsf{Sig}(\mathcal{K}) = \{A, B, C, D, E, F, R\}$, and the concept A is unsatisfiable w.r.t $\mathcal{K}$.

Given $A, \mathcal{K}$ as input, the algorithm proceeds as follows:

Step 1: First, we obtain an equivalent KB $\mathcal{K}_s$ that is split as much as possible using the procedure explained earlier:

$T_s = \{A \sqsubseteq \exists R.H^1; B \sqsubseteq \exists R.H^1; A \sqsubseteq D^{1,2}; B \sqsubseteq D^1; A \sqsubseteq E^1; B \sqsubseteq E^1; H \sqsubseteq C^1; H \sqsubseteq \neg C^1; A \sqsubseteq \neg D^2; A \sqsubseteq B^2; A \sqsubseteq F^2; A \sqsubseteq \forall R.\bot^2; E \sqsubseteq \forall R.\neg C^3; E \sqsubseteq \forall R.G^3\}$.

The superscript of each axiom in $\mathcal{K}_s$ denotes the corresponding axiom in $\mathcal{K}$ that it is obtained from. This correspondence is captured by the function σ in the Split-KB algorithm (see Table 4.4). Notice that the superscript of the axiom $A \sqsubseteq D$ in $\mathcal{K}_s$ is the set $\{1, 2\}$ since it can be obtained from two separate axioms in $\mathcal{K}$. Also, we have introduced a

new concept H in the split KB, which is used to split the concept $\exists R.(C \sqcap \neg C)$ in axiom 1.

Step 2: Now, we obtain the justifications for the unsatisfiability of A w.r.t $\mathcal{K}_s$. This gives us the following axiom sets J:

$J = \{\{A \sqsubseteq \exists R.H^1, H \sqsubseteq C^1, H \sqsubseteq \neg C^1\}; \{A \sqsubseteq D^{1,2}, A \sqsubseteq \neg D^2\}; \{A \sqsubseteq \exists R.H^1, H \sqsubseteq C^1, A \sqsubseteq E^1, E \sqsubseteq \forall R.\neg C^3\}; \{A \sqsubseteq \exists R.H^1, A \sqsubseteq \forall R.\bot^2\}\}$

J is the complete set of precise justifications for $A \equiv \bot$ in $\mathcal{K}$.

Step 3: Optionally, we can remove the concept H introduced in $\mathcal{K}_s$ from the justification sets in J to get:

$\mathcal{K}' = \{\{A \sqsubseteq \exists R.(C \sqcap \neg C)^1\}; \{A \sqsubseteq D^1, A \sqsubseteq \neg D^2\}; \{A \sqsubseteq D^2, A \sqsubseteq \neg D^2\}; \{A \sqsubseteq \exists R.C^1, A \sqsubseteq E^1, E \sqsubseteq \forall R.\neg C^3\}; \{A \sqsubseteq \exists R.\top^1, A \sqsubseteq \forall R.\bot^2\}\}$

4.4.5 Optimizations

The additional overhead incurred for capturing precise justifications is due to the splitting of the entire KB beforehand. The main concern, from a reasoning point of view, is the introduction of GCIs during the splitting process, e.g. $A \equiv B \sqcap C$ is replaced by (among other things) $B \sqcap C \sqsubseteq A$. Even though these GCIs are absorbed, they still manifest as disjunctions and hence adversely affect the tableau reasoning process.

Alternately, a more optimal version of the algorithm is the following: instead of splitting the entire KB beforehand, we can perform a *lazy splitting* of certain specific axioms (on the fly) in order to improve its performance. The modified algorithm with lazy splitting becomes:

- Given A unsatisfiable w.r.t $\mathcal{K}$, find a single justification set, $J \in \mathsf{JUST}(\mathsf{A} \equiv \bot, \mathcal{K})$
- Split axioms in J to give J_s. Prune J_s using the Black-box algorithm seen in Section 4.2.1 to arrive at a minimal precise justification set J_p
- Replace J by J_s in $\mathcal{K}$.
- Form Reiter's HST using J_p as a node, with each outgoing edge being an axiom $\alpha \in J_p$ that is removed from $\mathcal{K}$

The advantage of this approach is that it only splits axioms in the intermediate justification sets in order to arrive at precise justifications, and re-inserts split axioms back into the KB dynamically.

Finally, we mention one other optimization (heuristic) that can be used to easily identify and remove irrelevant parts of axioms in the justification set, even before we perform any splitting operation. The idea is the following: given any one justification J for a particular entailment α, let J_α be the set of axioms in the justification plus the axiom denoting the entailment itself. Now, if we consider the set of symbols appearing in the signature of $\mathcal{J}_\alpha$, then symbols that appear only once in any of the axioms in J_α can be considered irrelevant for the entailment.

For example, suppose the following three axioms constitute the justification J for the entailment $\alpha : C \sqsubseteq D$ in some ontology (where $A - G$ are atomic concepts and R is an atomic role):

1. $C \sqsubseteq A \sqcap \exists R.E \sqcap \forall R.F$
2. $A \sqsubseteq B \sqcap G$
3. $G \sqsubseteq D$

In the above case, $J_\alpha = \{1, 2, 3\} \cup \{C \sqsubseteq D\}$. Now, the concepts B, E, F appear only once in J_α and hence can be considered irrelevant for the entailment. Similarly, given that F is irrelevant, the expression $\forall R.F$ can be considered irrelevant as well. Thus, we only need to split $\{C \sqsubseteq A \sqcap \exists R.\top\ A \sqsubseteq G, G \sqsubseteq D\}$.

4.5 Applications of Axiom Pinpointing

Obviously, the main use of the Axiom Pinpointing service is for explaining the output of the description logic reasoner to the user – it can be used to extract and display the minimal set of axioms in the KB responsible for a particular entailment. This also implies that removing (or possibly rewriting) any one of the axioms in each of the justification sets will drop the entailment from the KB. This is especially useful in the context of debugging, where the goal is to get rid of the unsatisfiability entailment or the KB inconsistency itself.

In the case of precise justifications, the service displays minimal set of axioms in a more fine-grained, but equivalent version of the KB, which helps in focusing on only the relevant parts of the original axioms responsible for the entailment. Even in this case, removing axioms in the precise justification sets will drop the concerned entailment. Though, the advantage in the latter case (precise justifications) is that less additional entailments are lost compared to the former case (justifications), where entire axioms are removed.

Chapter 5
Auxillary Debugging Service: Root Error Pinpointing
5.1 Introduction

The core debugging service developed so far, Axiom Pinpointing, can be used to understand and resolve a particular semantic defect, e.g., an unsatisfiable class, since it provides the precise set of axioms responsible for it. However, consider what happens when dealing with an ontology that has a large number of unsatisfiable classes, e.g., the original OWL version[1] of the Tambis ontology in which 144 out of 395 classes are unsatisfiable. In this case, the user can adopt a brute force approach and iterate through the list of unsatisfiable classes, fixing each one in turn by invoking the axiom pinpointing service separately for every defect. Besides being pointlessly exhausting, there are two serious problems here. Firstly, many of the unsatisfiable classes depend in simple ways on other unsatisfiable classes, e.g., the class protein is defined as a subclass of the unsatisfiable class macromolecular_compound, and the class protein_part is related to protein by forcing an existential restriction on the property part_of. In such cases, a brute approach may not necessarily produce correct results, e.g., the user could remove the subsumption protein $\sqsubseteq$ macromolecular_compound instead of resolving the source of the problem which lies in the unsatisfiable class macromolecular_compound. Secondly, there are large, far-reaching effects of assertions in a logic like OWL, e.g., in one case, three changes in Tambis repair over seventy other unsatisfiable classes. Thus, it is not sufficient to take on defects in isolation.

In this chapter, we design a service that given an ontology with numerous defects, detects dependencies between them and identifies the *source* of the problems.

We first consider each of the semantic defects, i.e., unsatisfiable classes and inconsistent ontologies, separately. For the former, we categorize unsatisfiable classes into two types, *root* (or critical) and *derived* (or dependent), and propose a set of algorithms to separate them. For the latter problem of inconsistent ontologies, we show techniques to reduce the problem to unsatisfiable classes where possible, or present alternate solutions to highlight the core inconsistency causing axioms. In both cases, we discuss the significance and drawbacks of the algorithms developed using appropriate examples.

Finally, in the last section, we pull together the algorithms described earlier in the chapter into a single coherent debugging service for *Root Error Pinpointing*.

5.2 Dealing with Numerous Unsatisfiable Classes

In this section, we consider the problem of debugging a consistent ontology that has a large number of unsatisfiable classes. Typically, ontology users or modelers are concerned about the unsatisfiability of the *atomic* or named classes in the ontology, since they represent key classes in the domain of the ontology.

[1] http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/tambis-full.owl

5.2.1 Root and Derived

We start by broadly categorizing unsatisfiable classes into two main types:

1. **Root Class** - this is an unsatisfiable atomic class in which a clash or contradiction found in the class definition (axioms) does not *depend on the unsatisfiability* of another atomic class in the ontology. More specifically, the unsatisfiability bug for a root class cannot be fixed by *simply* correcting the unsatisfiability bug in some other class, instead, it requires fixing some contradiction stemming from its own definition. Example of a root class is: $\mathsf{nonmetal} \sqsubseteq \geq 2.\mathsf{atomic\_number} \sqcap \leq 1.\mathsf{atomic\_number}$, given that this is the only definition of $\mathsf{nonmetal}$.

2. **Derived Class** - this is an unsatisfiable atomic class in which a clash or contradiction found in a class definition either directly (via explicit assertions) or indirectly (via inferences) *depends on the unsatisfiability* of another atomic class (we refer to it as the *Parent* dependency). Hence, this is a less critical bug in that resolving it involves fixing the unsatisfiability of the parent dependency. Example of a derived class is: $\mathsf{carbon} \sqsubseteq \mathsf{nonmetal}$, where $\mathsf{nonmetal}$ is an unsatisfiable class itself, in this case, its parent.

We give formal definitions for root and derived unsatisfiable classes in terms of the justification for their unsatisfiability, and also formalize the related notion of *parent* dependency for a derived class:

Definition 10 *(Root, Derived and Parent) Let $C_1, C_2, ...C_n$ be a set of unsatisfiable atomic classes in a consistent ontology $\mathcal{O}$. Let J_i be the justification for the unsatisfiability of the class C_i, i.e., J_i =JUSTIFY($C_i \equiv \bot$, $\mathcal{O}$). C_i is a **derived** unsatisfiable class iff there exists an axiom set $s_i \in J_i$ such that $s_i \supseteq s_j$, where $s_j \in J_j, (j \neq i)$. In this case, the class C_j is a **parent** dependency of C_i if there exists no axiom set $s_k \in J_k, (k \neq j, k \neq i)$ such that $s_k \supset s_j$. An unsatisfiable class that is not derived is a **root** unsatisfiable class.*

Intuitively, a derived unsatisfiable class C depends on parent D if the unsatisfiability of D in the ontology causes C to be unsatisfiable. A derived class can have more than one parent dependency, e.g., given the following axioms:

$$\boxed{A \sqsubseteq B \quad A \sqsubseteq \exists R.C \quad B \sqsubseteq D \sqcap \neg D \quad C \sqsubseteq E \sqcap \neg E}$$

the unsatisfiable class A has two parents, B and C.

Furthermore, if resolving the error in each of its parents turns a derived class satisfiable, we refer to it as *purely* derived, otherwise we refer to it as *partially* derived. In the case above, A is purely derived.

We capture this notion formally by extending the definition above to include the two types of derived classes:

Definition 11 *(Pure and Partially Derived) Let C_i be a derived unsatisfiable class. C_i is purely derived if for every axiom set $s_i \in J_i$ there exists a set s_j, $s_j \in J_j, (j \neq i)$ such that $s_i \supseteq s_j$, otherwise it is partially derived.*

Note that a partially derived unsatisfiable class has at least one standalone contradiction. This implies that if one were to adopt an iterative process to debugging, i.e., fix all the root unsatisfiable concepts in each iteration (as discussed in the next subsection), then the partially derived unsatisfiable classes would be exposed as roots in later iterations, and thus would need specific attention at that point. This is unlike purely derived unsatisfiable classes where one needs to focus on it's parent bugs alone.

5.2.2 Significance and Drawbacks of Root/Derived

The process of debugging an ontology that has numerous unsatisfiable classes can be performed from two different points of view – an *axiom-driven* view or a *class-driven* view. In the former approach, the user focuses on a set of erroneous axioms in the ontology that entail the unsatisfiability of *at least one atomic class*, and resolve the modeling error in the axioms to get rid of the unsatisfiability. This process can be repeated until all the unsatisfiable classes are fixed. In the latter approach, the user can focus on a particular unsatisfiable class, resolve the contradiction in its definition before proceeding to the next class, and repeat the process till all the classes are fixed. The difference is subtle since there is an obvious and strong correlation between classes and axioms in the ontology, i.e., the meaning of the class is specified by the axioms that define it. However, the choice of view is influenced by whether the ontology modeler cares more about *erroneous axioms* or *erroneous classes* per se. Another factor which dictates the view is the support provided by the debugging/editing tool or environment – typical ontology editors such as Protege [76], OntoEdit [98], Swoop [57] etc. provide a class-based view of the ontology instead of an axiom-centered view.

Obviously, a service that identifies root/derived unsatisfiable classes comes into play when the modeler adopts a class-driven view to debugging. The significance of the service is clear: the modeler needs to fix the root unsatisfiable classes first, which automatically reduces the problem causing conditions in the derived classes, possibly turning some of them satisfiable immediately. This gives rise to an iterative debugging process – in each iteration, the modeler focuses on the current roots, the resolution of which uncovers a new set of unsatisfiable classes containing new roots, that lead to the next iteration.

There is one other interesting aspect of root/derived classes that needs to be addressed, i.e., the possibility of a pair of unsatisfiable classes being *mutually dependent*, making them both derived.

For example, consider an ontology $\mathcal{O}_1$ with the following axioms: $\{A \sqsubseteq C \sqcap \neg C, B \sqsubseteq D \sqcap \neg D, A \equiv B\}$. In $\mathcal{O}_1$, the atomic classes A, B are unsatisfiable. Moreover, according to Definition 10, both A, B are classified as derived classes, with the parent dependency of one being the other, due to the equivalence relation between them. Thus, in this case, the ontology has only derived unsatisfiable classes with no roots. Here, emphasizing the error dependence between unsatisfiable classes can help understand the reason for this result and point the modeler to the appropriate classes to be fixed.

5.2.3 Detecting Root/Derived: Using the Axiom Pinpointing Service

We now present a straightforward approach to finding the root/derived unsatisfiable classes in an ontology. The idea behind this approach is to make use of the Axiom Pinpointing service (seen in Chapter 4) to determine the justification set for each unsatisfiable class and then use the property of justification containment, as seen in Definition 10, in order to determine error dependence and thereby separate the root from the derived unsatisfiable classes.

Given an ontology with unsatisfiable classes $C_1, ..C_n$, the algorithm generates an error-dependency graph $EDG = (V, E)$ where the vertices $V = \{v_1, ..v_n\}$ denote unsatisfiable classes, i.e., $\mathcal{L}(v_i) = C_i$, and a directed edge $e_{(i,j)}$ from vertex v_i to v_j denotes that C_j is the parent dependency of the class C_i. A vertex of the graph without any outgoing edges represents a root unsatisfiable class, whereas a vertex with least one outgoing edge represents a derived unsatisfiable class.

Algorithm: *Generate_DependencyGraph* *Input*: Ontology $\mathcal{O}$, Classes $\{C_1, ..C_n\}$ *Output*: Error Dependency Graph EDG
$EDG \leftarrow (V, E)$ **for each** unsatisfiable class $C_i \in \{C_1..C_n\}$ $V \leftarrow V \cup v_i$ $\mathcal{L}(v_i) \leftarrow C_i$ **for each** $j_i \in$ JUSTIFY($C_i \equiv \bot$, $\mathcal{O}$) $parent \leftarrow \emptyset$ **for each** unsatisfiable class $C_k \in \{C_1..C_n\}, k \neq i$ **for each** $j_k \in$ JUSTIFY($C_k \equiv \bot$, $\mathcal{O}$) **if** $j_i \supseteq j_k$ **and** ($parent = \emptyset$ **or** $\forall_{p \in parent}\ j_k \not\subset j_p$), **then** $parent \leftarrow parent \cup k$ **for each** $p \in parent$ $E \leftarrow E \cup e_{(i \rightarrow p)}$

Table 5.1: Algorithm to Generate EDG

Algorithm Analysis and Discussion

The algorithm *Generate_DependencyGraph* creates an error-dependency-graph EDG given a consistent ontology O that has a set of unsatisfiable classes $C_1, ...C_n$. In the first stage of the algorithm, it cycles through the unsatisfiable classes in $\mathcal{O}$, adding each class to the label of a distinct node in the EDG, and obtaining the justification for the unsatisfiability entailment of the class. In the second stage of the algorithm, it adds directed edges in the graph by looping through the unsatisfiable classes, determining parent dependencies, if any, using the precomputed justification sets (based on Definition

Figure 5.1 shows a sample error-dependency-graph generated by the algorithm for an ontology $\mathcal{O}_2$ consisting of five axioms as shown. The classes A, B, C, D in the ontology are unsatisfiable. In this case, D is the only root unsatisfiable class, whereas B and C and mutually dependent.

To verify this result, consider the justifications for each unsatisfiability entailment: (Note: we use numbers to denote axioms)

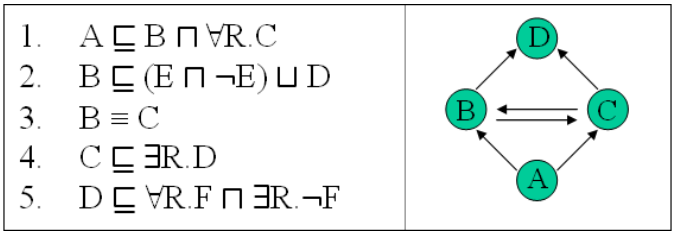

Figure 5.1: Sample Error Dependency Graph

- J_A = JUSTIFY($A \equiv \bot$, $\mathcal{O}_2$) $= \{\{1, 2, 5\}, \{1, 3, 4, 5\}\}$
- J_B = JUSTIFY($B \equiv \bot$, $\mathcal{O}_2$) $= \{\{2, 5\}, \{3, 4, 5\}\}$
- J_C = JUSTIFY($C \equiv \bot$, $\mathcal{O}_2$) $= \{\{4, 5\}, \{2, 3, 5\}\}$
- J_D = JUSTIFY($D \equiv \bot$, $\mathcal{O}_2$) $= \{\{5\}\}$

Based on Definition 10, A has parent dependencies B and C because the set $\{1, 2, 5\} \in J_A$ is a superset of $\{2, 5\} \in J_B$ and the set $\{1, 3, 4, 5\} \in J_A$ is a superset of $\{4, 5\} \in J_C$. Thus, there exist directed edges from the node A to the nodes B, C in the EDG. The remaining dependency relations are computed in a similar manner.

As shown above, the advantage of this algorithm is that it clearly distinguishes between the root, derived and mutually-dependent unsatisfiable classes by highlighting the dependencies between the various errors. Also, the correctness of the algorithm is evident given that it enforces the semantics of Definition 10.

The output of the algorithm can be enhanced in several ways in order to help the ontology debugger understand the relationships between the errors better. For example, we could

- *label edges with the number of dependencies between unsatisfiable classes*, i.e., if a derived unsatisfiable concept x has n of its justification sets subsumed by the justifications of its parent y, we could set $\mathcal{L}(x \rightarrow y) = n$.
- *differentiate between purely and partially derived unsatisfiable classes*, e.g., we could add a self-loop to nodes representing unsatisfiable concepts that have atleast one stand-alone justification set. This would include both, root and partially derived unsatisfiable classes.

However, the algorithm has some drawbacks. The main problem is that it requires computing the justification for every (atomic) unsatisfiability entailment, which as seen in Chapter 4, is an expensive process given its complexity (2NExpTime [103]). A secondary problem is that the algorithm does not highlight the *dependency axioms*, i.e., axioms which relate a derived unsatisfiable class to its parent. In the next subsection, we look at an alternate solution to determine the root/derived unsatisfiable classes that addresses both of these problems. The solution is sound, though incomplete, and hence is appropriate as a pre-processing optimization step before invoking the *Generate_DependencyGraph* algorithm.

5.2.4 Alternate Detection of Root/Derived: Structural Analysis

In this subsection, we present a dependency-detection algorithm that does not rely on the computation of justification for each unsatisfiability entailment, instead it analyzes the structure of the axioms in the ontology in order to ascertain the root and derived unsatisfiable classes. The structural analysis also helps identify the corresponding axioms that link a derived class to its parent dependency.

Given a consistent ontology $\mathcal{O}$ with a known set of unsatisfiable classes $\{C_1, ...C_n\}$, the algorithm returns an error dependency-graph, similar to the type discussed in the previous section, with the difference being that an edge from a derived unsatisfiable class C_d to it's parent dependency C_p is labeled with a set of dependency axioms linking C_d to C_p.

The algorithm consists of two phases: *asserted* dependency detection and *inferred* dependency detection and we describe each in detail.

Detecting Asserted Dependencies: Structural Tracing

This phase is used to detect dependencies between unsatisfiable classes by analyzing the asserted axioms in the ontology. Before we proceed to the description of the algorithm, we provide an example to illustrate the main intuitions.

Consider an ontology $\mathcal{O}_3$ with the following axioms:

1. $A \sqsubseteq \forall R.C \sqcap B \sqcap \exists P.D$	2. $A \sqsubseteq \geq 1.R$	3. $B \sqsubseteq (D \sqcap \neg D) \sqcap (C \sqcup \forall R.E)$
4. $C \sqsubseteq E \sqcap \neg E$	5. $D \sqsubseteq F \sqcap \neg F$	

Table 5.2: Structural Tracing Example

In $\mathcal{O}_3$, the atomic classes A, B, C, D are unsatisfiable. Note that B, C, D are roots, whereas A has three different parents: B, D due to axiom $\{1\}$, and C due to axioms $\{1, 2\}$. Determining that B, D are parents of A is rather straightforward because of the direct relation in axiom 1, i.e., B is a superclass of A, and D is related to A by an existential role P. On the other hand, realizing that C is a parent of A requires correlating between the universal restriction on role R in axiom 1 and the cardinality restriction on the same role in axiom 2, which forces the existence of the role to the unsatisfiable concept C. Non-local effects such as these need to be taken into account when designing this algorithm.

We now present the basic cases of the tracing approach.

Given an ontology $\mathcal{O}$ in which class A is unsatisfiable. The class A is derived if it satisfies any of the conditions shown in Table 5.3.

These basic cases can be extended to identify more non-local dependencies. For example, in cases (4), (5), instead of a single role restriction leading to an unsatisfiable class, we can consider a role-chain, i.e., a chain of role successors that lead to an unsatisfiable class. Also, in cases (6), (7), we can make an additional check to see whether the domain (/range) of any *ancestor* role of R(/R^-) is unsatisfiable.

