Constraint Generation and Reasoning in OWL

by

Thomas Henry Briggs, VI

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

APPROVAL SHEET

Title of Thesis: Constraint Generation and Reasoning in OWL

Name of Candidate: Thomas H. Briggs, VI
Doctor of Philosophy, 2008

Thesis and Abstract Approved: _____

Dr. Yun Peng
Professor
Department of Computer Science and
Electrical Engineering

Date Approved: November 17, 2008

Curriculum Vitae

Name: Thomas Henry Briggs, VI.

Permanent Address: 203 Whitmer Road, Shippensburg, Pennsylvania.

Degree and date to be conferred: PhD, December 2008.

Date of Birth: September 29, 1971.

Place of Birth: Allentown, Pennsylvania.

Secondary Education: Parkland High School, Orefield, Pennsylvania.

Collegiate institutions attended:

University of Maryland, Baltimore County, PhD Computer Science, 2008.
Shippensburg University, MS Computer Science, 2001.
Shippensburg University, BS Computer Science, 1996.

Major: Computer Science.

Professional publications:

Briggs, T. and Girard, C. D. Tools and techniques for test-driven learning in CS1. J. Computing Sciences in Colleges, Volume 22, Number 3 (Jan. 2007), pp. 37-43.

Wellington, C. A., Briggs, T., and Girard, C. D. 2005. Examining team cohesion as an effect of software engineering methodology. In Proceedings of the 2005 Workshop on Human and Social Factors of Software Engineering (St. Louis, Missouri, May 16 - 16, 2005). HSSE '05. ACM Press, New York, NY, 1-5..

Carol Wellington, Thomas Briggs, C. Dudley Girard, The Impact of Agility on a Bachelors Degree in Computer Science, agile, pp. 400-404, AGILE 2006 (AGILE'06), 2006

Wellington, C., Briggs, T., Girard, D., Comparison of Student Experiences with Plan-Driven and Agile Methodologies. In proceedings of the 20th Frontiers in Education Conference, 2005. Indianapolis, IN. IEEE Press, ISBN: 0-7803-9077-6.

Briggs, T. Techniques for Active Learning in CS Courses, Presented to the Eastern Conference of the Consortium for Computing Sciences in Colleges (CCSCE 2005), October 11, 2005, New Rochelle, NY.

Briggs, T., Oates, T. Discovering Domain Specific Composite Kernels, In Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Annual Conference on Innovative Applications of Artificial intelligence, 732-739. Menlo Park, Calif.: AAAI Press.

Professional positions held:

Assistant Professor of Computer Science. (2002 – Present)
Shippensburg University, Shippensburg, PA.

Academic Information Manager. (1999 – 2002)
Shippensburg University, Shippensburg, PA.

Regional Hub Administrator. (1997 – 1999)
Shippensburg University, Shippensburg, PA.

Library Systems Manager. (1996 – 1997)
Shippensburg University, Shippensburg, PA.

PC & Network Specialist. (1995 – 1996)
Shippensburg University, Shippensburg, PA.

Systems Administrator. (1991 – 1993).
Joseph Ciccone & Sons, Inc., Whitehall, PA

ABSTRACT

Title of Thesis: CONSTRAINT GENERATION AND REASONING IN OWL

THOMAS H. BRIGGS, DOCTOR OF PHILOSOPHY, 2008

Thesis directed by: Dr. Yun Peng, Professor
Department of Computer Science and
Electrical Engineering

The majority of OWL ontologies in the emerging Semantic Web are constructed from properties that lack domain and range constraints. Constraints in OWL are different from the familiar uses in programming languages and databases. They are actually type assertions that are made about the individuals which are connected by the property. Because they are type assertions these assertions can add vital information to the individuals involved and give information on how the defining property may be used. Three different automated generation techniques are explored in this research: disjunction, least-common named subsumer, and vivification. Each algorithm is compared for the ability to generalize, and the performance impacts with respect to the reasoner. A large sample of ontologies from the Swoogle repository are used to compare real-world performance of these techniques. Using generated facts is a type of default reasoning. This may conflict with future assertions to the knowledge base. While general default reasoning is non-monotonic and undecidable a novel approach is introduced to support efficient contraction of the default knowledge. Constraint generation and default reasoning, together, enable a robust and efficient generation of domain and range constraints which will result in the inference of additional facts and improved performance for a number of Semantic Web applications.

Constraint Generation and Reasoning in OWL

by

Thomas Henry Briggs, VI

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

To my family.

ACKNOWLEDGMENTS

My first acknowledgement is to my family, especially my wife, Susan. Without their support, understanding, patience, and making due while I travelled to Baltimore and worked late nights this would not have been possible. Without their love and confidence I would not have had the courage or will to pursue this degree.

I would also like to acknowledge Dr. Carol Wellington who encouraged me to begin the degree and helped coach me through the process.

Finally, I would like to acknowledge Dr. Yun Peng's mentorship, patience, and understanding. I hope that sometime in my career I can show the same qualities to my students that he has shown in me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The Semantic Web is an evolving set of technologies used to publish data to enable intelligent agents. One major step in the development of the future Semantic Web must be a set of enabling technologies that will ease the transition from data to knowledge. This research explores one frequently overlooked aspect of this transition: property constraints. This chapter provides an introduction to the issues and an overview of the proposed solutions. The chapter is organized as follows: a description of the problem appears in Section 1.1, the formal thesis statement appears in Section 1.2, and an overview of the dissertation outline appears in Section 1.3.