The pseudo code for this algorithm is shown in Table 5.4. It uses a recursive subroutine $Trace\_Concept$ to determine unsatisfiable parent dependencies in the RHS of each

1. $A \sqsubseteq B \in \mathcal{O}$ and class B is unsatisfiable
2. $A \sqsubseteq C_1 \sqcap C_2 ... \sqcap C_n \in \mathcal{O}$ and *any* class C_i ($1 \leq i \leq n$) is unsatisfiable
3. $A \sqsubseteq D_1 \sqcup D_2 ... \sqcup D_n \in \mathcal{O}$ and *all* D_i ($1 \leq i \leq n$) are unsatisfiable
4. $A \sqsubseteq \exists R.B \in \mathcal{O}$ and B is unsatisfiable
5. $A \sqsubseteq \forall R.B, A \sqsubseteq \geq$n.$R(or A \sqsubseteq \exists R.C) \in \mathcal{O}$ and B is unsatisfiable
6. $A \sqsubseteq \geq$n.$R(or A \sqsubseteq \exists R.C), domain(R) = B \in \mathcal{O}$ and B is unsatisfiable
7. $A \sqsubseteq \geq$n.$R(or A \sqsubseteq \exists R.C), range(R^-) = B \in \mathcal{O}$ and B is unsatisfiable

Table 5.3: Base Cases of Structural Tracing

class definition axiom, based on the basic cases and the two extensions listed above.

Algorithm Analysis and Discussion

By making calls to the subroutine $Trace\_Concept$ recursively, the algorithm is able to detect *nested* dependencies, e.g., given the axiom $A \sqsubseteq \exists R.(B \sqcup C) \sqcap D$ in an ontology $\mathcal{O}_4$, where A, B, C are unsatisfiable concepts, and D is satisfiable in $\mathcal{O}_4$, the algorithm correctly determines that B, C are *both* the parents of A, since their being unsatisfiable makes A unsatisfiable as well.

Also note that there is a point in the algorithm where it checks for a correlation between an existential and a universal restriction on the same role leading to an unsatisfiable concept (as mentioned in the example in Table 5.2). In this case, it is possible to use a pre-processing step as seen in [58], where we trace the concept definition using the same procedure ($Trace\_Concept$), and collect all the necessary information to check for beforehand.

One of the main advantages of the structural tracing algorithm is its complexity: given that all the definition axioms for an unsatisfiable concept can be laid out into a single concept description (by taking the conjunction), the complexity of the structural tracing is linear in the size of the description created, as each conjunct is examined only once sequentially in a deterministic manner. This is a definite improvement over the previous approach. For realistic KBs, we have found it's performance to be reasonably fast as shown in Chapter 7, e.g., in the case of the Tambis OWL ontology[2], which has 144 out of 395 unsatisfiable concepts, the algorithm identifies the 3 roots in under five seconds.

We now list a straightforward, yet important theorem related to structural tracing:

Theorem 5 *(Soundness)*

Let $\mathcal{O}$ be an ontology with unsatisfiable classes $C_1, ...C_n$. Let $EDG = (V, E)$ be the error dependency graph output by the Structural_Tracing algorithm when given inputs $\mathcal{O}, \{C_1, ...C_n\}$. Let v, C be a vertex (in V), unsatisfiable class (in $\mathcal{O}$) respectively such that $\mathcal{L}(v) = C$, and suppose there exists at least one outgoing edge $e_{v \rightarrow v'} \in E$ with $\mathcal{L}(e) = S_{ax}$ and $\mathcal{L}(v') = D$. Then C is a derived unsatisfiable class.

Proof We are given $\mathcal{L}(e) = S_{ax}$, where e is an edge from concept C to D in the EDG. Thus,

[2]http://protege.stanford.edu/plugins/owl/owl-library/tambis-full.owl

Algorithm: *Structural_Tracing*
Input: Ontology $\mathcal{O}$, Classes $\{C_1, ..C_n\}$
Output: Error Dependency Graph EDG

$EDG \leftarrow (V, E)$
for each unsatisfiable class $C_i \in \{C_1..C_n\}$
  $S_D \leftarrow$ set of concept definition axioms of C_i in $\mathcal{O}$
  for each axiom $ax \in S_D$, ($ax : C_i \sqsubseteq D$ or $C_i \equiv D$)
    role chain $rc \leftarrow \emptyset$
    $S_\tau \leftarrow$ Trace_Concept($D, \{ax\}$)
    for each tuple $\langle D, S_{ax} \rangle \in S_\tau$,
      $V \leftarrow V \cup \{v_1, v_2\}; E \leftarrow E \cup \{e_{(v_1 \rightarrow v_2)}\}$
      $\mathcal{L}(v_1) \leftarrow \{C_i\}; \mathcal{L}(v_2) \leftarrow \{D\}; \mathcal{L}(e) \leftarrow S_{ax}$

subroutine: Trace_Concept(Class C, Axiom Set S)
$S_\tau \leftarrow \emptyset$
if C is *atomic* **and** C is *unsatisfiable* **then**
  $S_\tau \leftarrow S_\tau \cup \{(C, S)\}$
else if C is of the form $c_1 \sqcap c_2 .. \sqcap c_n$, **then**
  for each conjunct $c_i \in C$,
    $S_\tau \leftarrow S_\tau \cup$ Trace_Concept(c_i, S)
else if C is of the form $d_1 \sqcup d_2 .. \sqcup d_n$, **then**
  $S_{all} \leftarrow \emptyset$
  for each disjunct $d_i \in C$,
    $S_{disj} \leftarrow Trace\_Concept(d_i, S)$
    if $S_{disj} = \emptyset$, **return** $\emptyset$
    else $S_{all} \leftarrow S_{all} \cup S_{disj}$
  $S_\tau \leftarrow S_\tau \cup S_{all}$
else if C is of the form $\exists R.D$ **or** $\geq n.R$ **or** $\exists R.\{I\}$, **then**
  $rc \leftarrow rc|R$
  if $C = \exists R.D$, $S_\tau \leftarrow S_\tau \cup$ Trace_Concept(D, S)
  for each role R' that is equivalent to R, or an ancestor-role of R
    $S_\tau \leftarrow S_\tau \cup$ Trace_Concept($domain(R'), S$)
    $S_\tau \leftarrow S_\tau \cup$ Trace_Concept($range(R'^{-}), S$)
else if C is of the form $\forall R.D$, **then**
  $rc \leftarrow rc|R$
  if there exists an axiom $\alpha \in O$ of the form $C_i \sqsubseteq \exists rc.E$, **then**
    $S_\tau \leftarrow S_\tau \cup$ Trace_Concept($D, S \cup \{\alpha\}$)
return S_τ

Table 5.4: Structural Tracing Algorithm

based on the procedure followed by the tracing algorithm, we can conclude that S_{ax} is a set of axioms that satisfies the following two properties:

1. $S_{ax} \models C \sqsubseteq \bot$ (since e.g., $S_{ax} \models C \sqsubseteq D$, or $S_{ax} \models C \sqsubseteq \exists R_1..R_n D$ where D is unsatisfiable)
2. any proper subset $S'_{ax} \subset S_{ax}$ does not satisfy property (1) above

Now, given that D is unsatisfiable, let J_D = JUSTIFY($D \equiv \bot$, $\mathcal{O}$) and consider any arbitrary set $S_D \in J_D$. Let $S \leftarrow S_D \cup S_{ax}$. Obviously, $S \subseteq \mathcal{O}$.

Since S_{ax} satisfies property (1), it follows that $S \models (C \equiv \bot)$. Moreover, as S_{ax} satisfies property (2), and S_D satisfies the notion of minimality (see Chapter 4), there exists no proper subset $S' \subset S$ such that $S' \models (C \equiv \bot)$. Hence, $S \in$ JUSTIFY($C \equiv \bot$, $\mathcal{O}$). Therefore, C is a derived unsatisfiable class. □

However, the main drawback of the algorithm is that it is incomplete, i.e., it does not discover all dependency relations between unsatisfiable classes.

For example, it does not detect *inferred* equivalence or subsumption between two unsatisfiable classes. Consider two atomic unsatisfiable classes A and B in an ontology that do not have an explicit (asserted) subsumption relation between them but the reasoner can infer one, e.g., $A \equiv (\geq 1p)$ and $B \equiv (\geq 2p)$. Even though there is no subclass axiom relating the two classes, a reasoner can infer that $B \sqsubseteq A$. However, the tracing algorithm shown above cannot find the *hidden* dependence of B on A. In this case, even using a reasoner to infer the subsumption relation will not work as both classes are unsatisfiable and hence effectively equivalent to the bottom class, *owl:Nothing*. As a result, we need an alternate way to discover hidden dependencies between unsatisfiable classes.

Detecting Inferred Dependencies: Subsumption-Revealing Transformations

The problem with detecting hidden dependencies between unsatisfiable classes in an ontology is the masking of useful subsumption relationships, since all unsatisfiable classes are implicitly subsumed by every other class in the ontology.

To resolve this problem, we consider the notion of *subsumption-revealing* transformations to an ontology, i.e., transformations that weaken an ontology by getting rid of the unsatisfiability-causing errors, while preserving the *intended* subsumption hierarchy as much as possible. The weakened ontology can help expose subsumption relationships between the previously unsatisfiable classes.

We use a simple example to illustrate this point. Consider an ontology $\mathcal{O}_5$ with the following four axioms:

$$\boxed{A \equiv D \sqcap \exists R.D \quad A \sqsubseteq \neg D \quad B \equiv C \sqcap \exists R.C \quad C \sqsubseteq D}$$

In $\mathcal{O}_5$, the classes A, B are unsatisfiable. Now, we could argue that the unsatisfiability masks the *intended* subsumption of B by A. This is because the ontology fragment $\{1, 3, 4\} \models (B \sqsubseteq A)$ whereas the addition of axiom 2 to the fragment causes A to be unsatisfiable, which in turn makes B unsatisfiable as well (making both classes equivalent to each other and to $\bot$). Here, detecting that the class B depends on A for its unsatisfiability can help the ontology modeler focus on the root of the problem.

One possible modification that we could make to the above ontology in order to get rid of the unsatisfiability error, while preserving as much information as possible, is to

replace the class $\neg D$ in the RHS of axiom 2 by a new class D' that is previously undefined in the ontology. After applying the above transformation, we can use the reasoner to classify the new ontology in order to detect the hidden subsumption between B and A.

Thus, we have the following algorithm to detect hidden dependencies:

Algorithm: *Subsumption-Revealing_Transformation* *Input*: Ontology $\mathcal{O}$ *Output*: Ontology $\mathcal{O}'$

initialize substitution cache $cache_{sub}$
for each axiom $x \in \mathcal{O}$,
  $x_{NNF} \leftarrow$ normalized version of x in Negation Normal Form (NNF)
  if the formula $\neg C$ is present in x_{NNF}, **then**
  if $C \in cache_{sub}$, **then**
  $D \leftarrow cache_{sub}(C)$
  else
  $D \leftarrow$ new atomic class undefined in $\mathcal{O}$
  substitute $\neg C$ by D in x_{NNF}
  $\mathcal{O}' \leftarrow \mathcal{O}' \cup x_{NNF}$
for each pair of classes $C_1, C_2 \in cache_{sub}$,
  $D_1 \leftarrow cache_{sub}(C_1)$
  $D_2 \leftarrow cache_{sub}(C_2)$
  if $C_1 \sqsubseteq C_2$ **and** $C_1 \neq C_2 \neq \bot$, **then**
  $\mathcal{O}' \leftarrow \mathcal{O}' \cup (D_1 \sqsubseteq D_2)$

Table 5.5: Inferred Dependency Detection Algorithm

Algorithm Analysis and Discussion

The motivation for the above approach is to remove the main cause of class unsatisfiability – negation. Also, given the monotonicity of the logic (OWL-DL), underspecifying the axioms by replacing the negated classes with new classes in the ontology ensures that no new subsumptions are introduced. In order to recover some of the subsumptions that are lost upon substitution, the last 5 lines of the algorithm check the substitution cache for subsumptions between the satisfiable classes (that are replaced) and insert subsumption relations between the corresponding new classes in the ontology.

Suppose $\mathcal{O}, \mathcal{O}'$ are the respective input/output of the algorithm above, and after the classification of $\mathcal{O}'$ by a reasoner, a subsumption relationship is discovered between two previously unsatisfiable classes, say $C \sqsubseteq D$, then it follows that C must be a derived unsatisfiable class in $\mathcal{O}$ and D must be its parent. This is a consequence of Theorem 5 if we consider any set in JUSTIFY($C \sqsubseteq D, \mathcal{O}'$) to reduce to S_{ax}.

Note that the algorithm (heuristic) to detect inferred dependencies is clearly incomplete. However, it provides a cheap and easy solution to detecting more dependencies between unsatisfiable classes, over and above those found by structural tracing.

5.3 Dealing with Inconsistent OWL Ontologies

Many of the techniques discussed in the prior section are, in fact, applicable to the diagnosis of inconsistent ontologies, with a few slight twists. This should be no surprise as unsatisfiability detection is performed by attempting to generate an inconsistent ontology.

First, consider the different kind of reasons for inconsistent ontologies:

1. *Individuals Related to Unsatisfiable Classes or by Unsatisfiable Roles*: There is an unsatisfiable class description and an individual is asserted to belong to that class.

   Similarly, an ontology is inconsistent if there is an unsatisfiable role and there exists a pair of individuals that is an instance of the role. For example, consider a role hasParent whose range is accidentally set to the intersection of classes Father and Mother instead of their union, where Father, Mother are disjoint classes. Here, hasParent is an unsatisfiable role. Thus, defining a relation between individuals using this role, e.g., $\mathsf{hasParent}(\mathsf{Bob}, \mathsf{Mary})$ results in an inconsistent ontology.[3]
2. *Inconsistency of Assertions about Individuals*: There are no unsatisfiable classes in the ontology but there are conflicting assertions about one individual, e.g., an individual is asserted to belong to two disjoint classes or an individual has a cardinality restriction but is related to more distinct individuals.
3. *Defects in Class Axioms Involving Nominals*: It might be the case that inconsistency is not directly caused by type or property assertions, i.e., ABox assertions, but caused by class axioms that involve nominals, i.e., TBox axioms. Nominals are simply individuals mentioned in owl:oneOf and owl:hasValue constructs. As an example consider the following set of axioms:

   $\mathsf{MyFavoriteColor} \equiv \{\mathsf{Blue}\}$
   $\mathsf{PrimaryColors} \equiv \{\mathsf{Red}, \mathsf{Blue}, \mathsf{Yellow}\}$
   $\mathsf{MyFavoriteColor} \sqsubseteq \neg\mathsf{PrimaryColors}$

   These axioms obviously cause an inconsistency because the enumerated classes MyFavoriteColor and PrimaryColors share one element, i.e., individual named Blue, but they are still defined to be disjoint.

Now, irrespective of the type of inconsistency, a generic debugging solution is to use the Axiom Pinpointing service developed in Chapter 4 to obtain all the minimal justification sets (axioms) responsible for the inconsistent ontology.

For example, consider the ontology $\mathcal{O}_6$ shown in Table 5.6:

1. $A \sqsubseteq C \sqcap \neg C$	2. $B \sqsubseteq \exists R.D \sqcap A$	3. $C \sqsubseteq E \sqcap A$
4. $D \sqsubseteq \neg E$	5. $A(a)$	6. $B(b)$
7. $C(c)$	8. $D(d)$	9. $E(e)$
10. $R(b, e)$		

Table 5.6: Example to Capture Core Inconsistency Causing Axioms

$\mathcal{O}_6$ is inconsistent and the justification for this inconsistency is the following axiom sets $\{\{1, 5\}, \{1, 2, 6\}, \{1, 3, 7\}, \{2, 4, 6, 9, 10\}\}$.

[3]Note that debugging an unsatisfiable role R is equivalent to debugging the unsatisfiable concept $\geq 1.R$

The contradiction in each justification set needs to be resolved in order to make the ontology consistent, and no justification set is subsumed by any other as all are minimal by definition. Here, the only analogue to root unsatisfiable classes is *shared* axioms across justification sets, which we can consider as *core* inconsistency causing axioms. In this case, axiom 1 appears in three justification sets and can be seen as a major source of the problems.

The following simple algorithm can be used to return an axiom dependency map that associates each axiom with the justification sets it appears in. The output of the algorithm is a dependency map which can be used to sort and rank axioms based on their *arity*, i.e., the number of justification sets that they jointly appear in.

Algorithm: Core_Axiom Input: Set of axiom sets (S) Output: Axiom Dependency Map η

initialize axiom dependency map η
$total \leftarrow \emptyset$
for each set $s \in S$,
  $total \leftarrow total \cup s$
for each axiom $x \in total$,
  $S_x \leftarrow \emptyset$
  for each set $s \in S$,
  if $x \in s$, **then** $S_x \leftarrow S_x \cup s$
  store $(x \mapsto S_x)$ in η

Table 5.7: Detecting Core Inconsistency Axioms

5.3.1 Special Case: Reduction to Unsatisfiable Classes/Roles

For the first type of inconsistency, instead of using the Axiom Pinpointing service to determine all the justifications, which may be time consuming if the ontology has numerous inconsistency-causing conditions, we can perform some simple ontology modifications to directly expose the main problems.

We can get rid of all the ABox assertions, i.e., assertions of the form $C(a)$ or $R(a, b)$, where C is a class, R is a role and a, b are individuals. Removing these assertions would immediately reveal all the unsatisfiable classes or unsatisfiable roles, which can then be debugged using structural analysis (described in Section 5.2.4) by focusing directly on the root unsatisfiable classes/roles. This approach is likely to give better performance results than using the Axiom Pinpointing service to compute all the minimal justifications (even though the service output would directly point to the root classes) because of the cheap cost of structural analysis.

5.4 Putting It All Together: Service Description

In the previous sections, we have a described a set of algorithms for identifying the main source of the semantic errors in an ontology, both, from a concept and an axiom point of view. We now describe one possible coherent version of the *Root Error*

Pinpointing service that invokes the previous algorithms as and when necessary.

The service outline is shown below:

Service: *Root Error Pinpointing* *Input*: Ontology $\mathcal{O}$

if $\mathcal{O}$ is *inconsistent*, **then**
  $\mathcal{O}' \leftarrow \mathcal{O}$ minus all ABox assertions $C(a), R(a,b) \in \mathcal{O}$
  if $\mathcal{O}'$ is *inconsistent*, **then**
  $S_J \leftarrow JUSTIFY(\mathcal{O}$ is inconsistent)
  invoke algorithm $Core\_Axiom(S_J)$
  else
  Debug_Unsatisfiable($\mathcal{O}'$)
else
  Debug_Unsatisfiable($\mathcal{O}$)
 	
$subroutine$: Debug_Unsatisfiable($\mathcal{O}$)
$S \leftarrow \{C_1, ...C_n\}$ (set of unsatisfiable classes in $\mathcal{O}$)
if $n \geq threshold$, **then**
  invoke algorithm $Structural\_Analysis(\mathcal{O}, S)$
  $S \leftarrow S$ minus derived classes found by $Structural\_Analysis$
invoke algorithm $Generate\_DependencyGraph(\mathcal{O}, S)$
$S_R \leftarrow$ set of root unsatisfiable classes, $S_{J_R} \leftarrow$ set of justifications for each root $r \in S_R$
invoke algorithm $Core\_Axiom(S_{J_R})$

Table 5.8: Root Error Pinpointing

The service receives an ontology that has semantic defects, i.e., either it is inconsistent, or it is consistent with some unsatisfiable classes. In the former case, the service attempts to remove the inconsistency if it is due to unsatisfiable classes by getting rid of all the ABox assertions. The reason for this step is that, if applicable, it highlights the cause of the inconsistency immediately, and moreover, if there are a large number of unsatisfiable classes, it allows us to use $Structural\_Analysis$ (as seen below) to eliminate the less critical unsatisfiable classes quickly.

If the ontology still remains inconsistent after the modification, the service obtains the justification for the inconsistency using the Axiom Pinpointing service and invokes the algorithm $Core\_Axiom$ to generate an axiom dependency map from the justification sets. This map can then be used to highlight the core-erroneous axioms by displaying the corresponding justification sets they fall in.

On the other hand, if the ontology turns consistent as a result of the modification, the service calls the sub-routine $Debug\_Unsatisfiable$ which is used to separate the root from the derived unsatisfiable classes. In this case, depending on whether the number of unsatisfiable classes exceeds some user-specified *threshold*, the service either directly invokes the $Generate\_DependencyGraph$ algorithm, or uses $Structural\_Analysis$ to prune the problematic space quickly by reducing the number of derived unsatisfiable classes before generating the graph for the remaining erroneous ones.

Finally, once the user has narrowed down the root unsatisfiable classes to focus on, the Axiom Pinpointing Service can be used to obtain the justifications for each of the roots, and the $Core\_Axiom$ algorithm can be used (as seen above) to generate an axiom dependency map from the justification sets.

Chapter 6
Ontology Repair Service
6.1 Introduction

In Chapters 4, 5, we have devised a set of ontology debugging services that can be used to highlight the core erroneous axioms and concepts in a defected ontology. After identifying and understanding the cause of the error, the next step is to act upon it, i.e., resolve the error by modifying the ontology in an appropriate manner. Though in most cases, repairing errors is left to the ontology modelers' discretion, and understanding the cause of the error certainly helps make resolving it much easier, bug resolution can still be a non-trivial task, requiring an exploration of remedies with a cost/benefit analysis. For this reason, we present a service specifically catered towards ontology repair.

Given an OWL ontology with one or more unsatisfiable classes (or alternately, an inconsistent OWL ontology), the ontology repair service automatically generates repair solutions, i.e., a set of ontology changes, which if applied to the ontology eliminate all the concerned errors.

In designing this service, we consider various strategies to *rank* erroneous axioms in order to arrive at sensible solutions. For example, one of the metrics used for axiom ranking is the impact of removing the axiom on the remaining entailments of the ontology. Roughly, the idea here is to assign a high rank to an erroneous axiom if removing it from the ontology has a very small impact on the semantics of the ontology. In order to generate repair solutions based on the axiom ranks, we use a standard uniform cost search algorithm. We modify the algorithm to allow for easy customization of the repair solutions based on the modelers' preferences.

We also note that the repair service considers axiom additions or rewrites as well, and not just the removal of axioms. Axiom rewrites are desirable because they attempt to preserve the meaning of the axioms as much as possible, while eliminating the problematic parts. In quite a few cases that we have observed, the quality of the repair solutions is greatly enhanced when rewrites are considered.

In the remainder of this chapter, we describe the key components of the repair service when used to debug unsatisfiable classes in a consistent ontology. Since the underlying problem involves dealing with and rectifying a set of erroneous axioms, the same principles for generating repair solutions are applicable when debugging an inconsistent ontology.

6.2 Repair Overview: Scope and Limitations

In this section, we provide a brief overview of how the repair service works, and discuss it's scope and limitations.

We consider a simple example to illustrate the main points. Let $\mathcal{O}_1$ be an ontology composed of the following axioms:

1. Person $\sqsubseteq (= 1)$.hasGender
2. Gender(male)
3. Gender(female)
4. Person $\sqsubseteq \neg$Animal
5. {male} $\sqsubseteq \neg${female}
6. domain(hasGender) = Animal
7. range(hasGender) = {male} $\sqcap$ {female}
8. Student $\sqsubseteq$ Person

The classes Person, Student are unsatisfiable in $\mathcal{O}_1$. The objective of the repair service is to generate a solution (set of ontology changes) to fix the two unsatisfiable classes.

Now, the Axiom Pinpointing service devised in Chapter 4 can be used to obtain the justification for the unsatisfiability of each of these classes, i.e., the minimal set of axioms from the ontology which is responsible for their unsatisfiability. For example, the justification for the entailment Person $\equiv \bot$ is the two axiom sets: $\{1, 4, 6\}, \{1, 5, 7\}$. This justification is also referred to as the MUPS of the unsatisfiable concept, as seen in Chapter 4.

From a repair point of view, the significance of the MUPS(Person) is clear – in order to make the class Person satisfiable in $\mathcal{O}_1$, we need to remove from $\mathcal{O}_1$ at least one axiom in each set present in the MUPS(Person). Thus, the repair service uses this information to automatically generate a minimal repair solution to fix this bug, e.g., assuming we do not want to remove axiom 1 since it is the concept definition axiom, one solution is to remove axioms $\{4, 7\}$ from $\mathcal{O}_1$.

Moreover, if the ontology has numerous unsatisfiable classes, as is the case above, the Root-Error Pinpointing Service devised in Chapter 5 can be used to identify the core errors, by separating the *root* from the *derived* unsatisfiable classes, e.g., in $\mathcal{O}_1$, the class Student is *purely derived*, while Parent is a root, because of the subclass dependency relation in axiom 8. This implies that any repair solution to fix Parent is guaranteed to make the class Student satisfiable as well. Thus, the repair service makes use of this knowledge to arrive at a solution that removes the *least number of axioms* from the ontology while repairing *all* the unsatisfiable bugs.

However, the main drawback of automatically generating a solution is that it is impossible to determine what a *correct* or *appropriate* repair solution is for every case, since assessing the quality of a solution is left to the ontology authors' discretion. In the $\mathcal{O}_1$ example, an alternate solution that contains a minimal axiom set (not including the concept definition axiom 1) is the set $\{5, 6\}$, though this solution is probably undesired. Obviously, there is no way a tool can automatically distinguish between desired and undesired solutions. In the absence of any domain knowledge or modeler intent, the only option is to take into account suitable heuristics to ensure that the service arrives at reasonable solutions, present alternatives to the user and facilitate feedback to improve their quality.

Thus, the naive, straightforward design of the repair service that uses the previous debugging services to come up with axiom-removal solutions is modified as shown in Figure 6.1. It includes the following modules: an *Axiom Ranking* module that uses various strategies to prioritize erroneous axioms, a *Solution Generation* module that automatically generates repair plans which can be customized easily, and an *Axiom Rewrite* module that

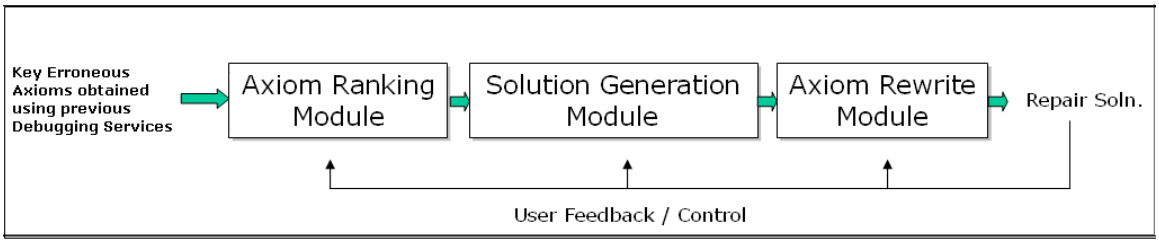

Figure 6.1: **Ontology Repair Service**

enhances the solutions by suggesting appropriate axiom edits where possible to the user. The purpose of these modules is to create a service that aids the user in understanding and evaluating the options available for repair.

6.3 Axiom Ranking Module

Given a set of erroneous axioms in an ontology, the key task for repair is selecting *which* of the axioms need to be modified or removed. For this purpose, we consider whether axioms can be *ranked* in order of importance. Repair is then reduced to an optimization problem whose primary goal is to get rid of all the inconsistency errors in the ontology, while ensuring that the highest rank axioms are preserved and the lowest rank axioms removed from the ontology.

In this section, we describe the Axiom Ranking module of our Ontology Repair service. This module uses the following strategies to rank erroneous axioms:

- *Frequency*: the number of times the axiom appears in the MUPS of the various unsatisfiable concepts in an ontology. If an axiom appears in n different MUPS (in each set of the MUPS), removing the axiom from the ontology ensures that n concepts turn satisfiable. Thus, higher the frequency, lower the rank assigned to the axiom
- *Semantic* relevance to the ontology, in terms of the impact (i.e., entailments lost or added) when the axiom is removed or altered. Greater the impact caused by removing the axiom, higher it's assigned rank and vice versa.
- *Test cases* specified manually by the user to rank axioms. Axioms are ranked in direct (or inverse) proportion to desired (or undesired) entailments specified by the user.
- *Syntactic* relevance to the ontology, in terms of the usage of the elements in the axiom signature. Axioms related to elements that are strongly connected in the ontology graph are ranked higher and vice versa.

Among the above strategies, determining the *frequency* of the axiom is straightforward once the MUPS of the unsatisfiable concepts has been determined (using the Axiom Pinpointing service). We now describe each of the remaining strategies in detail in the following subsections.

6.3.1 Semantic Relevance: Impact Analysis

The basic notion of revising a knowledge base while preserving as much information as possible has been discussed extensively in belief revision literature [1]. We now apply the same principle to repairing unsatisfiable concepts in an OWL ontology, i.e., we determine the impact of the changes made to the ontology in order to get rid of unsatisfiable concepts, and identify minimal-impact causing changes. Since repairing an unsatisfiable concept involves removing axioms in it's MUPS, we consider the impact of axiom removal on the OWL ontology.

A fundamental property of axiom removal based on the monotonicity of OWL-DL is the following: *removing an axiom from the ontology cannot add a new entailment.* Hence, we only need to consider entailments (subsumption, instantiation etc.) that are lost upon axiom removal, and need not consider whether other concepts in the ontology turn unsatisfiable.

For the purpose of impact analysis, we present a simple definition of *semantic relevance*.

Definition 1 *(Semantic Relevance)*

Given an ontology $\mathcal{O}$ with axiom α, the semantic relevance of α, given by SR_α, is a set of entailments $\{\beta_1, ..\beta_n\}$ such that for each entailment $\beta_i \in SR_\alpha$ $(1 \leq i \leq n)$, it holds that $\mathcal{O} \models \beta_i$ but $(\mathcal{O} - \alpha) \not\models \beta_i$.

The above definition is quite broad as it allows an arbitrarily infinite set of entailments to be considered as semantically relevant (e.g., if an ontology entails $C \sqsubseteq D$, it also entails $C \sqsubseteq D \sqcup D'$ where D' is any arbitrary concept), hence we shall only consider subsumption/disjointness between *atomic* concepts and instantiation of *atomic* concepts as the key entailments to check for when an axiom is removed. In the next subsection, we discuss how the user can provide a set of test cases as additional interesting entailments to check for.

Note that axiom ranks are assigned in direct proportion to their semantic relevance, i.e., higher the semantic relevance, more the entailments that are lost upon it's removal, and hence higher the axiom rank.

Computing Semantic Relevance

In order to compute the semantic relevance of an axiom w.r.t. some key entailments, a brute-force technique involves processing the ontology using a DL reasoner by removing the concerned axiom and noting the entailments lost. Obviously, performance issues are the main concern here, especially when dealing with large ontologies containing thousands of axioms. Though we are exploring techniques for incremental reasoning for dynamic (changing) ontologies [80], this is still largely an unexplored field.

A more optimal solution is employed by our Ontology Repair service and the algorithm is shown in Table 6.1.

The algorithm accepts as input the OWL ontology $\mathcal{O}$, a set of erroneous axioms S responsible for the various logical errors in it, and a weighting factor wt used for computing ranks. It returns a map (bijection) M that associates each erroneous axiom with the

Algorithm: *Compute Semantic Relevance*
Input: Ontology $\mathcal{O}$, Set of erroneous axioms S, weighting factor wt
Output: Entailment Map M, Rank function $rank$

while classifying $\mathcal{O}$ using a reasoner,
  for each subsumption $C \sqsubseteq D$,
    if C is unsatisfiable,
      $handleUnsat(C \sqsubseteq D)$
    else
      compute $JUSTIFY(C \sqsubseteq D)$ using Axiom Pinpointing (tableau tracing, ref. Ch4)
      for each axiom $\alpha \in S$ s.t. $\alpha \in JUSTIFY(C \sqsubseteq D)$,
        $M(\alpha) \rightarrow M(\alpha) \cup \{C \sqsubseteq D\}$
while realizing $\mathcal{O}$ using a reasoner,
  for each instantiation $C(a)$,
    compute $JUSTIFY(C(a))$ using Axiom Pinpointing (tableau tracing, ref. Ch4)
    for each axiom $\alpha \in S$ s.t. $\alpha \in JUSTIFY(C(a))$,
      $M(\alpha) \rightarrow M(\alpha) \cup \{C(a)\}$
for each axiom entry α in M and entailment $\mathcal{E} \in M(\alpha)$
  if $(\mathcal{O} - \alpha) \models \mathcal{E}$
    $M(\alpha) \leftarrow M(\alpha) - \mathcal{E}$
for each axiom $\alpha \in S$,
  $rank(\alpha) \leftarrow sizeof(M(\alpha)) * wt$

subroutine: $handleUnsat(C \sqsubseteq D)$
  use *Structural Analysis* (ref. Ch5) to obtain $T \subseteq \mathcal{O}$ s.t. $T \models C \sqsubseteq D$ and C is satisfiable in T
  for each axiom $\alpha \in S$ s.t. $\alpha \in T$,
    $M(\alpha) \rightarrow M(\alpha) \cup \{C \sqsubseteq D\}$
return

Table 6.1: Computing Semantic Relevance

entailments that are lost from the ontology when the axiom is removed, and a function $rank$ that assigns an axiom rank based on the entailments associated with the axiom in M and the value of wt specified.

The idea behind the algorithm is the following: we use the Axiom Pinpointing service (seen in Chapter 4) to obtain the justification sets (axioms) responsible for the significant subsumption and/or instantiation relationships in the ontology, and then directly determine the justification sets the axiom falls in. Since the tableau tracing (Glass-box) version of the Axiom Pinpointing service does not impose much overhead over the regular reasoning procedure, we can easily compute a single justification set for each entailment during reasoning. However, since we only find one justification set for the entailment, we need to check whether the entailment would actually be lost when the axiom in the set is removed. The second to last loop in the main algorithm verifies this. Note that the number of entailments tested as a result of this algorithm is a fraction of the total set of entailments that would have been tested if one were to use the brute force method.

In addition, the algorithm makes use of a subroutine $handleUnsat(..)$ to deal with entailments related to unsatisfiable classes, which represent a special case. This is because when a concept is unsatisfiable, it is equivalent to the bottom concept (or in the OWL language, `owl:Nothing`), and hence is trivially equivalent to all other unsatisfiable concepts, and is a subclass of all satisfiable concepts in the ontology. In this case, we need

to differentiate between the stated or explicit entailments related to unsatisfiable concepts and the trivial ones. Thus, we apply the following strategy: if a given entailment related to an unsatisfiable concept holds in a *fragment* of the ontology in which the concept is *satisfiable*, we consider the entailment to be explicit. While this is a hard problem, the subroutine uses the *Structural Analysis* techniques seen in Chapter 5 to detect explicit relationships involving unsatisfiable concepts without performing large scale ontology changes. [1]

We consider a few examples that highlight the significance of semantic relevance.

Example 1: In the Tambis OWL ontology[2], the following set of axioms cause 77 unsatisfiable classes:

1. metal $\equiv$ chemical $\sqcap (= 1)$.atomic-number $\sqcap \exists$atomic-number.integer
2. non-metal $\equiv$ chemical $\sqcap (= 1)$.atomic-number $\sqcap \exists$atomic-number.integer
3. metalloid $\equiv$ chemical $\sqcap (= 1)$.atomic-number $\sqcap \exists$atomic-number.integer
4. metal $\sqsubseteq \neg$non-metal
5. metalloid $\sqsubseteq \neg$non-metal
6. metalloid $\sqsubseteq \neg$metal

In this case, though the disjoint axioms appear in the MUPS of each of the three unsatisfiable concepts, metal, non-metal, metalloid, removing them is not the correct solution, since eliminating them removes the disjointness relations between numerous other classes in the ontology and also makes all three concepts above equivalent which is probably undesired.

In fact, a better solution is to weaken the equivalence to a subclass relationship in each concept definition, thereby getting rid of the subclasses: chemical $\sqcap (= 1)$atomic-number $\sqcap \exists$atomic-number.integer $\sqsubseteq$ metal/non-metal/metalloid; and we find that removing these relationships has no impact on other entailments in the ontology.

Example 2: Consider the following $MUPS$ of an unsatisfiable concept OceanCrustLayer w.r.t. the Sweet-JPL ontology $\mathcal{O}_2$:

1. OceanCrustLayer $\sqsubseteq$ CrustLayer
2. CrustLayer $\sqsubseteq$ Layer
3. Layer $\sqsubseteq$ Geometric_3D_Object
4. Geometric_3D_Object $\sqsubseteq \exists$hasDimension.{"3D"}
5. OceanCrustLayer $\sqsubseteq$ OceanRegion
6. OceanRegion $\sqsubseteq$ Region
7. Region $\sqsubseteq$ Geometric_2D_Object
8. Geometric_2D_Object $\sqsubseteq \exists$hasDimension.{"2D"}
9. hasDimension is Functional

Note that in $\mathcal{O}_2$, each of the concepts CrustLayer, OceanRegion, Layer, Region,

[1] For example, we use the *Inferred Dependency Detection* heuristic to get rid of the contradictions in the ontology while revealing the hidden subsumption entailments. Our evaluation in Chapter 7 demonstrates that heuristics based on this technique work quite well in practice.

[2] Note: All ontologies mentioned in this paper are available online at http://www.mindswap.org/ontologies/debugging/

Geometric_3D_Object, Geometric_2D_Object, has numerous individual subclasses.

In this case, removing the functional property assertion on hasDimension from $\mathcal{O}_2$ eliminates the disjoint relation between concepts Geometric_2D_Object and Geometric_3D_Object, and between all it's respective subclasses. Also, removing any of the following axioms $2, 3, 4, 6, 7, 8$ eliminates numerous subsumptions from the original ontology. Thus, using the minimal impact strategy, the only option for repair is removing either 1 or 5, which turns out to be the correct solution, based on the feedback given by the original ontology authors.

6.3.2 User Test Cases

In addition to the standard entailments considered in the previous subsection, the user can specify a set of test cases describing desired entailments (similar to the idea proposed in [35]). Axioms to be removed are then directly ranked based on the desired entailments they break.

Also, in some cases, the user can specify *undesired* entailments to aid the repair process. For example, a common modeling mistake is when an atomic concept C inadvertently becomes equivalent to the top concept, `owl:Thing`. Now, any atomic concept disjoint from C becomes unsatisfiable. This phenomenon occurred in the CHEM-A ontology, where the following two axioms caused concept A (anonymized) to become equivalent to $\top$: $\{A \equiv \forall R.C, domain(R, A)\}$.

To incorporate test cases such as these into the algorithm shown in Table 6.1, we modify it to allow two more input arguments – a set of user-specified entailments and a function which annotates entailments as desired or undesired. Then, during the main routine, we obtain the justifications of the manually specified entailments (in addition to the standard ones) and verify if the axiom removal breaks such entailments or not (by checking the justifications). Finally, while computing the function $rank$ based on the entailment map M, we use the information about whether the entailment is desired or not to assign a positive or negative valued weight respectively.

6.3.3 Syntactic Relevance

There has been research done in the area of ontology ranking [81], [29], where for example, terms in ontologies are ranked based on their structural connectedness in the graph model of the ontology, or their popularity in other ontologies, and the total rank for the ontology is assigned in terms of the individual entity ranks. Since an ontology is a collection of axioms, we can, in theory, explore similar techniques to rank individual axioms. The main difference, of course, lies in the fact that ontologies as a whole can be seen as documents which link to (or import) other ontology documents, whereas the notion of linkage is less strong for individual axioms.

Here, we present a simple strategy that ranks an axiom based on the *usage* of elements in it's signature. For this, we define the notion of syntactic relevance.

Definition 2 *(Syntactic Relevance)*

Given an ontology $\mathcal{O}$ with axiom α, let $sign(\alpha) = \{\mathcal{E}_1, ..\mathcal{E}_n\}$ be the signature of α, where $\mathcal{E}_i$ is either an atomic concept, role or individual in the vocabulary of $\mathcal{O}$. The usage of an entity $\mathcal{E}_i$, given by $usage(\mathcal{E}_i)$, is the set of axioms $S = \{\alpha_1, ...\alpha_m\}$, ($S \subseteq \mathcal{O}$), s.t. for each $\alpha_i \in S$, $\mathcal{E}_i \in sign(\alpha_i)$. Then, the syntactic-relevance rank of the axiom α is given by: $\mathsf{size}(\mathsf{usage}(\mathcal{E}_1)).. \cup \mathsf{usage}(\mathcal{E}_n))$.

The significance of this strategy is based on the following intuition: if the entities in the axiom are used (or are referred to) often in the remaining axioms or assertions of the ontology, then the entities are in some sense, core or central to the overall theme of the ontology, and hence changing or removing axioms related to these entities may be undesired. For example, if a certain concept is heavily instantiated, or if a certain property is heavily used in the instance data, then altering the axiom definitions of that concept or property is a change that the user needs to be aware of. Similarly, in large ontologies where certain entities are accidentally underspecified or unused, axioms related to these entities may be given less importance.

An algorithm to determine the syntactic relevance is shown below. Similar to the algorithm depicted in Table 6.1, it accepts as input the OWL ontology, a set of erroneous axioms and a weighting factor used to compute axiom ranks. It enforces the semantics of Definition 2 and assigns ranks based on the usage of entities in the signature of the erroneous axiom.

Algorithm: *Compute Syntactic Relevance* *Input*: Ontology $\mathcal{O}$, Set of erroneous axioms S, weighting factor wt, axiom type weight function τ *Output*: Rank function $rank$

initialize entity usage map M_u
for each axiom $\alpha \in \mathcal{O}$,
  $sign(\alpha) \leftarrow$ signature of axiom α
  for each entity (class, property, individual) $\mathcal{E} \in sign(\alpha)$,
  $M_u(\mathcal{E}) \leftarrow M_u(\mathcal{E}) \cup \alpha$
for each axiom $\alpha \in S$, s.t. $rank(\alpha) = 0$
  for each entity $\mathcal{E} \in sign(\alpha)$,
  $rank(\alpha) \leftarrow rank(\alpha) \cup M_u(\alpha) * \tau(\alpha)$
  $rank(\alpha) \leftarrow rank(\alpha) * wt$

Table 6.2: Computing Syntactic Relevance

In order to make the ranking approach more flexible, an additional input to the algorithm is a function τ that assigns weights based on various axiom types, e.g., it allows weighing property attribute assertions such as owl : InverseFunctional higher. This function specified by the user would be motivated by the ontology modeling philosophy and purpose (e.g., as is done in OntoClean [38], where certain concept / property definition types are given higher importance).

6.4 Solution Generation Module

So far, we have devised a procedure to find ranks for various erroneous axioms (MUPS) in the ontology. The next step is to generate a repair plan (i.e., a set of ontology changes) to resolve the unsatisfiable or inconsistency errors taking into account their

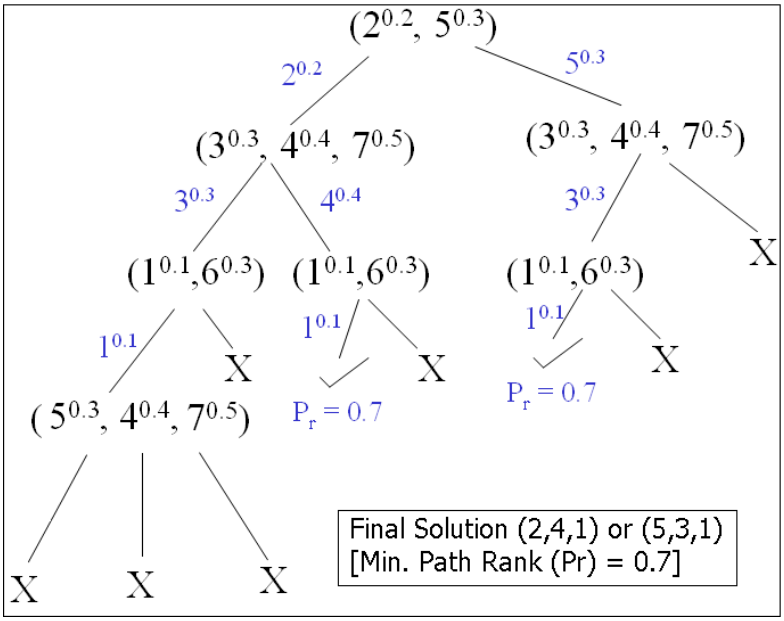

Figure 6.2: **Uniform Cost Search:** Generating a repair plan based on ranks of axioms in the MUPS of unsatisfiable concepts.

respective MUPS and axiom ranks. This is handled by the solution generation module, which uses a standard *uniform cost search* algorithm taking the computed axiom ranks as the cost.

Figure 6.2 shows an example of a search tree generated by the algorithm for a collection of erroneous axiom sets $C = \{\{2,5\},\{3,4,7\},\{1,6\},\{4,5,7\},\{1,2,3\}\}$ with the axioms $1-7$ ranked as follows: $r(1) = 0.1$, $r(2) = 0.2$, $r(3) = 0.3$, $r(4) = 0.4$, $r(5) = 0.3$, $r(6) = 0.3$, $r(7) = 0.5$, where $r(x)$ is the rank of axiom x. The ranks are computed based on the factors mentioned earlier, such as frequency, impact analysis etc. each weighed separately if needed using appropriate weight constants. The superscript for each axiom-number denotes the rank of the axiom, and P_r is the path rank computed as the sum of the ranks of axioms in the path from the root to the node. For example, for the leftmost path shown: $P_r = 0.2 + 0.3 + 0.1 + 0.3 = 0.9$.

As shown in the figure, the repair solution found with the minimal cost is either {2,4,1} or {5,3,1}.

However, there is a drawback of using the above procedure to generate repair plans, i.e., impact analysis is only done at a single axiom level, whereas the cumulative impact of the axioms in the repair solution is not considered. This can lead to non-optimal solutions. For example, in the Tambis OWL ontology seen earlier, where the three root classes are asserted to be mutually disjoint, removing any one of the disjoint axioms does not cause as large an impact as removing all the disjoints together.

In order to resolve this issue, we propose a slight modification to the algorithm: each time a solution is found, we compute a new cost based on the cumulative impact of the axioms in the solution. The algorithm now finds repair plans that minimize these updated costs.

6.4.1 Improving and Customizing Repair

The algorithm described above can be used in general to fix any arbitrary set of unsatisfiable concepts, once the MUPS of the concepts and the ranks for axioms in the MUPS is known. Thus, a brute force solution for resolving *all* the errors in an ontology involves determining the MUPS (and ranking axioms in the MUPS) for *each* of the unsatisfiable concepts. This is computationally expensive and moreover, unnecessary, given that strong dependencies between unsatisfiable concepts may exist. Thus, we need to focus on the MUPS of the critical or root contradictions in the ontology.

To achieve this, we make use of the Root-Error Pinpointing service described in Chapter 5 that identifies the *root* unsatisfiable concepts in an ontology, which propagate and cause errors elsewhere in the ontology, leading to *derived* unsatisfiable concepts. Recall that a root unsatisfiable concept is one in which a clash or contradiction found in the concept definition does not *depend on the unsatisfiability* of another concept in the ontology; whereas, a derived unsatisfiable concept acquires a contradiction due to it's dependence on another unsatisfiable concept. For example, if A is an unsatisfiable concept, then a concept B ($B \sqsubseteq A$) or C ($C \sqsubseteq \exists R.A$) also becomes unsatisfiable due to it's dependence on A, and is thus considered as derived. From a repair point of view, the key advantage here is that one needs to focus on the MUPS of the root unsatisfiable concepts alone since fixing the roots effectively fixes a large set of directly derived concept bugs.

Also, the service guides the repair process which can be carried out by the user at three different granularity levels:

- *Level 1: Reparing a single unsatisfiable concept at a time*: In this case, it makes sense to deal with the root unsatisfiable concepts first, before resolving errors in any of the derived concepts. This technique allows the user to monitor the entire debugging process closely, exploring different repair alternatives for each concept before fully fixing the ontology. However, since at every step in the repair process, the user is working in a localized context (looking at a single concept only), the debugging of the entire ontology could be prolonged due to new bugs introduced later based on changes made earlier. Thus, the repair process may not be optimal.

- *Level 2: Repairing all root unsatisfiable concepts together*: The user could batch repair all the root unsatisfiable concepts in a single debugging iteration before proceeding to uncover a new set of root/derived unsatisfiable concepts. This technique provides a cross between the tool-automation (done in level 3) and finer manual inspection (allowed in level 1) with respect to bug correction.

- *Level 3: Repairing all unsatisfiable concepts*: The user could directly focus on removing all the unsatisfiable concepts in the ontology in one go. This technique imposes an overhead on the debugging tool which needs to present a plan that accounts for the removal of all the bugs in an optimal manner. The strategy works in a global context, considering bugs and bug-dependencies in the ontology as a whole, and thus may take time for the tool to compute, especially if there are a large number of unsatisfiable concepts in the ontology (e.g. Tambis). However, the repair process is likely to be more efficient compared to level 1 repair.

Asserted	Meant	Reason for Misunderstanding
$A \equiv C$	$A \sqsubseteq C$	Difference between Defined and Primitive concepts
$A \sqsubseteq C$ $A \sqsubseteq D$	$A \sqsubseteq C \sqcup D$	Multiple subclass has intersection semantics
`domain(P,A)` `range(P,B)`	$A \sqsubseteq \forall P.B$	Global vs. Local property restrictions
`domain(P,A)` `domain(P,B)`	domain(P, $A \sqcup B$)	Unclear about multiple domain semantics

Table 6.3: Common Errors in OWL

The number of steps in the repair process depends on the granularity level chosen by the user: for example, using Level 1 above, the no. of steps is atleast the no. of unsatisfiable concepts the user begins with; whereas using Level 3 granularity, the repair reduces to a single big step. To make the process more flexible, the user is allowed to change the granularity level, as and when desired, during a particular repair session (see section 6.6: Putting It All Together).

6.5 Axiom Rewrite Module

To make our repair solution more flexible, the Axiom Rewrite module considers strategies to edit erroneous axioms instead of strictly removing them from the ontology. An important point to note here is that if the rewrite is not a *strict weakening*, we need to determine the new entailments that are *introduced* because of the rewrite. At the very least, no new unsatisfiable concepts should arise because of the rewrite, and currently we test this using the reasoner directly. As future work, we plan to perform a more elaborate analysis of the added entailments by making use of techniques we are developing for incremental reasoning [42], [80].

Using Erroneous Axiom Parts

As shown in Chapter 4, the Axiom Pinpointing service can be used to output precise justifications (in addition to asserted justifications) which identify parts of axioms in the MUPS responsible for making a concept unsatisfiable. Having determined the erroneous part(s) of axioms, the module suggests a suitable rewrite of the axiom that preserves as much as information as possible while eliminating unsatisfiability.

Identifying Common Pitfalls

Common pitfalls in OWL ontology modeling have been enumerated in literature [87]. We have summarized some commonly occurring errors that we have observed (in addition to those mentioned in [87]), highlighting the *meant* axiom and the reason for the mistake in each case.

The library of error patterns is used in the axiom rewrite module as follows: once an axiom responsible for an unsatisfiable concept is identified, we check if the axiom has

a pattern corresponding to one in the library, and if so, suggest the *meant* axiom to the user as a replacement. As future work, we plan to make this library easily extensible and shareable among ontology authors and users.

6.6 Putting It All Together: Service Description

In the previous sections, we have described the various modules of the Ontology Repair Service. We now describe a single coherent version of the service that ties the modules together (see Table 6.4).

During the execution of the repair service, the user is asked for his/her preferences regarding the granularity level (g) of the solution, and additional information used to compute ranks for erroneous axioms (weight-function for the various axiom types (τ) and ranking metrics weights). Based on these preferences, the service computes an appropriate repair solution by using a uniform cost search algorithm as described earlier (it uses the subroutine $GenerateSolution$ for this task).

Note that, where necessary, the service makes use of the Axiom Pinpointing service to find the precise justifications of the unsatisfiable concept, and the Root Error Pinpointing service to find root unsatisfiable concepts. Also note that when generating a solution to repair all the unsatisfiable concepts, the service works iteratively – considering only the root unsatisfiable concepts in each iteration.

Finally, the Axiom Rewrite module (denoted by AXIOM-REWRITE(..) in Table 6.4 but not explicated), replaces *subaxioms* in the solution set by appropriate axiom rewrites, e.g., removing newly introduced terms in the subaxioms; and also performs a lookup in the error pattern library for possible rewrite suggestions.

6.7 Conclusion / Outlook

In this chapter, we have discussed the key design factors of our Ontology Repair service, namely, metrics for ranking axioms that contribute to the inconsistency, generation of repair plans based on axiom ranks, and techniques to suggest axiom rewrites when possible. A nice outcome has been the use of services devised in the earlier chapters in the various stages of repair, e.g., ranking axioms based on entailments they justify, generating plans faster using root/derived unsatisfiable classes, and suggesting rewrites based on precise justifications.

However, the repair service is different in spirit from the services seen in Chapters 4, 5. The latter services are not dependent on human factors such as modeler's intent, background domain knowledge etc., making them more concrete or well-defined, whereas the repair process is more interactive, heuristic-based and user-driven. This also implies that User Interface (UI) issues play a larger role in determining the effectiveness of the repair service, as compared to the earlier two services. We discuss the UI details of our repair tool in the implementation and evaluation chapter (Chapter 7).

Algorithm: Ontology Repair Service
Input: OWL Ontology $\mathcal{O}$

while $\mathcal{O}$ contains some unsatisfiable concepts,
  ask user for: repair granularity level g, axiom-type weight fn. τ, ranking metric weights w_f, w_i, w_u
  $GenerateSolution(\mathcal{O}, g, \tau, w_f, w_i, w_u)$
  $soln \leftarrow$ AXIOM-REWRITE$(soln)$

subroutine: $GenerateSolution(\mathcal{O}, g, \tau, w_f, w_i, w_u)$
  $S \leftarrow \emptyset$, $soln \leftarrow \emptyset$
  if g = 1 (repair one unsatisfiable concept at a time, say some arbitrary concept C)
    $S \leftarrow JUSTIFY_{precise}(C \sqsubseteq \bot)$ (obtained using Axiom Pinpointing)
    $soln \leftarrow repairAxiomSets(\mathcal{O}, S, \tau, w_f, w_i, w_u)$
  else if g = 2 (repair all roots)
    $R \leftarrow$ set of root unsatisfiable concepts (obtained using Root Error Pinpointing)
    for each root concept $r \in R$,
      $S \leftarrow S \cup JUSTIFY_{precise}(r \sqsubseteq \bot)$ (obtained using Axiom Pinpointing)
    $soln \leftarrow repairAxiomSets(\mathcal{O}, S, \tau, w_f, w_i, w_u)$
  else g = 3 (repair all unsatisfiable concepts)
    while there exists at least one unsatisfiable concept in $\mathcal{O}$,
      $R \leftarrow$ set of root unsatisfiable concepts (obtained using Root Error Pinpointing)
      for each root concept $r \in R$,
        $S \leftarrow S \cup JUSTIFY_{precise}(r \sqsubseteq \bot)$ (obtained using Axiom Pinpointing)
      $soln_{itn} \leftarrow repairAxiomSets(\mathcal{O}, S, \tau, w_f, w_i, w_u)$
      remove axioms in $soln_{itn}$ from $\mathcal{O}$
      $soln \leftarrow soln \cup soln_{itn}$
  return $soln$

subroutine: $computeRanks(\mathcal{O}, S, \tau, w_f, w_i, w_u)$
  for each set $m \in S$,
    for each axiom $\alpha \in m$
      $freq \leftarrow$ number of sets in S that α falls in
      $rank_f(\alpha) \leftarrow -w_f * freq$
    $rank_i \leftarrow compute\_Semantic\_Relevance(\mathcal{O}, m, w_i)$
    $rank_u \leftarrow compute\_Syntactic\_Relevance(\mathcal{O}, m, w_u, \tau)$
  for each axiom $\alpha \in m$ where $m \in S$,
    $rank(\alpha) \leftarrow rank_f(\alpha) + rank_i(\alpha) + rank_u(\alpha)$
  return $rank$

subroutine: $repairAxiomSets(\mathcal{O}, S, \tau, w_f, w_i, w_u)$
  $rank \leftarrow computeRanks(\mathcal{O}, S, \tau, w_f, w_i, w_u)$
  $soln \leftarrow$ uniform-cost-search$(S, rank)$
  return $soln$

Table 6.4: Ontology Repair Service

Chapter 7
Implementation and Evaluation

In this chapter, we discuss the key issues involved in deploying the debugging and repair services seen in Chapters 4-6, and present results demonstrating the practical significance of these services.

The chapter is divided into two main sections – in Section 7.1, we describe implementation details of each service, discuss human factors involved, and present performance evaluations of the service. In section 7.2, we describe the results of two pilot studies that were conducted to judge the overall use and benefit of the services.

We note that all the debugging and repair services have been implemented in the OWL-DL reasoner Pellet[1], and as part of the OWL Ontology Editor toolkit, Swoop[2]. For a detailed background of Swoop and Pellet, we refer the reader to [57], [97], and the Appendix.

Before proceeding, we mention the main sample data used in our experiments – we selected existing OWL ontologies that varied greatly in size, complexity and expressivity[3]. The details of the ontologies are given in Table 7.1.

The table displays the DL expressivity of each ontology, followed by the number of axioms, classes/properties/individuals and unsatisfiable classes in the ontology, and a small background description. With the exception of the University OWL ontology that we built for training purposes, all the remaining ontologies have been built by third parties.

Ontology	**DL Expressivity**	**Axioms**	**C/P/I/Unsat.**	**Comments**
Chemical	$\mathcal{ALCH}(\mathcal{D})$	254	48/ 20/ -/ 37	Ontology dealing with chemical elements containing real modeling errors
DOLCE	$\mathcal{SHOIN}(\mathcal{D})$	1417	200/ 299/ 39/ -	Foundational ontology for linguistic and cognitive engineering
Economy	$\mathcal{ALH}(\mathcal{D})$	1704	338/ 53/ 481/ 51	Mid-level ontology by Teknowledge
Galen	$\mathcal{SHF}$	6580	2749/ 413/ -/ -	An adaptation of an early prototype of the GALEN Clinical Terminology
Sweet-JPL	$\mathcal{ALCHO}(\mathcal{D})$	3833	1537/ 121/ 150/ 1	NASA ontology dealing with Earthscience
Tambis	$\mathcal{SHIN}$	800	395/ 100/ -/ 144	A biological science ontology developed by the TAMBIS project
Transport	$\mathcal{ALH}(\mathcal{D})$	2051	444/ 93/ 183/ 55	Mid-level ontology by Teknowledge
University	$\mathcal{SIOF}(\mathcal{D})$	169	30/ 12/ 4/ 8	Training ontology hand-crafted to demonstrate non-trivial errors
Wine	$\mathcal{SHIF}(\mathcal{D})$	856	77/ 18 /206/ -	Expressive Ontology used in the OWL Guide (modified to remove nominals)

Table 7.1: Sample OWL Data used in our Debugging/Repair Experiments

[1]Pellet: http://www.mindswap.org/2003/pellet

[2]Swoop: http://www.mindswap.org/2004/SWOOP

[3]Note: The ontologies are available at http://www.mindswap.org/ontologies/debugging.

7.1 Deploying the Debugging & Repair Services

In this section, we focus on the three ontology debugging/repair services – *Axiom Pinpointing*, *Root Error Pinpointing* and *Ontology Repair* separately and discuss their implementation and presentation issues in Swoop. The first two services also include a performance (timing) evaluation. For the third service, i.e., Ontology Repair, timings are included as part of the user study described in section 7.2.

7.1.1 Implementing Axiom Pinpointing

Recall that the Axiom Pinpointing service is used to obtain the justification set for any entailment of an ontology, i.e., the minimal set of axioms from the ontology responsible for the entailment. For debugging purposes, we can use it to either obtain the minimal set of axioms responsible for an unsatisfiable class in a consistent ontology, or responsible for an inconsistent ontology itself.

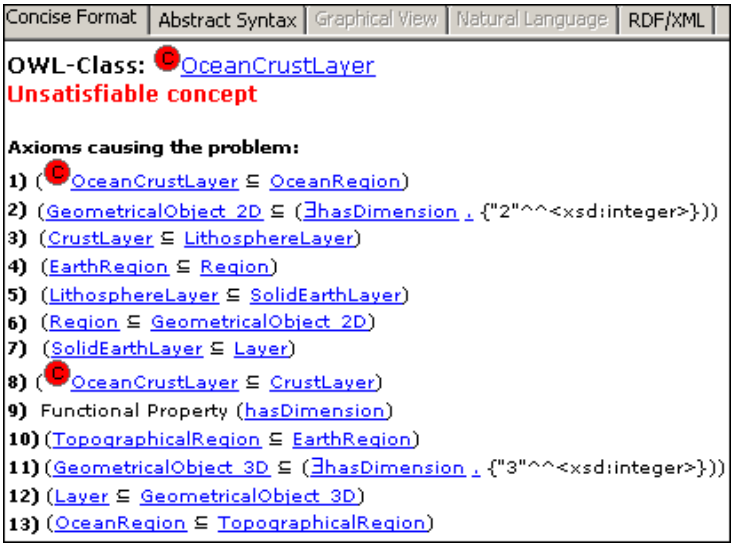

Figure 7.1: Displaying the Justification Axioms as a Single Unordered List

Figure 7.1 shows an example of this feature when invoked from within Swoop. The figure displays the thirteen axioms (of the only justification set) responsible for the unsatisfiability of the class OceanCrustLayer in the Sweet-JPL OWL Ontology.

From a debugging point of view, the advantage of this presentation is clear – among the (roughly) three thousand axioms present in the Sweet-JPL ontology, only the thirteen axioms that make the class unsatisfiable are displayed, and moreover, if any one of these axioms is removed from the ontology, the class OceanCrustLayer is guaranteed to turn satisfiable (since this is the only justification). However, on the downside, displaying the axioms as a single unordered list makes it difficult to see the interaction between the axioms and understand the real cause of the unsatisfiability.

To address this issue, we have made several enhancements in the presentation of the axioms in order to facilitate the understanding of the problem. These include:

- *Displaying Clash Information* (when using the tableau-based version of the Axiom Pinpointing service)
- *Improving Axiom Layout, i.e., ordering and indenting axioms*
- *Striking out Irrelevant Parts* (when using the service to obtain *precise justifications*)

Figure 7.2 shows an enhanced version of the earlier example.

Figure 7.2: Improved Presentation of the Justification

We now describe each of these enhancements in detail.

Displaying Clash Information

As noted in Chapter 2, there are many different ways for the axioms in an ontology to cause an inconsistency. But these different combinations boil down to some basic contradictions in the description of an individual. Tableaux algorithms apply transformation rules to individuals in the ontology until no more rules are applicable or an individual has a clash. The basic set of clashes in a tableaux algorithm are:

- *Atomic*: An individual belongs to a class and its complement.
- *Cardinality*: An individual has a cardinality restriction on a property that is violated.
- *Datatype*: A literal value violates the (global or local) range restrictions on a datatype property.

When using the tableau-based (hybrid) version of the Axiom Pinpointing service, it is easy to modify the internals of the tableau algorithm to expose and display the clash causing the inconsistency (as seen in Chapter 4).

One of the main challenges is to *usefully* present this clash information to the user since the normalization and decomposition of expressions (done by the reasoning algorithms) can obscure the error by getting away from the concepts actually used by the

modeler. Thus, we maintain the correspondence between the original asserted terms and their normalized versions, and display only the asserted information to the user.

Also, the clash may involve some individuals that were not explicitly present in the ontology, but generated by the reasoner in order to try to adhere to some constraint. Those generated individuals may not even exist (or be relevant) in all models. For example, if an individual has a owl : someValuesFrom restriction on a property, the reasoner would generate a new anonymous individual that is the value of that property. In this case, since these individuals do not have a name (URI) associated with them, we use paths of properties for identification (see Figure 7.3 for an example).

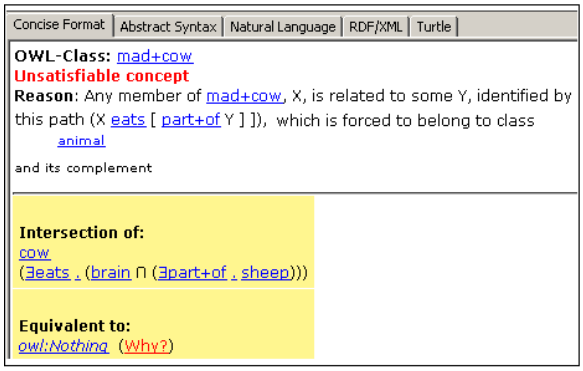

Figure 7.3: Displaying clash information using a property-path and variables to denote anonymous individuals. This example has been taken from the Mad-Cow ontology used in the OilEd [8] Tutorials.

Improving Axiom Layout

In order to improve the axiom layout, we use a recursive ordering strategy that starts with the unsatisfiable class definition axioms, and arranges axioms such that atleast one element (i.e., class, property or individual) in the signature of the right hand side (RHS) of the current axiom matches with the left hand side (LHS) of the next. The motivation here is to present a chain of reasoning by suitably aligning related axioms, i.e., axioms sharing elements in their signature. We discuss the pros and cons of this strategy with some sample cases.

Figure 7.4 shows three cases based on our axiom layout strategy. In each case, the ordering and indentation of the axioms helps leads the user down several reasoning chains, with the end-points of each chain being a direct pointer to the contradiction.

For example, in case (A), by following axioms $1 \rightarrow 2, 1 \rightarrow 3$ the user can see that an instance of class AIStudent is related to an instance of class ProfessorInHCIorAI by property hasAdvisor, whose `inverse` property advisorOf adds the type HCIStudent back to the first instance. Finally, the sole axiom 4 highlights the disjointness between the classes AIStudent, HCIStudent thus making the contradiction clear. Similarly, in case (B), the reasoning chain consisting of axioms $1..6$ indicates that an instance of class oxidation is related to an instance of class $\neg$regulation (via property involves), whereas the reasoning chain $[1..4, 4 \rightarrow 7]$ points to the contrary. Finally, in case (C), the axioms $1 \rightarrow 9$ indicate

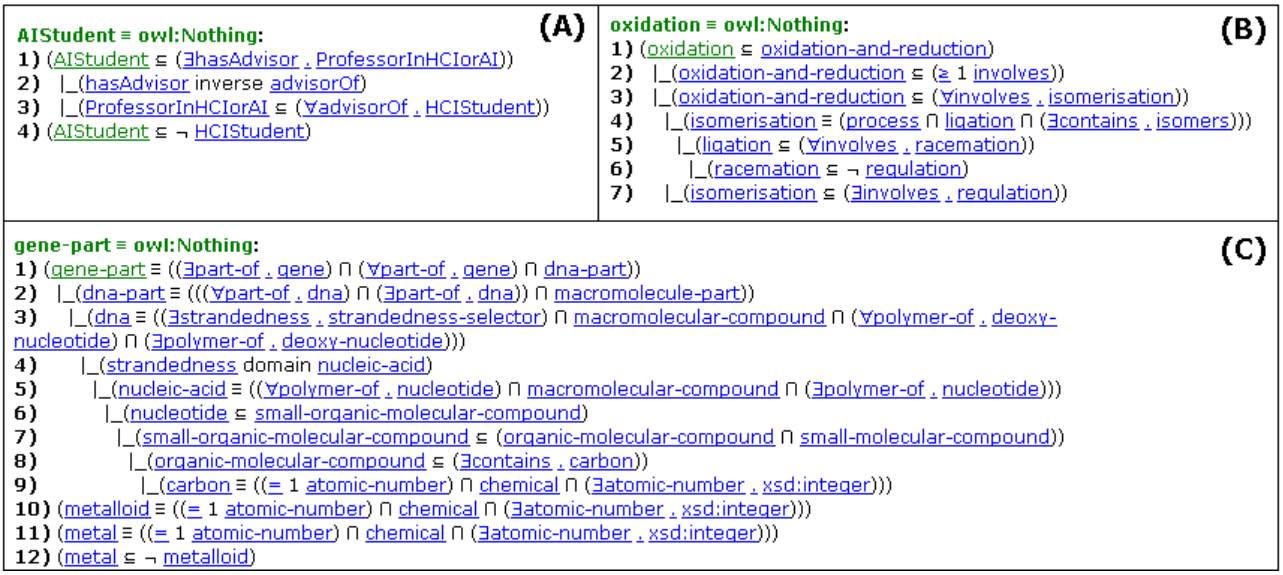

Figure 7.4: **Ordering and Indenting Justification Axioms**. Example (A) has been taken from the University OWL Ontology, whereas examples (B),(C) are from the Tambis Ontology.

that an instance of gene − part is related to an instance of carbon, whereas the last three unordered axioms $10..12$ point to the source of contradiction in carbon.

The reason this strategy works well in practice is because, typically, most of the axioms in an OWL ontology are subclass or equivalent axioms, which correspond to implications in FOL, i.e., $C \sqsubseteq D \mapsto \forall(x)C(x) \rightarrow D(x)$. Hence, a chain of sub-class relations forms a chain of implications, which is easy for the user to understand. Thus, this strategy relies on leading the user systematically from the base set of facts to the inferred ones until the source of the contradiction is reached.

However, there are cases when the interaction between the axioms is difficult to grasp even when the axioms are laid out as shown above. Figure 8.2 illustrates this point.

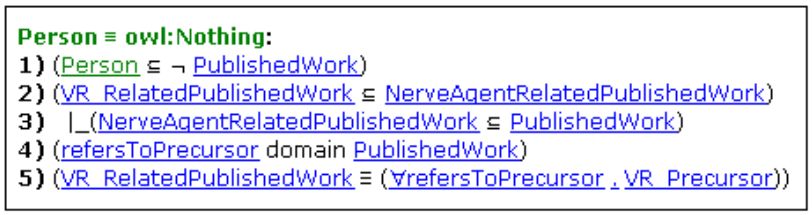

Figure 7.5: Justification example where ordering/indenting of axioms fails

In the figure, the cause of the unsatisfiability of Person is highly non-trivial. This is due to the interaction between axioms $2 - 5$ which makes the class PublishedWork equivalent to $\top$ (Top concept or in the OWL language, owl : Thing). Thus, axiom 1 which asserts the disjointness of Person and PublishedWork causes the former to become unsatisfiable. However, notice that it is difficult to get an indentation of the axioms that illustrates this form of interaction.

One way to alleviate the problem is to display critical intermediate inferences (e.g., the equivalence between PublishedWork and $\top$) to help understand the error better, as discussed in the future work section in Chapter 8.

Striking out Irrelevant Parts

When the Axiom Pinpointing Service is used to obtain *precise justifications*, we can directly strike out the parts of axioms that do not contribute to the unsatisfiability entailment. Figure 7.6 shows some examples that highlight this feature.

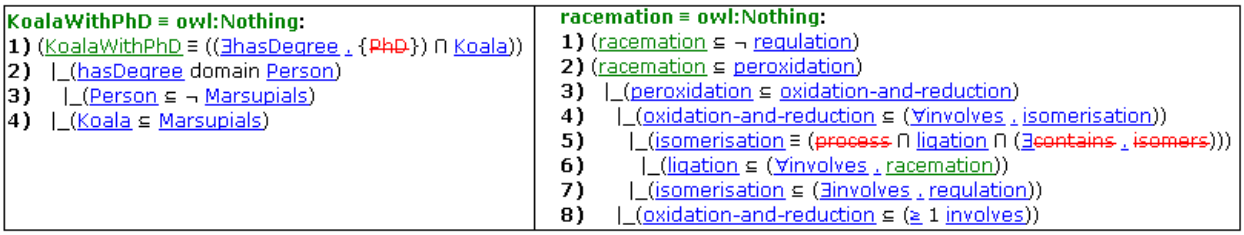

Figure 7.6: Striking out parts of axioms that are irrelevant to the entailment

Notice that keeping the original asserted axioms in view, with the irrelevant parts struck out, is done in order to maintain context. An alternative would be to hide the irrelevant information and display the smaller axioms (*sub-axioms*) directly, but this would require a correlation between the sub-axioms and the corresponding asserted axioms, which is an additional burden for the user.

7.1.2 Axiom Pinpointing: Performance Analysis

For the performance evaluation, we randomly selected 10 inferred entailments (including unsatisfiable class entailments if any) in each ontology present in Table 7.1. For each entailment, we first compared the performance of the base consistency checking algorithm versus the pure Black-box and Hybrid solutions for computing a *single justification*. We then evaluated the performance of the algorithm based on Reiter's Hitting Set Trees which computes *all* the justifications. The experiments have been performed on a Pentium Centrino 1.6GHz computer with 1.5GB memory, with 256MB (maximum) memory allotted to Java.

Computing a Single Justification

The second column of Table 7.2 shows the average runtime of the consistency test used to verify an entailment; the third and fourth columns depict the average times to find a *single* justification using the pure Black-box and the Hybrid solutions respectively. Timings for individual entailment tests in the ontologies are shown in Figure 7.7. Also shown are the average and maximum size of the justifications (in terms of axioms) in the last two columns.

There are two key points to note here:

OWL Ontology	Base Time	Single Just. (Black Box)	Single Just. (Hybrid)	Avg. Just. size	Max. Just. size
Chemical	0.285	0.68	0.295	6.9	9
Dolce	0.863	0.213	0.888	2	2
Economy	0.179	0.054	0.199	3.5	4
Galen	1.232	0.341	1.291	3.6	7
Sweet-JPL	0.29	0.187	0.301	4.2	13
Tambis	0.434	9.421	0.455	8.3	17
Transport	0.59	0.274	0.609	5.2	8
University	0.045	0.074	0.05	4.2	9
Wine	0.034	0.39	0.036	5.1	7

Table 7.2: Performance of Algorithms that find a *Single* Justification.

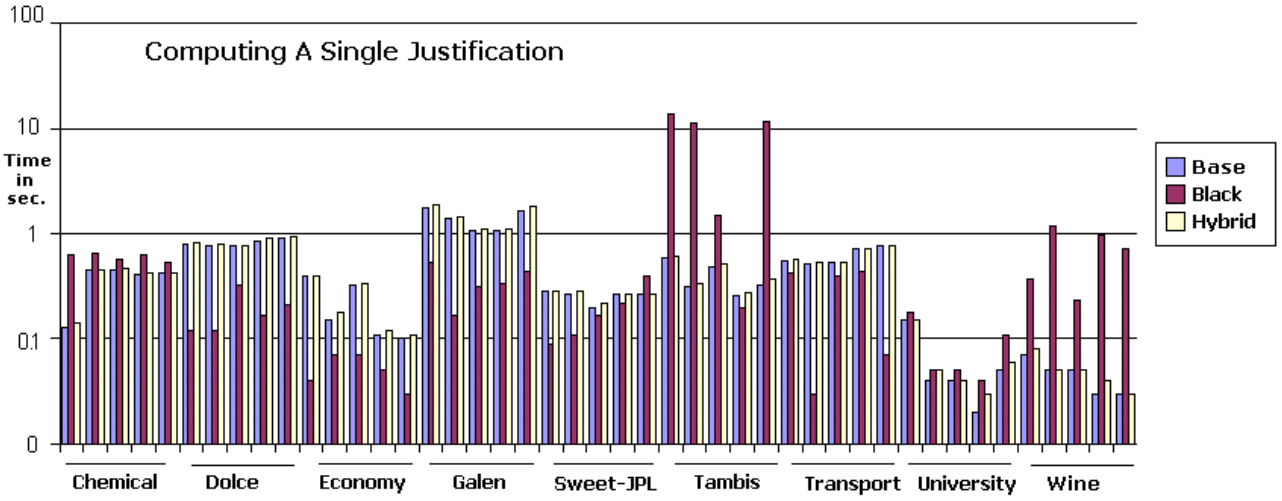

Figure 7.7: **Evaluating Algorithms to Compute a Single Justification**

1. The performance of the tableau-based hybrid algorithm to find a single justification is only marginally worse than the base consistency checking performance. This is not surprising, given that the 'axiom tracing' is tightly integrated into the standard tableau expansion process and the final stage of the algorithm which reduces the non-minimal axiom set (output by the tracing) to a minimal one by pruning out extraneous axioms includes very few such axioms – in all the tested cases, we found that the tracing output included atmost 5-10 irrelevant axioms. Thus, the final pruning, which involves reasoning over a very small fraction of the axioms (i.e., justification set + irrelevant axioms, which totals around 20-25 in all), introduces very little timing overhead.
2. The performance of the pure black-box solution to finding a single justification depends entirely on the locality of the problem, i.e., in a lot of cases, where the axioms responsible for the entailment are small in number (less than 10) and are closely related to the concerned entity definitions, the black-box algorithm performs well. However, for entailments in ontologies such as Chemical or Tambis, which are mainly caused by highly non-local conditions, the performance is degraded, as

the algorithm needs to span out to find relevant axioms sometimes including many irrelevant axioms which need to be pruned out subsequently.

One surprising result based on the timings shown in Figure 7.7 is that the black-box solution beats the hybrid solution (even surpassing the base consistency checking times) for entailments in an equal number of ontologies. The reason is that the input to the black-box algorithm is a small fragment, say $\mathcal{O}'$, of the original ontology $\mathcal{O}$ ($\mathcal{O}' << \mathcal{O}$) and thus the time taken by the reasoner to process $\mathcal{O}'$ is much smaller than $\mathcal{O}$ (e.g., since many General Concept Inclusion axioms in $\mathcal{O}$ are not considered initially). Thus, if the entailment is satisfied in $\mathcal{O}'$ and the algorithm does not need to expand $\mathcal{O}'$ any further, the algorithm terminates in lesser time as it never has to deal with the entire ontology.

However, the significance of the black-box timings must be taken into proper context – in order to determine whether a particular entailment holds in an ontology, we need to perform a consistency test over the entire ontology in the first place, and the timings do not reflect this. The advantage of the hybrid solution is that the justification finding can be done simultaneously (inline) during this consistency test used to verify the entailment.

Computing All Justifications

Table 7.3 depicts the average runtimes obtained when executing the Axiom Pinpointing service to compute all the Justifications for the unsatisfiable concepts in the above ontologies. Timings for individual entailment tests in the ontologies are shown in Figure 7.8.

Ontology	**Base Time** (s)	**All Justifications** (s)	**Avg. #Just.**	**Max. #Just.**
Chemical	0.285	1.431	2.8	6
Dolce	0.863	1.034	1	1
Economy	0.179	1.318	1.1	2
Galen	1.232	10.177	1.3	2
Sweet-JPL	0.29	2.541	1.2	2
Tambis	0.434	34.727	3.4	6
Transport	0.59	17.987	2.2	3
University	0.045	0.062	1	1
Wine	0.034	1.137	2.3	5

Table 7.3: Performance of Algorithm to find All Justifications.

We found that our algorithm performs well in practice, for two main reasons. First, the tableau-based hybrid algorithm for finding a single justification does not introduce a significant overhead w.r.t. the $\mathcal{SHOIN}$ satisfiability algorithm as seen in Table 7.2. Second, although the complexity of generating the Hitting Set Tree (HST) is exponential with the number of justifications, most of the tested entailments exhibited at most three or four justifications, with five to ten axioms each. For example, in the case of the Tambis OWL ontology, where each of the entailments in Figure 7.8 have at least 3 justifications, the algorithm terminated in less than a minute for most entailments.

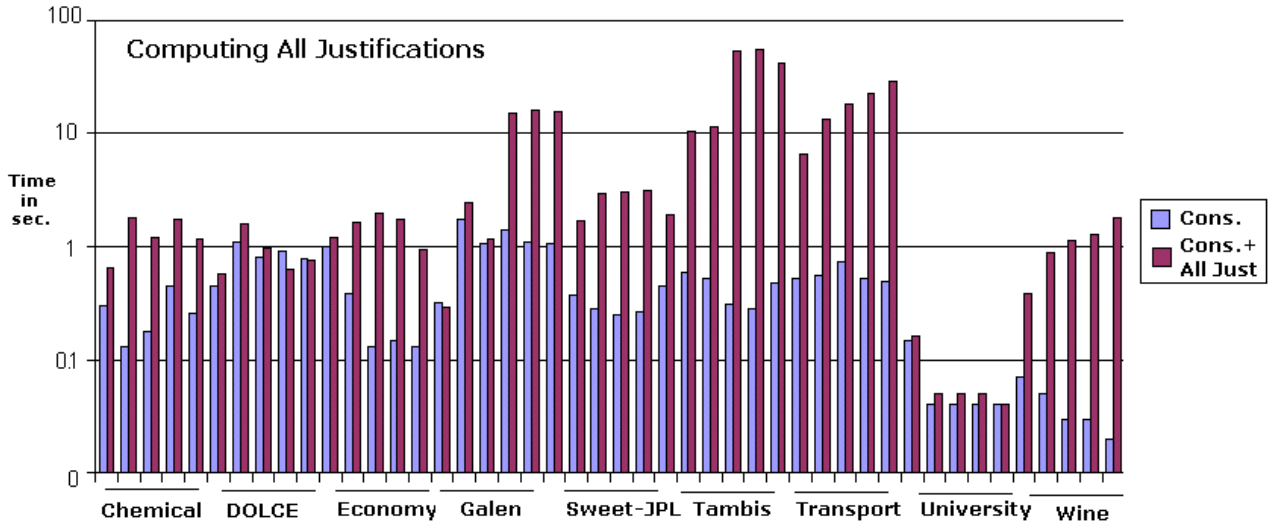

Figure 7.8: Evaluating Algorithms to Compute All Justifications. Time scale is logarithmic.

Computing Justifications using other DL reasoners

We have also tested the Black-box Axiom Pinpointing algorithms with two other DL reasoners besides Pellet - RACER Pro v1.9 [104] (the commercialized version of the RACER system) and KAON2 [78] (the ontology management infrastructure built at the University of Karlsruhe). Both, RACER Pro and KAON2, support the full OWL standard with the exception of nominals, but in addition, allow for qualified cardinality restrictions (hence supporting the logic $\mathcal{SHIQ}(\mathcal{D})$). While reasoning in RACER Pro is based on state-of-the-art tableaux algorithms (like Pellet), reasoning in KAON2 is implemented by reducing a SHIQ(D) knowledge base to a disjunctive datalog program [55].

For comparing the performance of the Axiom Pinpointing algorithms based on the three reasoners, we had to select OWL ontologies that could successfully be handled by all of them (e.g., excluding ontologies that made use of nominals). Figure 7.9 shows the results of the smaller evaluation on a few selected ontologies – Chemical, Economy, mini-Tambis, Transport, and University (minus nominals). As noted earlier, these ontologies have numerous unsatisfiable classes with many containing non-local errors (i.e., where all the erroneous axioms are not local to the concept definition).

Figure 7.9 depicts the time taken by the Black-box Axiom Pinpointing algorithm to find a single justification using RACER (Pro), KAON2 and Pellet respectively. The X-axis denotes individual entailments tests (randomly chosen) for each of the ontologies while the Y-axis denotes time in seconds.

The results show that RACER and Pellet both perform equally well and consistently outperform KAON2, which is expected, given that the former (besides having been around for a longer time) are based on highly optimized tableau algorithms making it better suited to handle class-based reasoning for expressive DLs that underlie the ontologies.

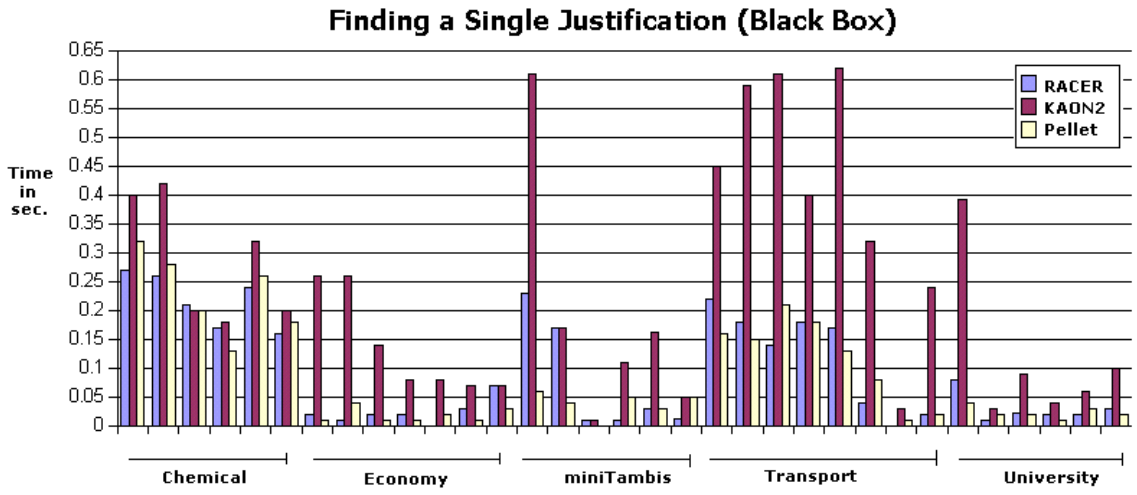

Figure 7.9: Comparison of DL reasoners to find Justifications

7.1.3 Implementing Root Error Pinpointing

The Root Error Pinpointing service described in Chapter 5 contains a set of algorithms for separating the root or critical errors in an ontology from the derived or dependent ones. Figure 7.10 shows an example of this service where the *Structural Analysis* algorithm is used to obtain a dependency table that highlights the parent dependencies of any partially derived unsatisfiable classes and emphasizes the roots at the top.

In addition to using the service output, we have made simple modifications in the UI to highlight error dependency. For example, all unsatisfiable named classes, and even class expressions, are marked with red icons whenever rendered — a useful pointer for identifying dependencies between inconsistencies. In Figure 7.11 (the Tambis ontology), note how simply looking at the class definition of $\mathsf{gene} - \mathsf{part}$ makes the reason for the inconsistency apparent: it is a subclass of the inconsistent class $\mathsf{dna} - \mathsf{part}$ and the inconsistent class expression $\exists \mathsf{partof}.\mathsf{gene}$. The hypertextual navigation feature of Swoop allows the user to follow these dependencies easily, and reach the root cause of the inconsistency, e.g., the class which is independently inconsistent in its definition (i.e., no red icons in its definition). In this manner, the UI guides the user in locating and understanding bugs in the ontology by narrowing them down to their exact source.

7.1.4 Root/Derived Performance Analysis

We have tested this service on various OWL ontologies that have a large number of unsatisfiable concepts and found it to be very useful in narrowing down the error space substantially, with its performance being reasonably fast.

Table 7.4 shows the summary of our evaluation of this service. The ontologies Tambis, DICE-A (Anonymized version of the DICE terminology), Chemical and Terrorism contain real modeling errors, while Transportation and Economy ontologies have unsatisfiable concepts introduced in them using the Strong Disjointness Assumption (SDA) as noted in [92], i.e., by adding disjoint statements between siblings.

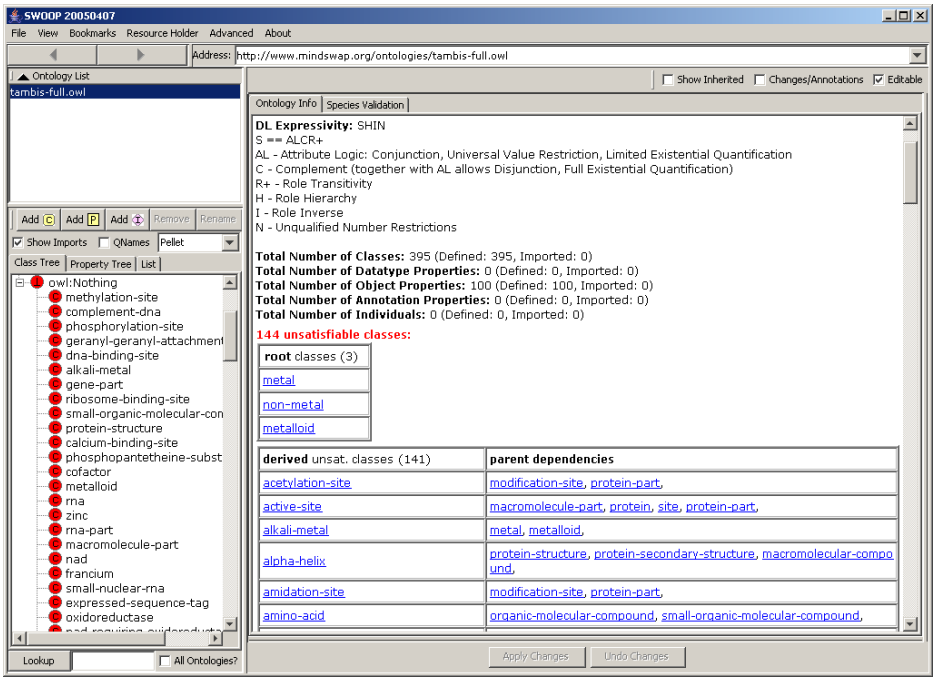

Figure 7.10: Root/Derived Debugging in Tambis using Structural Analysis

OWL Ontology	Unsat. Concepts	Root / Derived	Step1-time / Step2-time
Tambis	144	3/141	0.033s / 1.893s
DICE-A	76	5/71	0.01s / 7s
Transportation	62	5/57	0.02s / 2.8s
Economy	51	34/17	0.01s / 2.5s
Chemical	37	2/35	0.01s / 0.14s
Terrorism	14	5/9	0s / 0.951s

Table 7.4: Evaluation of the Root/Derived Debugging Service

As can be seen, the number of root concepts found in each case is a fraction of the total number of unsatisfiable concepts (with the exception of the Economy ontology where it is still a reasonable reduction). The last column displays separate timings (in seconds) for the two steps in the service algorithm, i.e., structural tracing and inferred dependency detection as described in Chapter 5.The results clearly show that the service plays a key role in pruning errors quickly.

7.1.5 Ontology Repair

The Ontology Repair service described in Chapter 6 is used to generate repair plans to fix errors in an ontology based on various metrics for ranking erroneous axioms.

The key design goal for its UI in Swoop is to provide a flexible, interactive framework for repairing the ontology by allowing the user to analyze erroneous axioms, weigh axiom ranks as desired, explore different repair solutions by generating plans on the fly, preview change effects before executing the plan and compare different repair alternatives. Moreover, the tool also suggests axiom edits where possible.

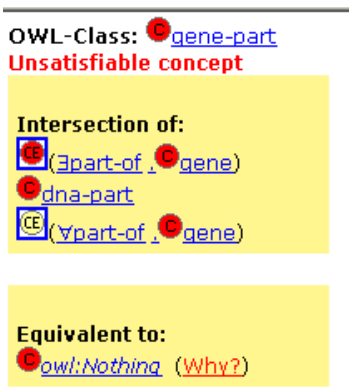

Figure 7.11: The class `gene-part` is unsatisfiable on two counts: its defined as an intersection of an unsatisfiable class (`dna-part`) and an unsatisfiable class expression (∃`partof.gene`), both highlighted using red tinted icons.

Figure 7.12 is a screenshot of the Swoop repair plugin when used to debug the University OWL ontology. As can be seen, the top segment of the repair frame displays a list of unsatisfiable concepts in the ontology, with the *root* classes marked. The adjacent pane renders the axioms responsible for making the concepts selected in the list unsatisfiable. There are two view modes for this pane – the one shown in Figure 7.12 displays the erroneous axioms for each unsatisfiable class in separate tables with axioms indented (as described earlier), and common axioms responsible for causing multiple errors highlighted as shown. The other view displays all erroneous axioms globally, i.e., in a single list as shown in Figure 7.13.

The tables in both views display for each axiom, its arity, impact and usage, computed as described in Chapter 6. The values for these parameters are hyperlinked, clicking on which pops up a pane which displays more details about the parameter (not shown in the figure). For example, clicking on a value for the arity displays the concepts whose justification the axiom falls in, while clicking on a value for the impact displays entailments that are dependent on this axiom.

To see how the impact analysis is useful, see Figure 7.14. The figure displays the entailments that are dependent on the axiom $\mathsf{Lecturer} \equiv \mathsf{hasTenure.false} \sqcap \mathsf{TeachingFaculty}$. In this case, the tool has displayed *useful* entailments related to unsatisfiable classes (highlighted in red), as described in Chapter 6. The user can see the reason for each of these entailments by clicking on the *Why?* link, e.g., the two axioms which cause the entailment $\mathsf{Lecturer} \equiv \mathsf{AssistantProfessor}$, and use this information to reach a suitable plan as discussed below.

Also, clicking on the table headers re-sorts the results based on the parameter selected. The total rank for each axiom, displayed in the last column of the table, is the weighted sum of the parameter values, with the weights (and thus ranks) being easily reconfigurable by the user. For example, users interested in generating minimal impact plans can assign a higher weight to the impact parameter, while users interested in smaller sized plans can weigh arity higher. The range of the weights is from -1.0 to 1.0.

As discussed in Chapter 6, we provide three different granularities for the repair

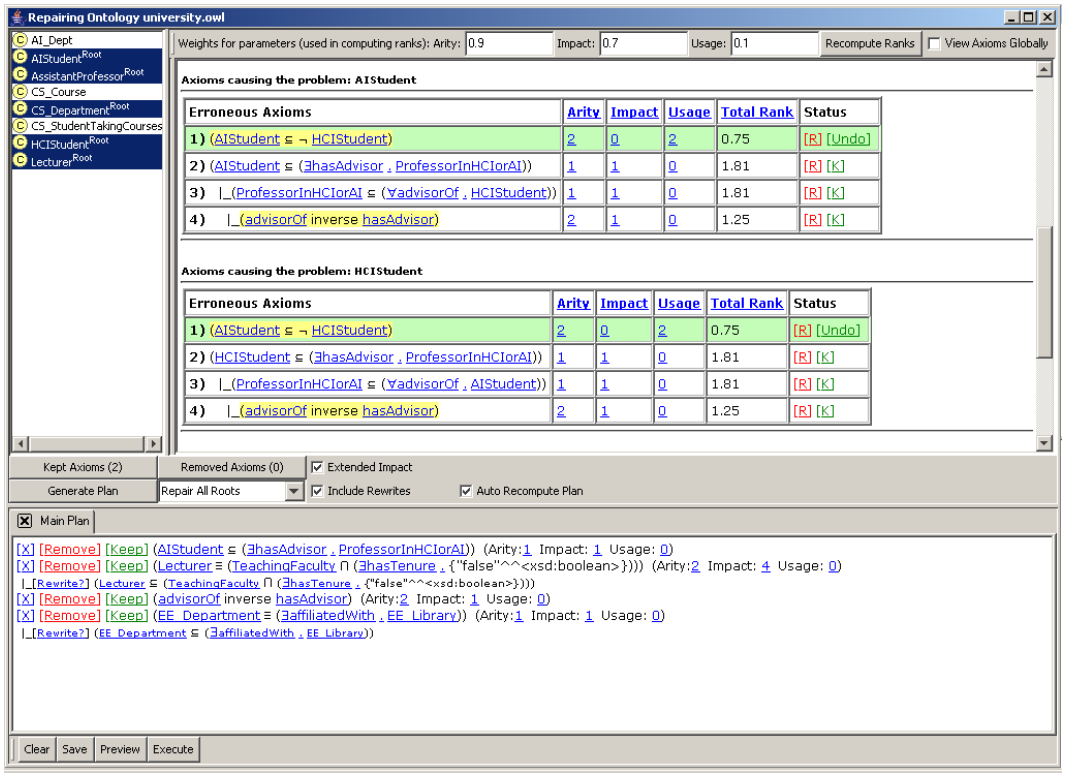

Figure 7.12: **Interactive Repair in Swoop:** Generating a repair plan to remove all unsatisfiable concepts in the University OWL Ontology

process, i.e., the ability to fix a particular set of unsatisfiable concepts; all the *roots* only; or all the unsatisfiable classes directly in one go. For example, in Figure 7.12, the user has asked the tool to generate a plan to repair all the roots.

For a repair tool to be effective, it should support the easy customization of the plan to suit the user's needs. In the simple case, the user can either choose to *keep* a particular axiom in the ontology, or forcibly *remove* a particular one. These user-enforced changes are automatically reflected in the plans. In Figure 7.12, the user has chosen to *keep* the disjoint axioms $\mathsf{AIStudent} \sqsubseteq \neg\mathsf{HCIStudent}$, and $\mathsf{Lecturer} \sqsubseteq \neg\mathsf{AssistantProfessor}$ in the ontology (highlighted in green in the Table). In the advanced case, the user can choose to keep or remove a particular entailment of the ontology, e.g., a particular subclass relation. The tool then takes these desired and undesired entailments into account when generating a plan.

Finally, axiom rewrites suggested by the tool (based on the techniques described in Chapter 6) can be (optionally) included in the plan as well. In the figure, the tool has suggested weakening the two equivalence axioms to subclass relations, which removes the contradictions in the unsatisfiable classes, but preserves the semantics as much as possible. Obviously, the user can directly edit erroneous axioms if desired.

The repair plan can be saved, compared with other plans and executed, after which the ontology changes (which are part of the plan) are logged in Swoop. These changes can be serialized and shared among ontology users (as shown in Chapter 8).

Erroneous Axioms	Arity	Impact	Usage	Total Rank	Status
(AIStudent ⊑ ¬ HCIStudent)	2	0	2	0.75	[R] [Undo]
(Lecturer ⊑ ¬ AssistantProfessor)	2	0	3	0.85	[R] [Undo]
(advisorOf inverse hasAdvisor)	2	1	0	1.25	[R] [K]
(HCIStudent ⊑ (∃hasAdvisor . ProfessorInHCIorAI))	1	1	0	1.81	[R] [K]
(CS_Department ⊑ (∃affiliatedWith . CS_Library))	1	1	0	1.81	[R] [K]
(ProfessorInHCIorAI ⊑ (∀advisorOf . AIStudent))	1	1	0	1.81	[R] [K]
Transitive(affiliatedWith)	1	1	0	1.81	[R] [K]
(CS_Library ⊑ (∃affiliatedWith . EE_Library))	1	1	0	1.81	[R] [K]
(ProfessorInHCIorAI ⊑ (∀advisorOf . HCIStudent))	1	1	0	1.81	[R] [K]
(EE_Department ≡ (∃affiliatedWith . EE_Library))	1	1	0	1.81	[R] [K]
(AIStudent ⊑ (∃hasAdvisor . ProfessorInHCIorAI))	1	1	0	1.81	[R] [K]
(Lecturer ≡ (TeachingFaculty ⊓ (∃hasTenure . {"false"^^<xsd:boolean>})))	2	4	0	3.35	[R] [K]
(AssistantProfessor ≡ (TeachingFaculty ⊓ (∃hasTenure . {"false"^^<xsd:boolean>})))	2	4	0	3.35	[R] [K]
(EE_Department ⊑ ¬ CS_Department)	1	10	4	8.51	[R] [K]

Figure 7.13: Analyzing Erroneous Axioms in a Single (Global) View

1. (AssistantProfessor ⊑ Person) (Why?)
2. (AssistantProfessor ⊑ TeachingFaculty) (Why?)
3. (AssistantProfessor ⊑ Faculty) (Why?)
4. (Lecturer ≡ AssistantProfessor) (Why?)

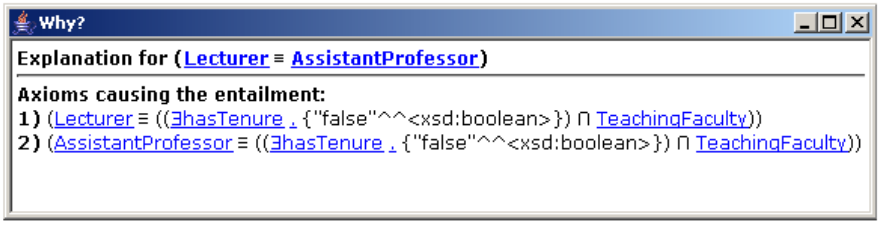

Figure 7.14: Displaying the Impact of Erroneous Axiom Removal

7.2 Usability Studies

In order to determine the practical use and efficiency of the debugging and repair features implemented in Swoop/Pellet, we conducted a small usability-study as follows:

1. We selected twelve subjects in all having at least 9 months of experience with OWL and with an understanding of description-logic reasoning that varied greatly (novices to experts). Most of the subjects were undergraduate and graduate students at the University of Maryland in the Computer Science dept.
2. Each subject received a 20-30 minute orientation that covered:
   - an overview of the semantic errors found in OWL ontologies (using examples of unsatisfiable classes)
   - a brief tutorial of Swoop, demonstrating its key browsing, editing and search features
   - a detailed walkthrough of the debugging and repair support in Swoop using a set of toy ontologies

We then performed two separate studies, the first testing the debugging services, i.e. Axiom Pinpointing and Root Error Pinpointing; and the second evaluating the Ontology Repair service.

7.2.1 Evaluating Debugging

In this case, the twelve subjects were *randomly* divided into 4 groups of three subjects each as follows:
Group 1: Subjects in this group received *no debugging support* at all, i.e., only a list of unsatisfiable classes in the ontology was displayed by the reasoner
Group 2: Subjects in this group could only use the *Axiom Pinpointing* service
Group 3: Subjects in this group could only use the *Root Error Pinpointing* service
Group 4: Subjects in this group could use *both*, the Axiom Pinpointing and the Root Error Pinpointing services

Having formed the groups, each subject was given three erroneous ontologies – University.owl, SweetJPL.owl and miniTambis.owl (in random order), any of which the subject had not seen before. The subject was asked to debug the ontologies in Swoop (*independently*) using only the features assigned to the group the subject belonged to. The following guidelines were observed during the debugging process:

- The subject was given a maximum of 30 minutes to debug an ontology. He/she was free to stop the debugging process at any time.
- While debugging any unsatisfiable class, the subject was asked to write down a brief explanation of the contradiction for that class (in his/her own words) based on the understanding of the problem. In addition, the subject was asked to suggest a likely fix for the problem where possible
- The tool automatically counted the number of entity definitions viewed, and the changes made to the ontology during the entire debugging process, both of which we considered as key sub-tasks

Having obtained the times taken by a subject for debugging each of the three ontologies, we took the average of the times (for the group) in order to nullify the expertise and skill factor of the subject (note that the subjects were randomly assigned to the groups as mentioned earlier).

Finally, after working on all three ontologies, the subject was handed a questionnaire to elicit feedback on the entire debugging experience using Swoop

Key properties of the ontologies used in the study were:

Ontology	Total Classes	Unsat. Classes	Root/Derived
1. University.owl	28	8	5/3
2. SweetJPL.owl	1537	1	1/-
3. miniTambis.owl	183	30	5/25

Our hypothesis was as follows:

1. *The information provided by the Axiom Pinpointing service is better than no support for all the erroneous ontologies, i.e., the subject will take significantly less time*

to understand and fix the errors correctly using the service. The reason for this is that the information would help pinpoint and illustrate the source of the contradiction for the unsatisfiable class.

2. *For a relatively small no. of unsatisfiable classes (i.e., ontologies 1 and 2), the Axiom Pinpointing service information will outperform both, the Root Error Pinpointing service and the no support, and perform not too worse than the full-debug support.* The reason for this is that the subject could potentially distinguish the root from the derived classes by manually inspecting the justification axioms for each class, thus reducing the impact of automatically identifying them.
3. *For a large no. of unsatisfiable classes with different roots (i.e., ontology 3) the Root Error Pinpointing service support will match the performance of the Axiom Pinpointing information, and additionally, the full-debug support will be clearly better than either of the two.* The reason for this is that manually discovering the root/derived classes would often be hard and time-consuming in such cases, and the dependency detection technique would help narrow down the problem space tremendously.

The results of the usability study are summarized in the graph in Figure 7.15. The graph displays the average time taken (in mins) per group to debug all the errors for each of the three ontologies given (Note: 'F' represents a Failure to debug the error).

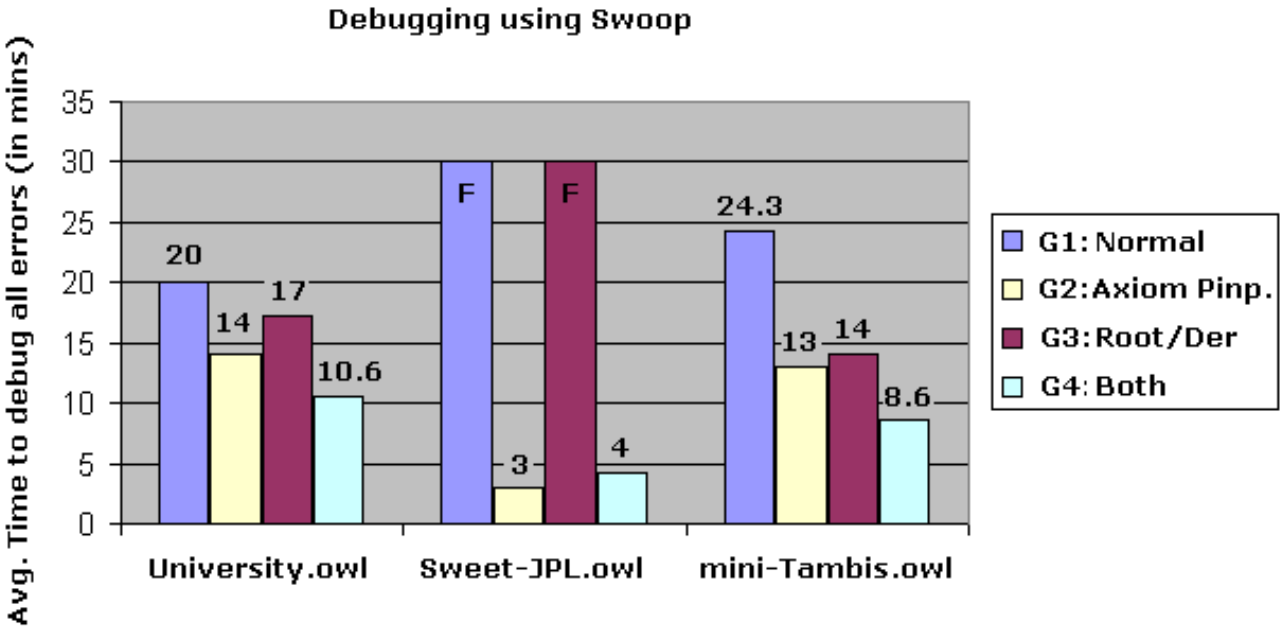

Figure 7.15: Results of the Debugging Usability Study

As seen from the graph, the statistical results obtained are in agreement with hypothesis (1), i.e., a 2-tailed T-test shows that debugging with clash/SOS is significantly better than debugging without it for $p \approx 0.01$. While the timings for the ontologies are in

agreement with hypothesis (2) and (3), given the small size of the study, a measure of the statistical results was not significant for verifying those hypothesis. We plan to conduct a more extensive evaluation to fully justify them.

For University.owl, all subjects were able to identify the erroneous axiom(s) for each of the unsatisfiable classes within the time period given, however, only 1 subject in normal/root-derived (black box) mode was able to understand the cause of the problem, whereas, 2/3 using the Axiom Pinpointing and 3/3 using the full-debug mode were able to understand and explain the problem correctly. Also, the time taken to fix all the errors using the full-debug mode was approx. half of that taken using the normal-mode.

In the case of SweetJPL.owl, without justification axioms no subject was able to understand the cause of the error due to the highly non-local interactions in the large ontology, whereas, using the axioms, each subject took under 5 minutes to understand and fix the problem correctly.

Interestingly, the results given only the Root Error Pinpointing service performed nearly as well as the Axiom Pinpointing in the case of miniTambis.owl since subjects found it easier to debug the roots identified by the former service than to manually discover them using the latter, due to the large number of unsatisfiable classes. Also, for this ontology, the subjects in the normal mode fixed only 2/3 roots in the time period given, i.e, they could not fully complete the debugging.

We learnt some useful lessons based on our observations of the debugging process and the feedback given by the subjects:

- **For Group 1 – no-debug mode:**
  - 3/3 subjects rated the hypertextual navigation (with back/next history) as the most useful feature for understanding relationships and causes of errors in the ontology
  - 2/3 subjects found ontology changes immensely useful to identify erroneous axioms by using a trial-and-error strategy
  - The *Show References* search feature was never used by any of the subjects. Based on their comments, it seemed that they were unclear about its use and significance. Interestingly, a subject in Group 3 found this feature very helpful, implying that the feature either supports a different debugging style (to that of the authors in this mode) or requires better presentation.
- **For Group 2 – Axiom Pinpointing:**
  - 3/3 subjects rated the justification axioms as the most useful feature
  - 2/3 subjects felt that a proof-style layout of the justification axioms with intermediate inferences shown as well would help explain the problem better.
  - Overall, 6 subjects were exposed to the Axiom Pinpointing service (3 from this group and 3 using the full-debug mode), and they were divided on the significance of the clash information. While half the subjects used the clash information to pinpoint relevant components of the justification axioms, the other half found the information poorly presented and redundant given the justification axioms, pointing us to a definite area of improvement.

- **For Group 3 – Root Error Pinpointing:**
  - 1/3 subject used the *Show-References* feature extensively to aid debugging, especially for mini-Tambis.owl, where discovering a commonly-used property restriction helped understand the source of the contradiction for a set of unsatisfiable classes
  - 1/3 subject felt that the Class-Expression (CE) support needed to be made more effective by allowing arbitrary queries on CEs
  - 2/3 subjects suggested displaying the number of derived dependencies that arise from each root to highlight the more significant roots
- **For Group 4 – full-debug mode:**
  - 3/3 subjects felt that it was the *combination* of the clash/SOS presentation and the root/derived identification and not one specific feature that was useful to debug all errors in the ontology

Overall, the response of the subjects in the study was very encouraging. Many relative newcomers to OWL and description-logic were impressed by the fact that they were able to correctly fix all the errors in ontologies which they had not seen before within the specified time period. Experts in the field who had experience in OWL ontology modeling and manual debugging were surprised at how easy the task of debugging was made for them.

7.2.2 Evaluating Repair

For this study, we selected two OWL Ontologies – University.owl and miniTambis.owl and asked each subject to debug all the unsatisfiable classes in a particular ontology using the Axiom/Root Error Pinpointing services, and in the other ontology using the Ontology Repair service. We had introduced new errors in these ontologies to make them different from the earlier study, however, the errors were realistic based on commonly observed patterns and misconceptions (e.g., errors enumerated in [87]).

The subjects were randomly assigned to the two cases, but the overall distribution was equally proportional in that given a particular ontology, an equal number of subjects (six) debugged it with and without using the Ontology Repair service. At the end of the study, our goal was to compare the performance improvement, if any, of using the Ontology Repair service over the other two debugging services, which were shown to be useful in the previous study.

The subjects were given a maximum of 45 minutes to debug the entire ontology, and as in the previous case, the tool recorded the use of the various repair features, e.g., granularity of the repair plan selected, ranking metrics viewed, number of rewrites included etc.

Before we proceed to our hypothesis, we discuss important points related to this study that were factored into its design. The first is that the subjects did not build the ontology themselves and hence did not have prior knowledge of the modeling intent. Secondly, the subjects did not possess domain knowledge related to the mini-Tambis (medical) ontology. Thirdly, in cases where there is more than one reasonable solution, determining

a 'correct' solution is subjective, and we took this fact into account when evaluating the results.

A key point to note is that we have basically divided the subjects into two groups – $\mathbf{G1}$: Axiom and Root/Error Pinpointing services, and $\mathbf{G2}$: Ontology Repair service, and subjects in $\mathbf{G2}$ have access to all the features available in $\mathbf{G1}$ (since the Ontology Repair UI displays the justification axioms responsible for an unsatisfiable class, and differentiates between the root/derived unsatisfiable classes), with the addition of the ranking metrics and the plan generation/customization options present in $\mathbf{G2}$. Thus, from a debugging and repair point of view, subjects in both groups were in a position to understand the error and determine the critical unsatisfiable classes, however, the difference was that in $\mathbf{G1}$, they had to manually repair one unsatisfiable class at a time, select appropriate axioms to remove and determine the impact of their solution, whereas in $\mathbf{G2}$, they had the necessary tool support to automate these tasks.

Based on these factors, our hypothesis was as follows: *The Ontology Repair service is more "effective" than a combination of both, the Axiom Pinpointing and the Root Error Pinpointing service, for repairing all the unsatisfiable concepts in an ontology, in that the quality of the repair solutions in both cases is comparable, but the time taken to arrive at a solution in the former case is significantly smaller than in the latter case.*

The results of the timings are displayed below. All times in the table are in minutes. As can be seen, the time taken to arrive at a solution in the repair case ($\mathbf{G2}$) was between 3-8 times less than in debugging case ($\mathbf{G1}$). A standard 2-tailed T-test on the data collected for the University / miniTambis ontologies indicated that $\mathbf{G2}$ is significantly faster than $\mathbf{G1}$ with $p < 0.05$ and $p < 0.001$ respectively.

University Debug ($\mathbf{G1}$) — Repair ($\mathbf{G2}$)	**miniTambis** Debug ($\mathbf{G1}$) — Repair ($\mathbf{G2}$)
8 — 2	11 — 2
9 — 2	12 — 2
9 — 3	15 — 2
12 — 4	16 — 4
14 — 5	17 — 5
33 — 6	22 — 6

We found that in both groups, the quality of the repair solution was quite similar, with the subjects in $\mathbf{G2}$ performing marginally better. For example, in the University ontology, all the subjects in both groups were able to correctly ensure that the concepts Lecturer, AssistantProfessor did not become equivalent. The rewrites suggested by the repair service ($\mathbf{G2}$) were useful in this regard as subjects always opted for the weakening of the concept definitions. However, for the slightly more difficult problem related to the concepts AIStudent, HCIStudent, subjects in $\mathbf{G2}$ were able to arrive at the correct solution that avoided these two concepts from becoming equivalent by using the impact analysis.

The miniTambis ontology posed a different challenge. Since subjects found this domain (medical) more foreign to that of the University ontology, the quality of the solutions in both groups were below par.

In addition, we learnt some valuable lessons based on our observations of the repair tool usage and the feedback provided by the subjects:

- All the subjects reached the desired repair solution within 0-3 changes from the initial plan suggested by the tool. This implies that the quality of the solutions based on the default ranking metrics/weights was reasonable.
- All the subjects appreciated the quality of the axiom rewrites suggested by the tool, and in every case that a rewrite was suggested, it was incorporated in the final solution.
- All the subjects preferred the 'local' axiom table view to the 'global' view, in order to understand the interaction among the axioms and identify common erroneous axioms.
- Only 3/12 subjects opted to repair all the unsatisfiable classes in one go, while the remaining chose to repair the unsatisfiable classes iteratively by focusing on the current roots.
- Only 3/12 subjects changed the default weights for the (axiom) ranking metrics suggested by the tool. The only change was weighing 'arity' less and/or 'usage' more. 'Impact' was consistently weighted high by all the subjects.
- Only 2/12 subjects found 'usage' as a significant metric and took it into account when arriving at a repair solution. This points to an area of improvement.

Chapter 8
Open Issues and Future Work

In this chapter, we enumerate the limitations and open issues of our OWL debugging services and outlines areas for future work.

8.1 Enhancing Debugging and Repair Services

8.1.1 Improving Algorithmic Performance

In Chapter 4, we have described a Black-box (reasoner independent) algorithm to find a justification for an arbitrary entailment of an OWL-DL ontology, and then developed a pre-processing Glass-box optimization procedure (tableau-tracing), which reduces the size of the input to the Black-box algorithm thereby providing a big performance improvement. However, ideally, we would like to have a purely Glass-box solution to finding a justification (minimal axiom set) since it would eliminate the additional step of pruning axioms, which may be time consuming in some cases (when there are a large number of role successors due to cardinality or existential restrictions) .

The challenge here is obtaining *minimality* of the axiom sets when building the tableau (completion graph) for expressive DL KBs. One of the main problems arises due to the presence of cardinality restrictions, and in particular, the $\leq n.R$ rule – when a node in the completion graph built by the tableau reasoner contains a concept $\leq n.R$ and if there exists more than 'n' R-successors of that node, then the $\leq n.R$ rule gets fired which arbitrarily merges any two successor nodes recursively till it is no longer applicable.

For example, consider an ontology with the following axioms:

1: $A \sqsubseteq \exists R.B$	2: $A \sqsubseteq \exists R.(C \sqcap \neg B)$	3: $A \sqsubseteq \exists R.(\neg C \sqcap \neg B)$
4: $A \sqsubseteq \exists R.C$	5: $A \sqsubseteq \leq 2.R$	

In the ontology above, the concept A is unsatisfiable and it's justification set is $\{1, 2, 3, 5\}$.

Consider the completion graph for the concept A shown in Figure 8.1, in which the reasoner has processed axioms $1 - 4$ and generated four R-successor nodes of the root node x (that represents concept A).

Now, when the algorithm unfolds axiom 5, the concept $\leq 2.R$ is added to $\mathcal{L}(x)$ and the presence of more than two R-successors of x causes the $\leq n.R$ rule to be applied recursively. We find that a clash occurs in the completion graph eventually irrespective of the choice of nodes to merge. This clash occurs because either both $B, \neg B$ or $C, \neg C$ are present in the label of the same successor node of x.

In this case, the key question is *which* successors should be considered responsible for the merge operation since there are greater than two successors of node x, and the restriction demands at most two. We certainly cannot consider all the successors since that would cause axiom 4 to be included in the trace of the clash, which is incorrect. On the other hand, if we consider any three successors arbitrarily (which is when the $\leq 2.R$

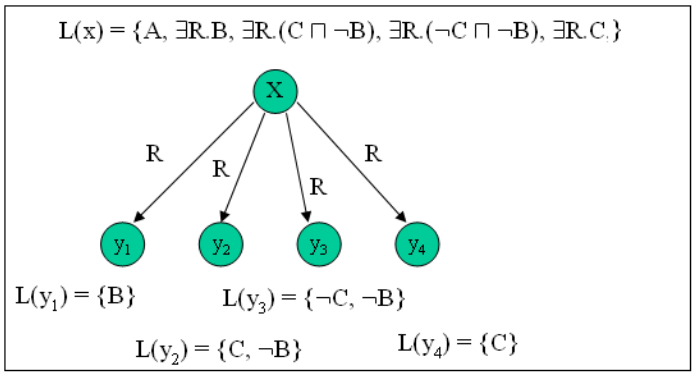

Figure 8.1: Finding minimal justification hard due to node merges

rule is applicable), we need to ensure that all combinations of merges involving those three successors results in a closure.

The matter is further complicated if the existential restriction in axiom 2 is replaced by a universal: $A \sqsubseteq \forall R(C \sqcap \neg B)$. In this case, the justification for the unsatisfiability of A reduces to the axiom set $\{1, 2\}$ – the clash occurs in node y_1 irrespective of the merge operation. Hence, an additional issue is identifying whether a clash *depends* on the merge or not. This can be done by introducing choice points in the trace of an event, and using this choice record to determine if an event could have occurred independent of the choice. We are currently working on an algorithm that takes into account issues such as this.

8.1.2 Improving Output Explanation

We are exploring the possibility of inserting intermediate steps (inferences) in the output of the Axiom Pinpointing service to help make the explanation easier to follow. Consider an example taken from the Chemical ontology shown in Figure 8.2 (the example was seen previously in Chapter 7):

Person ≡ owl:Nothing:
1) (Person ⊑ ¬ PublishedWork)
2) (VR_RelatedPublishedWork ⊑ NerveAgentRelatedPublishedWork)
3) |_(NerveAgentRelatedPublishedWork ⊑ PublishedWork)
4) (refersToPrecursor domain PublishedWork)
5) (VR_RelatedPublishedWork ≡ (∀refersToPrecursor . VR_Precursor))

Figure 8.2: Axiom Pinpointing example where cause of unsatisfiability is hard to understand by looking at the asserted axioms

In this case, the axioms $2 - 5$ cause the class PublishedWork to be equivalent to $\top$ (owl : Thing), which in turn renders the class Person unsatisfiable as it is disjoint with PublishedWork. Here, displaying the inference PublishedWork $\equiv \top$ that arises from axioms $2 - 5$ would help make the cause of the error clearer.

In general, determining which intermediate inference is critical to understanding

the error is not easy. There are two problems here: Firstly, the inference may be rather non-trivial as is the case above, i.e., simply looking at the axioms, it is difficult to tell that PublishedWork $\equiv \top$. At best, one could flag suspicious entailments such as this (atomic concept being equivalent to $\top$), however, good heuristics need to be developed to expose 'key' problematic inferences. Secondly, numerous trivial inferences can follow from the output axioms and one needs to be careful about cluttering the output with too much additional information, e.g., in the above case, axioms $\{2, 3\}$ entail

VR_RelatedPublishedWork $\sqsubseteq$ PublishedWork

though this simple subsumption may be avoided in the output.

8.1.3 Testing and Evaluating Repair

One of the known limitations of the conducted user study described in Chapter 7 was that the subjects did not author the ontologies themselves, and lacked domain knowledge, which adversely affected the quality of the repair solutions. A more thorough case study – that would involve placing the service in a real world ontology engineering/application context and having domain and ontology modeling experts use it over a period of time – would help us gauge the efficiency of this service better. Also, the notion of maintaining a library of error patterns as discussed in Chapter 6 would be more applicable in the context of this longer study.

We also discuss an interesting extension to the axiom rewrite module in the repair service. Currently, axiom rewrites are determined by inspecting the erroneous parts of axioms (obtained using the Axiom Pinpointing service), and using heuristics based on commonly occurring error patterns. We can also suggest rewrites that are in keeping with the update semantics proposed in [63].

We describe the idea using an example from the Koala ontology in which the concept Koala is unsatisfiable due to the following axioms:

Koala $\sqsubseteq \exists$isHardWorking.$false$
$domain$(isHardWorking, Person)
Koala $\sqsubseteq$ Marsupials
Marsupials $\sqsubseteq \neg$Person

An instance of Koala is inferred to belong to the class Person and Marsupials, which is disjoint with Person, hence the contradiction. In this case, one likely update that preserves the semantics as much as possible while getting rid of the unsatisfiability of Koala involves introducing a disjunction in axiom 2 as follows:

$domain$(isHardWorking, Person $\sqcup$ Marsupials).

This notion of introducing disjunctions in axioms to allow for additional models and get rid of the contradiction is discussed in [63]. Identifying meaningful updates on these lines in expressive DLs such as $\mathcal{SHOIN}$ (OWL-DL) is a hard and unresolved problem.

8.1.4 Debugging Non-Subsumptions

So far, we have presented techniques for diagnosing semantic defects, which are also directly applicable for any unintended entailments (not just logical inconsistencies).

As future work, we are looking at the problem of debugging *intended non-entailments* such as non-subsumptions, which is of interest to the OWL community. We present some initial thoughts on this problem, discussing the key challenges and outlining a possible solution.

Explaining why a particular entailment *fails* to hold in an ontology is much harder than explaining why it holds. This is because from a model theoretic point of view, a failed entailment implies that there exists at least one model of the ontology in which the entailment is false. From a tableau reasoning standpoint, this translates to the fact that a completion graph representing the ontology with the entailment refuted does *not* contain a clash. This makes explanation tricky since there is no one particular reason for the lack of a clash (i.e., there are potentially infinite ways to generate a clash) and presenting the entire graph as a counter-example is obviously not a sensible solution.

Also, in this case, there is no notion of justification for the failed entailment, since *all* the axioms in the ontology are responsible for the lack of the entailment. Finally, an additional issue that needs to be noted is that fixing the problem can be done rather trivially, by directly adding the entailment as an axiom to the ontology.

Based on these factors, we explore the problem of *debugging* non-subsumptions with a slightly different philosophy. The idea is to devise a service that displays *non-trivial* but sensible axiom changes which would result in the subsumption. Note that the main focus is not explanation, though displaying the *missing* components (axioms) may help the user understand the non-subsumption in the first place better.

Consider an ontology $\mathcal{O}_2$ with the following axioms:

$$\begin{array}{|l|}\hline \mathsf{TexasWine} \equiv \mathsf{Wine} \sqcap \exists \mathsf{locatedIn}.\mathsf{TexasRegion} \\ \mathsf{TexasRegion} \sqsubseteq \exists \mathsf{locatedIn}.\mathsf{USRegion} \\ \mathsf{AmericanWine} \equiv \mathsf{Wine} \sqcap \exists \mathsf{locatedIn}.\mathsf{USRegion} \\ \hline \end{array}$$

In this case, the desired subsumption is $\mathsf{TexasWine} \sqsubseteq \mathsf{AmericanWine}$. Hence, we generate a completion graph for the concept $\mathsf{TexasWine} \sqcap \neg\mathsf{AmericanWine}$ as shown in Figure 8.3.

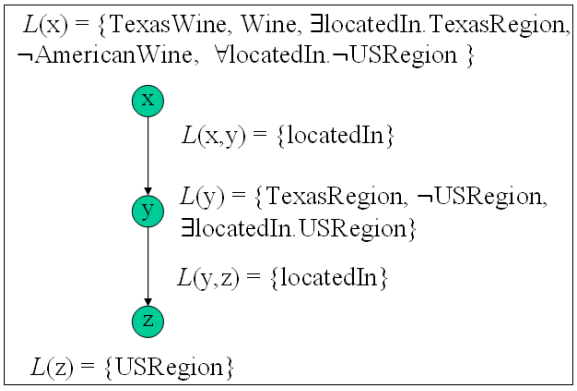

Figure 8.3: Open completion graph reflecting non-subsumption of TexasWine by AmericanWine

As can be seen, the completion graph is not closed and hence $\mathsf{TexasWine} \not\sqsubseteq \mathsf{AmericanWine}$. In order to determine which axioms can be added to $\mathcal{O}_2$ in order to get the desired sub-

sumption, we consider clash-causing changes to the completion graph that would result in it's closure.

Since a clash can be introduced in arbitrarily many ways, we need a heuristic approach to select sensible or likely changes. One heuristic is to consider possible clash interactions between concepts introduced by the subsumer and the subsumee separately in the graph, since this would prevent either the subsumer or the subsumee from becoming independently unsatisfiable. For example, in the above case, the concepts TexasRegion and $\neg$USRegion are introduced separately from TexasWine and AmericanWine, yet appear in the label of the same node, and hence we can consider an axiom such as

TexasRegion $\sqsubseteq$ USRegion

which would result in a clash. Note that it is not hard to translate a tableau event to the corresponding axiom which would cause it (on the lines of our tableau tracing algorithm seen in Chapter 4). Based on this heuristic, we identify the following clash-inducing axiom changes:

1. TexasWine $\sqsubseteq$ AmericanWine (trivial)
2. TexasRegion $\sqsubseteq$ USRegion
3. TexasWine $\sqsubseteq \exists$locatedIn.USRegion
4. $transitive$(locatedIn)

Note that adding any one of the above axioms to $\mathcal{O}_2$ would enable the desired subsumption, and yet prevent any of the concepts TexasWine or AmericanWine from becoming unsatisfiable, i.e., the clashes induced by the axioms only render the graph representing the concept TexasWine $\sqcap \neg$AmericanWine closed.

An additional heuristic to consider is the *size* of the justification set of the desired subsumption, after the axiom has been introduced in the ontology. The idea here is that larger the size of the justification set, the more non-trivial the entailment. Above, the axiom which results in the largest justification set is 4 and interestingly, it is the only case where the justification includes all the original axioms from the ontology. This notion is useful in situations where the user has pinpointed specific axioms (a fragment of the ontology) that he feels should cause the entailment – typically the justification set would need to include the specified axioms. Finally, also note that the justification set can be displayed using the ordering and indenting techniques described in Chapter 7, with the missing axiom highlighted separately. This might help the user understand the cause of the non-subsumption better.

8.2 Exploring Extensions to other Logics

The debugging and repair techniques in this thesis have been developed in the context of DLs. However, DLs are usually a subset of FOL and thus many of the techniques seen here can be directly applied for inconsistent FOL knowledge bases without much modification. For example, tableaux-based algorithms (*semantic tableaux*) are a well known proof procedure for automated reasoning in FOL, and thus the glass-box tableau tracing techniques for Axiom Pinpointing seen in Chapter 4 can be directly translated to the FOL tableau-reasoning case. The basic principle remains the same – trace the clauses

in the FOL KB responsible for the introduction of a particular formulae in a branch of the tableau, and identify the justification for the inconsistency of the KB by using the traces of the contradiction (the *False* clause) in each branch. In some sense, the tableau expansion rules in the FOL case are simpler than in the DL case (e.g., there is no merging of nodes due to cardinality restrictions as in DL) and thus the problems ensuring minimality of the final output (as seen in section 8.1.1) do not arise.

For the more popular FOL proof procedure typically used in Automated Theorem Provers (ATP) – *resolution* – we need to modify the tracing algorithm in accordance with the procedure. The main challenge lies in tracing through the steps of obtaining the CNF (first step of the resolution), which involves normalizing terms (e.g. pushing negation inwards), standardizing variables, splitting across conjunctions and eliminating existentials using *Skolem* functions. This is not impossible, as similar pre-processing steps are also carried out by the tableaux procedures for DLs (using techniques such as normalization, absorption etc.), but it introduces an additional level of complexity that needs to be dealt with.

Irrespective of the type of proof procedure used for FOL reasoning (whether tableaux-based or resolution), it is important to note that the black-box version of the Axiom Pinpointing service can be directly used for FOL debugging, though its performance needs to be tested on realistic FOL KBs to determine the practical use.

Finally, the relationship between description logics and modal logics has been extensively studied over the last decade. [90] pointed that the description logic $\mathcal{ALC}$ can be seen as a variant of the multi-modal logic K_m. Later, the relationship was investigated between more expressive DLs and modal logics, e.g., qualified cardinality restrictions correspond to graded modalities, and nominals in DL which are similarly present in hybrid modal logics. Thus, it is not surprising, that the tableaux algorithms in DLs are similar to the satisfiability checking algorithms in modal logics. This again means, just as in the previous case for FOL, that the diagnosis techniques for DLs can be translated in the modal case, and we leave this as future work.

8.3 Beyond Debugging

The core debugging service developed, Axiom Pinpointing, is used to explain the output of the description logic reasoner since it extracts the minimal set of axioms in the ontology (justifications) responsible for a particular entailment. This service can be utilized in ontology engineering applications outside of ontology debugging and we discuss one such area in detail.

8.3.1 Reasoning over Dynamic Ontologies

Justifications act as a form of truth-maintenance that can be used to optimize reasoning tasks for dynamic or changing ontologies. This is especially useful in the context of ontology editing (when coupled with a reasoner), where interactivity is essential from the user point of view.

To elaborate, once a reasoner has processed an ontology and derived its key entail-

ments (e.g., subsumption between atomic concepts), the justifications for the individual entailments can be stored separately. Then, when the ontology is modified by say removing an axiom, we can inspect the justification sets to determine which entailments are lost directly, i.e., the reasoner can skip entailment tests based on previously cached justifications. These justification sets can be updated on the fly as and when new axioms are introduced.

Recently, we have also explored the use of the glass-box version of Axiom Pinpointing (tableau tracing) to incrementally update the completion graph built internally by the reasoner, which speeds up the reasoning significantly when dealing with dynamic ontologies [42]. The basic idea is the following: the tracing algorithm computes the relation between axioms in the ontology and the various parts of the completion graph, and thus when the ontology is modified, instead of discarding the previous completion graph and starting from scratch (as is normally done by the reasoner), we update the graph in accordance with the added/removed axioms only. Obviously, this process saves a lot of time which was previously wasted in redoing the graph expansion each time the ontology is changed.

The current solution works for updating assertions related to individuals (ABox updates), which itself has many real-world use cases. Two popular examples include dynamic web services frameworks where devices register or deregister their logical descriptions (and supporting ontologies) quite rapidly; and Semantic Web portals, which often allow content authors to modify or extend the ontologies leading to a reorganization of the site structure/content. In both scenarios, optimizing reasoning helps reduce maintenance time and effort.

Appendix A

Appendix: Swoop – Web Ontology Browser/Editor

In this section, we discuss specific features in the OWL Ontology Editor, Swoop [57] that are tailored towards the understanding and analysis of OWL ontologies.

In particular, we focus on four different aspects:

- Explanation of concept definition (useful for understanding error cause)
- Browsing, comparing and querying ontological information (useful for understanding dependencies between entities)
- Change management (useful for experimenting / repair)
- Collaborative discussion and annotation of ontological data (useful for sharing explanations and repair solutions)

A.0.2 Explaining Concept Definition: Natural Language Paraphrases

In order to help users understand the meaning behind complex concept definitions, we have developed a plugin for Swoop that generates natural language (NL) paraphrases for OWL Concepts based on a variety of NLP techniques [45]. The goal is to ensure both fluency (readability) and accuracy of the output, in terms of preserving the meaning conveyed by it's description logic formalism (see Figure A.1 for an example). The NL generation approach is a generic domain-independent one, and is completely automated.

The algorithm works by building a parse tree from the concept definition axioms, and generating sentences by traversing the tree and inserting textual phrases denoting DL operators between ontological terms and relationships. Various heuristics - syntactic, using a part of speech (POS) tagger, and semantic, using a reasoner, are used to improve the quality of the NL sentences.

While there exist some obvious limitations of the work, such as it's reliance on standard naming conventions and it's inability to cope with deeply-nested logical operators, we have found that in a lot of tested ontologies, the algorithm generates readable NL paraphrases, which are useful for getting a quick overview of the concept meaning (see [41] for a related pilot study).

Figure A.1 shows an example of the NL generation when applied to a concept in the Wine OWL ontology.

A.0.3 Browsing, Comparing and Querying data

Swoop has a *debug* mode wherein the basic rendering of entities is augmented with information obtained from a reasoner. Different rendering styles, formats, and icons are used to highlight key entities and relationships that that are likely to be helpful to debugging process. For example, all *inferred* relationships (axioms) in a specific entity definition are italicized and are obviously not editable directly. On a similar note, in the

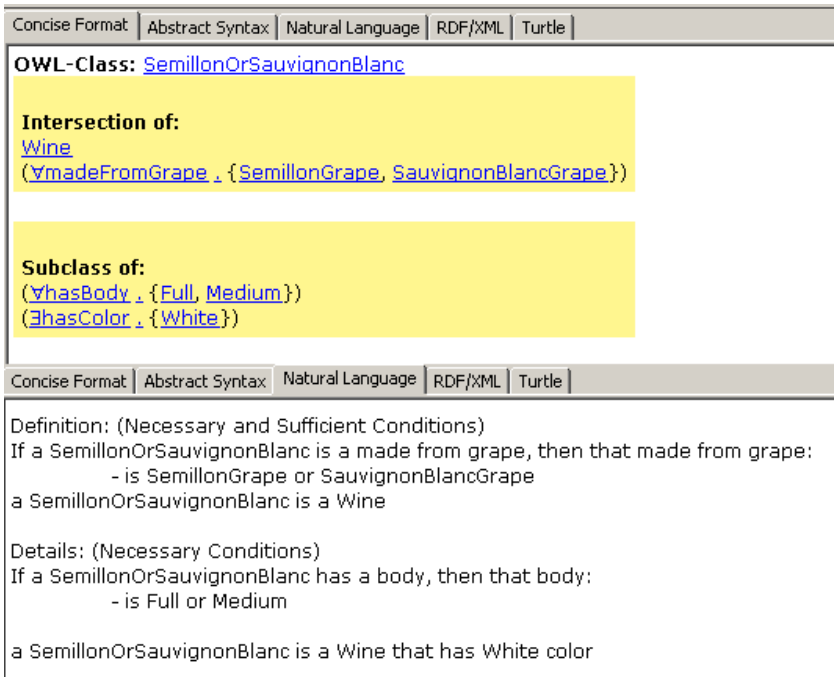

Figure A.1: **Natural Language:** paraphrase describing the concept in the Wine OWL Ontology.

case of multiple ontologies, i.e., when one ontology imports another, all *imported* axioms in a particular entity definition are italicized as well. Highlighting them helps the modeler differentiate between explicit assertions in a single context and the net assertions (explicit plus implied) in a larger context (using imports), and can also reveal unintended semantics.

In addition to displaying information about named classes, Swoop renders information such as sub/super classes of complex class expressions as shown in Figure A.2 (Region 2). This sort of ad hoc "on-demand" querying helps reveal otherwise hidden dependencies.

Consider the case of the unsatisfiable class Koala depicted in Figure A.2, which contains three labeled regions. The figure also emphasizes the *Comparator* feature in Swoop, which allows users to compare and contrast any arbitrary set of entities. Region 1 shows the definition of the Koala class in terms of it's subclass-of axioms: note the presence of the class expression ∃isHardWorking.false and the named class Marsupials mentioned here. Now, clicking on the class expression reveals that it is an inferred subclass of Person (Region 2)[1], and clicking on Marsupials shows that it is defined as *disjoint* with class Person (Region 3). Thus, the contradiction is found – an instance of Koala is forced to be an instance of Person and ¬Person at the same time, and the bug can be fixed accordingly.

Finally, Swoop has an interesting non-standard search feature which can be use-

[1]A simple heuristic to manually debug an unsatisfiable class is to inspect it is asserted and inferred subclass relationships that could potentially cause a contradiction, as is what motivates clicking the class expression link here.

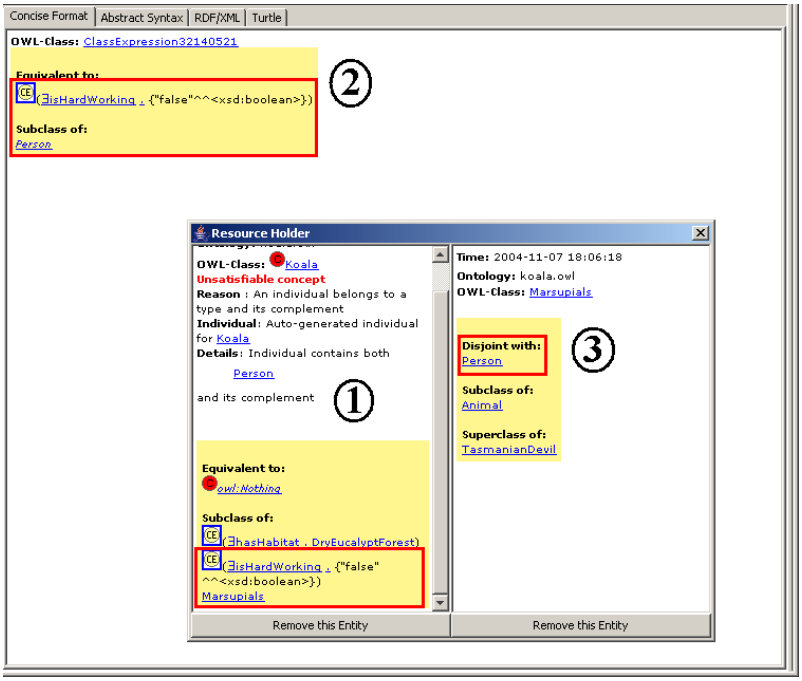

Figure A.2: The class Koala is unsatisfiable because (1) Koala is a subclass of ∃isHardWorking.false and Marsupials; (2) ∃isHardWorking.false is a subclass of Person; and (3) Marsupials is a subclass of ¬Person (disjoint). Note that the regions outlined in red are not automatically generated by the tool but are presented here for clarity.

ful during ontology debugging. This feature known as *Show References* highlights the usage of an OWL entity (concept/property/individual) by listing all references of that entity in local or external ontological definitions. The *Sweet-JPL* ontology set[2] presents an excellent use case for debugging using this feature. The class OceanCrustLayer is found to be unsatisfiable and a reason displayed for the clash is *'Any member of OceanCrustLayer has more than one value for the functional property hasDimension'* (Note: Clash detection is explained later). Now, running a ShowReferences search on the property hasDimension, returns four classes GeometricObject(0..3)D, each of which has a different value restriction on the functional property hasDimension. This suggests that the unsatisfiable class is somehow related to more than one of these four classes causing the cardinality violation. This is indeed the case since by looking at the class hierarchy, one can note that OceanCrustLayer is a subclass of both the classes, GeometricObject2D and GeometricObject3D, and thus the reason for the contradiction becomes apparent.

A.0.4 Change Management

Part of good debugging support for OWL ontologies is making experimentation involving ontology changes safe, easy, and effective. Swoop has an ontology evolution

[2]Sweet-JPL Ontologies are located at http://sweet.jpl.nasa.gov/ontology/. The bug in the ontology was fixed on May 24, 2005 after we e-mailed the ontology authors at NASA informing them about it. The previous faulty version can be found at http://www.mindswap.org/ontologies/debugging/buggy-sweet-jpl.owl

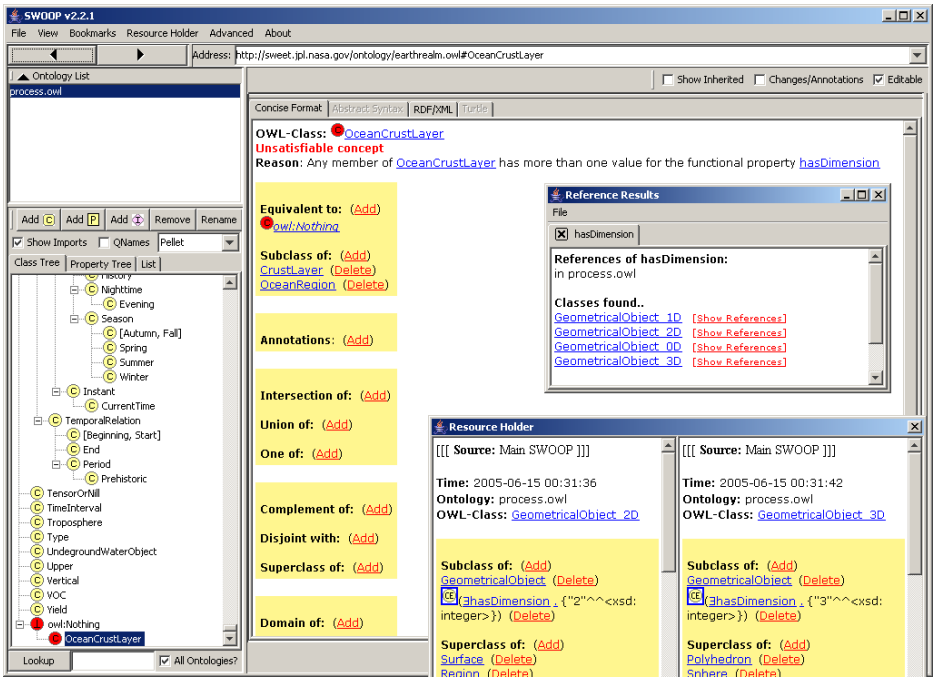

Figure A.3: The *Show References* feature (used along with the clash information and the resource holder) is used to hint at the source of the highly non-local problem for the unsatisfiable class OceanCrustLayer.

framework that supports the ad hoc undo/redo of changes (with logging).

Swoop uses the OWL API [9] to model ontologies and their associated entities, benefiting from it's extensive and clean support for changes. The OWL API separates the representation of changes from the application of changes. Each possible change type has a corresponding Java class in the API, which is subsequently applied to the ontology (essentially, the Command design pattern). These classes allow for the rich representation of changes, including metadata about the changes. The change sets can be serialized in RDF/XML and exchanged among ontology users, making it possible to apply patches of changes to ontologies as and when desired.

Swoop also provides the ability to checkpoint and archive different ontology versions. Each change set or checkpoint can be saved at three different granularity levels - entity, ontology, workspace, which basically specify it's *scope*. While the change logs can be used to explicitly track the evolution of an ontology, checkpoints allows the user to switch between versions directly exploring different modeling alternatives.

A.0.5 Collaborative Discussion Using Annotea

For collaborative discussion of ontologies using Swoop, we use the Annotea framework [56], which takes the idea of separating annotations about ontologies from the core ontologies themselves and provides both a specific RDF based, extensible annotation vocabulary, and a protocol for publishing and finding out-of-band annotations (annotations that do not live inside the document being annotated).

Annotea support in Swoop is provided via a simple plug in whose implementation is based on the standard W3C Annotea protocols [102] and uses the default Annotea RDF schema to specify annotations (see Figure A.4). Any public Annotea Server can then be used to publish and distribute the annotations created in Swoop. The default annotation types (comment, advice, example, etc) seem an adequate base for human oriented ontology annotations.

We have extended the Annotea Schema with the addition of an OWL ontology for a new class of annotations — ontology changes (similar to [59]). The "Change" annotation defined by the Annotea projected was designed to indicate a proposed change to the annotated document, with the proposal described in HTML-marked-up natural language. In our extended ontology, change individuals correspond to specific changes made in Swoop during editing.

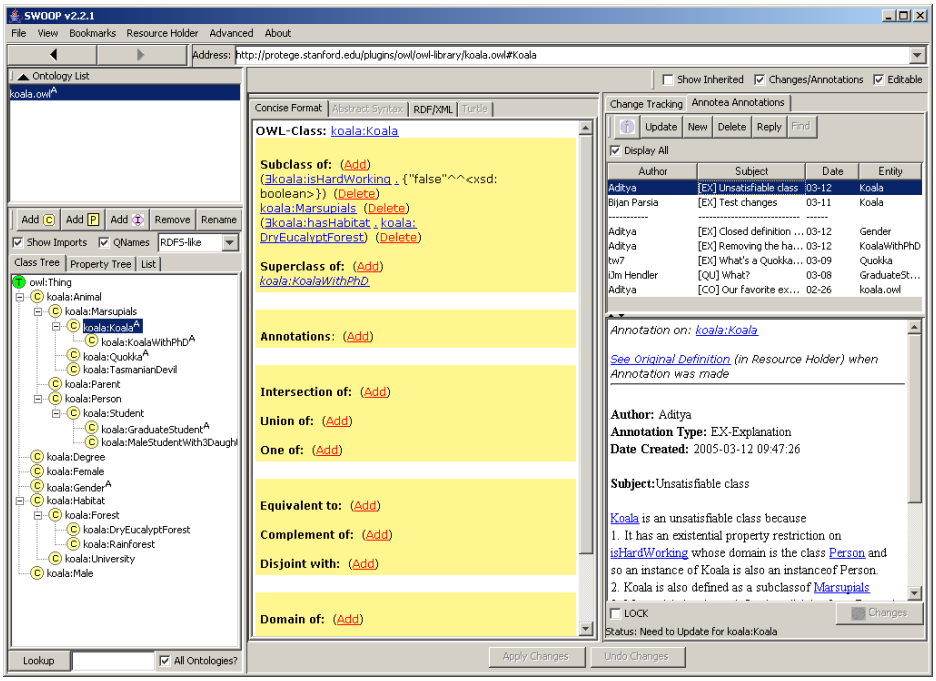

Figure A.4: Using Annotea Client to Collaboratively Discuss and Debug Ontology

The Swoop change annotations can be published and retrieved by Annotea servers, or any other annotation distribution mechanism. The retrieved annotations can then be browsed, filtered, endorsed, recommended, and selectively accepted. A similar collaborative framework based on an interactive dialogue was implemented in a more local (tool-specific) context in the WebOnto system [30]. However, we decided to exchange annotations using the Annotea protocol to make the collaboration less tool-specific (any Annotea client can be used to discuss ontology annotations), and to allow users to arbitrarily extend the Annotea schema the way we have for ontology-change sets. These change sets also make it possible to define "virtual versions" of an ontology, by specifying a base ontology and a set of changes to apply to it. [3]

[3]Note that in certain cases, changes may not be applicable to the ontology, if the change operation refers

Once a series of changes has proven effective in removing the defect and seems sensible, the modeler can use Swoop's integrated Annotea client to publish the set of changes plus a commentary as shown in Figure A.4. Other subscribers to the Annotea store can see these changes and commentary in context they were made, apply the changes to see their effect, and publish responses. These exchanges persist, providing a repository of real cases for subsequent modelers to study.

As future work, we plan on using the collaborative annotea-based framework in Swoop to maintain a robust and extensible library of error patterns. As seen in Chapter 6, the Ontology Repair service can make use of such a library to suggest axiom rewrites in the repair solutions.

to an entity that is not present (defined) in the ontology. In such cases, a warning message is reported to the user describing the reason for the change conflict.

BIBLIOGRAPHY

[1] Gardenfors P. Markinson D. Alchourron, C.E. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50:510–530, 1985.

[2] Gregoris Antoniou, Frank van Harmelen, Robert Plant, and Jan Vanthienen. Verification and validation of knowledge-based systems - report on two 1997 events. *AI Magazine*, 19(3):123–126, Fall 1998.

[3] F. Baader. Logic-based knowledge representation. In M. J. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today, Recent Trends and Developments*, number 1600, pages 13–41. Springer Verlag, 1999.

[4] F. Baader and W. Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95. Cambridge University Press, 2003.

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

[6] Franz Baader and Bernhard Hollunder. Embedding defaults into terminological knowledge representation formalisms. Technical Report RR-93-20, 1993.

[7] Kenneth Baclawski, Christopher J. Matheus, Mieczyslaw M. Kokar, Jerzy Letkowski, and Paul A. Kogut. Towards a symptom ontology for semantic web applications. In *International Semantic Web Conference*, pages 650–667, 2004.

[8] S. Bechhofer, I Horrocks, C. Goble, and R. Stevens. OilEd: a reason-able ontology editor for the Semantic Web. *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, September 2001.

[9] S. Bechhofer, P. Lord, and R. Volz. Cooking the semantic web with the owl api. *Proceedings of the International Semantic Web Conference*, October 2003.

[10] D. Beckett and B. McBride. RDF/XML syntax specification. W3C Recommendation, 2004.

[11] S. Benferhat, S. Kaci, D. Berre, and M. Williams. Weakening conflicting information for iterated revision and knowledge integration. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2004.

[12] Tim Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999.

[13] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. Classic: A structural data model for objects. In *Proc. of SIGMOD-89*, 1989.

[14] A. Borgida, E. Franconi, I. Horrocks, D. McGuinness, and P. Patel-Schneider. Explaining $\mathcal{ALC}$ subsumption. In *Proc. of DL-99*, 1999.

[15] Martin Brain and Marina De Vos. Debugging logic programs under the answer set semantics. In *Answer Set Programming*, 2005.

[16] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml). volume 2, pages 27–66, 1997.

[17] Dan Brickley and R Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/tr/rdf-schema/. February 2004.

[18] Laure Brisoux, Eric Gregoire, and Lakhdar Sais. Validation of knowledge-based systems by means of stochastic search. In *DEXA Workshop*, pages 41–46, 1998.

[19] T. Burners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5), May 2001.

[20] L. Byrd. Understanding the control flow of prolog programs. *Proceedings of the Workshop on Logic Programming*, 1980.

[21] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Daniele Nardi. Reasoning in expressive description logics. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1581–1634. Elsevier Science Publishers, 2001.

[22] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *IJCAI*, pages 1494–1501, 1993.

[23] Thierry Coupaye, Claudia Roncancio, and Christophe Bruley. A visualization service for event-based systems. In *Proc. 15emes Journees Bases de Donnees Avancees, BDA*, pages 181–199, 1999.

[24] P. Pinheiro da Silva, S. McGuinness, and R. Fikes. A proof markup language for semantic web services. Technical report, 2004. TR KSL-04-01, Stanford University, 2004.

[25] Carlos Viegas Damasio, Luis Moniz Pereira, and Michael Schroeder. REVISE: Logic programming and diagnosis. In *Logic Programming and Non-monotonic Reasoning*, pages 354–363, 1997.

[26] R. Davis. Application of meta-level knowledge to the construction, maintenance, and use of large knowledge bases. *Ph.D. Dissertation, Dept. of Computer Science*, 1976.

[27] M. Dean and G. Schreiber. OWL Web Ontology Language Reference W3C Recommendation. http://www.w3.org/tr/owl-ref/. February 2004.

[28] Xi Deng, Volker Haarslev, and Nematollaah Shiri. A framework for explaining reasoning in description logics. In *Proceedings of the AAAI Fall Symposium*, pages 55–61, 2005.

[29] Li Ding, Rong Pan, Tim Finin, Anupam Joshi, Yun Peng, and Pranam Kolari. Finding and Ranking Knowledge on the Semantic Web. In *Proceedings of the 4th International Semantic Web Conference*, LNCS 3729, pages 156–170. Springer, November 2005.

[30] John Domingue. Tadzebao and webonto: Discussing, browsing, and editing ontologies on the web. In *11th Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1998.

[31] Wlodzimierz Drabent, Simin Nadjm-Tehrani, and Jan Maluszynski. Algorithmic debugging with assertions. In *Workshop on Meta-Programming in Logic*, pages 501–521, 1988.

[32] A. Felty and D. Miller. Proof explanation and revision. *Dept. of Computer and Information Science School of Engg. and Applied Science Report*, 1988.

[33] Armin Fiedler. *P.rex*: An interactive proof explainer. In Rejeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning — 1st International Joint Conference, IJCAR 2001*, number 2083 in LNAI, pages 416–420, Siena, Italy, 2001. Springer Verlag.

[34] Giorgos Flouris, Dimitris Plexousakis, and Grigoris Antoniou. Updating dls using the agm theory: A preliminary study. In *Description Logics*, 2005.

[35] G. Friedrich and Shchekotykhin. K. Diagnosis of description logic knowledge bases. In *Proceedings of Fourth Internal Semantic Web Conference ISWC*, 2005.

[36] Peter Gardenfors. Belief revision: An introduction. *Cambridge Tracts in Theoretical Computer Science*, (29):1–28, 1992.

[37] H.P. Grice. Logic and conversation. *P. Cole and J.L. Morgan, editors, Syntax and semantics*, 3:43–58, 1975.

[38] Nicola Guarino and Christopher Welty. Evaluating ontological decisions with ontoclean. *Commun. ACM*, 45(2):61–65, 2002.

[39] Volker Haarslev and Ralf Moller. High performance reasoning with very large knowledge bases: A practical case study. In *IJCAI*, pages 161–168, 2001.

[40] Volker Haarslev, Ralf Möller, and Anni-Yasmin Turhan. Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. *Lecture Notes in Computer Science*, 2083:61–??, 2001.

[41] Christian Halaschek, Jennifer Golbeck, Bijan Parsia, Vladimir Kolovski, and Jim Hendler. Image browsing and natural language paraphrases of semantic web annotations. In *First International Workshop on Semantic Web Annotations for Multimedia (SWAMM)*, Edinburgh, Scotland, 2006.

[42] Christian Halaschek-Wiener, Aditya Kalyanpur, and Bijan Parsia. Extending tableau tracing for abox updates. In *UMIACS Tech Report*, 2006. http://www.mindswap.org/papers/2006/aboxTracingTR2006.pdf.

[43] Christian Halaschek-Wiener, Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Description logic reasoning for dynamic aboxes. In *Accepted in DL 2006*, 2006.

[44] P. Hayes. Resource description framework (RDF) semantics. W3C Recommendation, 2004.

[45] Daniel Hewlett, Aditya Kalyanpur, Vladamir Kovlovski, and Chris Halaschek. Effective natural language paraphrasing of ontologies on the semantic web. In *End User Semantic Web Interaction Workshop, International Semantic Web Conference (ISWC)*, Galway, Ireland, November 2005.

[46] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. DFKI Research Report RR-91-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, 1991.

[47] I. Horrocks and P. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In *Proc. of the 2nd International Semantic Web Conference (ISWC)*, 2003.

[48] I. Horrocks and U. Sattler. Ontology reasoning in the SHOQ(D) description logic. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

[49] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–264, 2000.

[50] Ian Horrocks. FaCT and iFaCT. In *Description Logics*, 1999.

[51] Ian Horrocks. Implementation and optimization techniques. pages 306–346, 2003.

[52] Ian Horrocks and Ulrike Sattler. A tableaux decision procedure for SHOIQ. In *Proc. of IJCAI 2005*, 2005.

[53] Xiaorong Huang and Armin Fiedler. Presenting machine-found proofs. In Michael McRobbie and John Slaney, editors, *Proc. 13th Conference on Automated Deduction, New Brunswick/NJ, USA*, volume 1104, pages 221–225. Springer-Verlag, 1996.

[54] Zhisheng Huang, Frank van Harmelen, and Annette ten Teije. Reasoning with inconsistent ontologies. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, page xxx, Edinburgh, Scotland, August 2005.

[55] U. Hustadt, B. Motik, and U. Sattler. Reducing $\mathcal{SHIQ}^-$ description logic to disjunctive datalog programs. In D. Dubois, C. Welty, and M.-A. Williams, editors, *Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR2004)*, pages 152–162. AAAI Press, 2004.

[56] J. Kahan, M-R. Koivunen, E. Prud'Hommeaux, and R. Swick. Annotea: An open RDF infrastructure for shared web annotations. *Proc. of the WWW10 International Conference*, May 2001.

[57] A. Kalyanpur, B. Parsia, E.Sirin, B. Cuenca-Grau, and J. Hendler. Swoop: A web ontology editing browser. *Journal of Web Semantics, Vol 4, Issue 2*, 2006.

[58] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging unsatisfiable classes in OWL ontologies. *Journal of Web Semantics, Vol 3, Issue 4*, 2005.

[59] M. Klein and N. Noy. A component-based framework for ontology evolution. *Workshop on Ontologies and Distributed Systems at IJCAI*, 2003.

[60] Kevin Lee and Thomas Meyer. A classification of ontology modification. In *Australian Conference on Artificial Intelligence*, pages 248–258, 2004.

[61] T. Liebig and O. Noppens. Ontotrack: Combining browsing and editing with reasoning and explaining for owl lite ontologies. In *Proceedings of the 3rd International Semantic Web Conference (ISWC) 2004*, Japan, November 2004.

[62] Thorsten Liebig and Michael Halfmann. Explaining Subsumption in $\mathcal{ALEHF}_{R^+}$ TBoxes. In Ian Horrocks, Ulricke Sattler, and Frank Wolter, editors, *Proc. of the 2005 International Workshop on Description Logics - DL2005*, pages 144–151, Edinburgh, Scotland, July 2005.

[63] Hongkai Liu, Carsten Lutz, Maja Milicic, and Frank Wolter. Updating description logic aboxes. In *KR*, Lake District, UK, 2006.

[64] C. Lutz. Complexity of terminological reasoning revisited. In *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning LPAR'99*, pages 181–200. Springer-Verlag, 6 – 10, 1999.

[65] C. Lutz. Description logics with concrete domains—a survey, 2002.

[66] F. Manola and E. Miller. RDF Primer W3C Recommendation. http://www.w3.org/tr/rdf-primer/. February 2004.

[67] D. McGuinness. *Explaining Reasoning in Description Logics*. PhD thesis, New Brunswick, New Jersey, 1996.

[68] D. McGuinness and P. Pinheiro da Silva. Infrastructure for web explanations. *In Second International Semantic Web Conference, ISWC*, 2003.

[69] Deborah McGuinness and Alexander Borgida. Explaining subsumption in description logics. In Chris Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 816–821, San Francisco, 1995. Morgan Kaufmann.

[70] Richard Fikes James Rice McGuinness, Deborah L. and Steve Wilder. The chimaera ontology environment. *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*, 2000.

[71] Thomas Meyer, Kevin Lee, and Richard Booth. Knowledge integration for description logics. In *AAAI*, pages 645–650, 2005.

[72] Guido Moerkotte and Peter C. Lockemann. Reactive consistency control in deductive databases. *ACM Transactions on Database Systems*, 16(4):670–702, 1991.

[73] Boris Motik. On the properties of metamodeling in owl. In *International Semantic Web Conference*, pages 548–562, 2005.

[74] B. Nebel. Reasoning and revision in hybrid representation systems. *Lecture Notes in AI*, 1990.

[75] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. *Proc. of the 25 Int. Conference on Software Engineering*, 2003.

[76] N. Noy, M. Sintek, S. Decker, M. Crubezy, R. Fergerson, and M. Musen. Creating semantic web contents with Protégé-2000. *IEEE Intelligent Systems*, 2001.

[77] Perkins W.A. Laffey T.J. Nyugen, T.A. and D. Pedora. Checking an expert system for consistency and completeness. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985.

[78] Daniel Oberle, Raphael Volz, Boris Motik, and Steffen Staab. An extensible ontology software environment. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, chapter III, pages 311–333. Springer, 2004.

[79] D.A.S. Oliveira, C.S. de Souza, and E.H. Haeusler. Structured argument generation in a logic-based kb-system. *Proceedings of the Second Conference on Information-Theoretic Approaches to Logic, Language and Computation*, pages 173–181, 1996.

[80] B. Parsia, C. Halaschek-Wiener, E. Sirin, and A. Kalyanpur. Classification maintenance for expressive description logics. Technical report, (in progress), 2005. Available online at http://www.mindswap.org/papers/TR-incclass.pdf.

[81] Chintan Patel, Kaustubh Supekar, Yugyung Lee, and E. K. Park. Ontokhoj: a semantic web portal for ontology searching, ranking and classification. In *WIDM*, pages 58–61, 2003.

[82] Michael J. Pazzani and Clifford A. Brunk. Detecting and correcting errors in rule-based expert systems: An integration of empirical and explanation-based learning. Technical Report ICS-TR-90-38, 1990.

[83] L.M. Pereira and J.J. Alferes. Well founded semantics for logic programs with explicit negation. *Proc. ECAI92*, pages 102–106, 1992.

[84] Luis Moniz Pereira, Carlos Viegas Damasio, and Jose Julio Alferes. Diagnosis and debugging as contradiction removal in logic programs. In *Portuguese Conference on Artificial Intelligence*, pages 183–197, 1993.

[85] Robert T. Plant. Tools for the validation & verification of knowledge-based systems. *University of Miami*, 1995.

[86] Alun Preece. Evaluating verification and validation methods in knowledge engineering. *University of Aberdeen*, 2001.

[87] Alan L. Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, and Chris Wroe. Owl pizzas: Practical experience of teaching owl-dl: Common errors & common patterns. In *EKAW*, pages 63–81, 2004.

[88] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

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

[90] Klaus Schild. A correspondence theory for terminological logics: preliminary report. In *Proceedings of IJCAI-91, 12th International Joint Conference on Artificial Intelligence*, pages 466–471, Sidney, AU, 1991.

[91] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proc. of IJCAI, 2003*, 2003.

[92] Stefan Schlobach. Debugging and semantic clarification by pinpointing. In *ESWC*, pages 226–240, 2005.

[93] Stefan Schlobach. Diagnosing terminologies. In *AAAI*, pages 670–675, 2005.

[94] Ehud Y. Shapiro. Algorithmic program debugging. *MIT Press*, May 1982.

[95] E. Shortliffe. Computer-based medical consultations: Mycin. In *Elsevier*, 1996.

[96] Evren Sirin, Bernardo Cuenca Grau, and Bijan Parsia. From wine to water: Optimizing description logic reasoning for nominals. In *International Conference on the Principles of Knowledge Representation and Reasoning (KR-2006)*, 2006. To Appear.

[97] Evren Sirin and Bijan Parsia. Pellet: An owl dl reasoner. In *Description Logics*, 2004.

[98] Y. Sure, M. Erdmann, J. Angele, S. Staff, R. Studer, and D. Wenke. OntoEdit: Collaborative ontology development for the Semantic Web. *Proceedings of the International Semantic Web Conference (ISWC)*, June 2002.

[99] M. Suwa, A. Scott, and E. Shortliffe. An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine*, 3(4):16–21, 1982.

[100] W. Swartout. XPLAIN: A system for creating and explaining expert consulting systems. *Artificial Intelligence*, 21(3):285–325, 1983.

[101] W. Swartout, C. Paris, and J. Moore. Explanations in knowledge systems: Design for explainable expert systems. *IEEE Intelligent Systems*, 6(3):58–64.

[102] R. Swick, E. Prud'Hommeaux, M-R. Koivunen, and J. Kahan. Annotea protocols. *http://www.w3.org/2001/Annotea/User/Protocol.html*, 2001.

[103] S. Tobies. Complexity results and practical algorithms for logics in knowledge representation. *PhD thesis, RWTH Aachen*, 2001.

[104] V.Haarslev and R.Moeller. Racer system description. In *Proc. of the Joint Conf. on Automated Reasoning (IJCAR 2001). Volume 2083 of Lecture Notes in Artificial Intelligence, pages 701-705*, 2001.

[105] Hai Wang, Matthew Horridge, Alan L. Rector, Nick Drummond, and Julian Seidenberg. Debugging owl-dl ontologies: A heuristic approach. In *International Semantic Web Conference*, pages 745–757, 2005.

[106] Geoffrey I. Webb, Jason Wells, and Zijian Zheng. An experimental evaluation of integrating machine learning with knowledge acquisition. *Machine Learning*, 35(1):5–23, 1999.