1.1 Problem Description

The Semantic Web languages, such as OWL, allow encoding and organization of domain specific knowledge in ontologies in order to support efficient reasoning processes. A domain is described with a collection of classes, properties, and individual definitions. Classes describe groups of individuals and define their criteria for membership. Properties describe relationships between individuals and between individuals and simple data values. Individuals describe specific instances of classes through assertions of class membership and properties. The terminological and assertional description contained in the ontology is

expected to be neither complete nor minimal. The reasoner is used as a tool to infer the missing information.

One specific area that is problematic for ontology development is the role of property domain and range constraints. In OWL, a constraint specifies the types of individuals which fill a particular property. These constraints are not interpreted as restrictions about which individuals can be used with a property, rather they serve as assertions about the types of individuals connected by the property. There is valuable information that can be inferred from domain and range constraints.

In many cases, constraints are not specified by ontology developers. The result of unspecified constraints are vague semantics for that property and the individuals that it connects. This work explores several reasons for this and provides techniques to construct constraints from the ontology.

1.1.1 Domain and Range Constraints in OWL

There are some subtleties to the current revision of OWL and its implementation of domain and range constraint definition and use. There are three main pitfalls when using domain and range constrains in OWL: the default domain and range are `owl:Thing`, the problematic interpretation of constraints between traditional programming languages and OWL; and the lack of specific mapping from a domain to a ranges.

When an OWL property's domain or range is left blank it defaults to be the concept `owl:Thing`, which is equivalent to the Description Logic concept top ($\top$) and is a concept that represents everything. This says very little about what types of individuals that are related to a property. These individuals are already a generic thing and this assertion does not add any information. This results in a potential loss of information if there are no other statements to stand in for the missing constraints. This also results in looser semantics to check for inconsistencies in the fillers of a property.

For example, we may see an assertion that individual (ALICE PROP BOB). If this assertion is the only information we have about these individuals then we know nothing else about ALICE or BOB, except that they are an owl : Thing and are connected through property PROP. Without knowledge of the domain being modeled (and depending on the names of the symbols used here), these are not very meaningful semantics.

Domain and range constraints are interpreted differently in OWL than they are in other traditional database and programming languages. In these traditional languages, when a method is restricted to allow only certain types of values, then any invocation of that method will check the types to see that they match and throw an error if they do not. In OWL, when an individual fills the slots of a property (either as subject or object) that individual is asserted to be of a type of the domain or range. The distinction is subtle. In traditional languages, domain and range statements describe a restriction on the valid types that may be used. In Description Logics, these statements are assertions about the type of the individual that are connected.

These differences cause real problems for users familiar with the traditional database and programming systems who are transitioning to OWL and other Description Logics. They will need to address the difference in interpretation of domains and ranges to avoid creating semantic errors in their knowledge bases (SWAD-Europe 2008). For example, suppose the property drivesCar is described as having a domain Person and range of Car. Later, the assertion that YELLOW is a Color, and YELLOW drivesCar HONDA is added to the knowledge base. In a traditional language this would cause an exception (YELLOW is a Color, not a Person). In the Description Logic case, the reasoner will make YELLOW a Person unless there is some assertion that prevents this situation.

In the current 1.0 version of OWL, domain and range constraints are specified as a collection of statements and cannot be paired with each other. For example, OWL 1.0 cannot describe that property hasRank has a range of ArmyRanks when the domain is

Soldier, and NavyRanks when the domain is Seaman. Modeling this situation requires one of two choices. The domain of hasRank could be Seaman ∪ Soldiers, and the range could be ArmyRanks ∪ NavyRanks. The other option is to split the the hasRank property into hasNavyRank and hasArmyRank. Neither of these options faithfully expresses the same concept as the original concept.

1.1.2 OWL in the Wild

The theoretical interpretations and implications of domain and range constraints are clear and well studied (World Wide Web Consortium 2005b). However, in practice, the way OWL domain and range constraints are actually being used by ontology developers is surprising. A survey of over 200,000 semantic web documents retrieved by the Swoogle semantic web crawler showed that nearly 75% of the object properties were defined without property constraints (see Section 6.1).

There are many reasons why these properties are not constrained. First, the information may not have been known to the ontology developer. For example, if the ontology were developed incrementally, the property may be defined before the classes that make up the domain and range. Second, the lack of constraints may have been an artifact of the ontology generation processes used to construct these OWL documents. For example, suppose an ontology generator is used to automatically translate data from a non-semantic source. The ontology generator found evidence of a role but did not find evidence of any valid constraints. Other reasons could include: missing information, user error, or incomplete specification of the property itself. In some cases, it could be the intention of the ontology author that a particular property does not have a constraint.

Given an ontology containing a property with an unconstrained domain or range, is there a way to determine whether the property should have been constrained to something more specific than owl : Thing? The answer depends on the assumptions made about the

universe being described. The Open World Assumption states that what is not stated cannot be assumed to be true or false - it is unknown. The lack of a domain or range constraint neither implies that a property is really unconstrainable nor does it imply that there is some constraint that is not known. This is a very unsatisfactory position. Without external domain knowledge, an examination of an ontology cannot determine whether the lack of a constraint is intentional or not, or even if the constraints exist or not.

1.1.3 Generating Domains

The way properties are used can provide hints about which domain and range constraints could apply. This work proposes three different techniques to generate domain and range constraints:

- Disjunction,
- Least Common Named Subsumer,
- Vivification.

These constraint generation algorithms share a method of collecting evidence for the domain and range constraints for a property. First, the list of properties in the ontology is computed. For each property P in the list, each class definition in the ontology is evaluated to determine if it contains a restriction involving the property P. If it does, then the defined class is added to the domain of P and the restricted class definition is added to the range of P. Each of the generation algorithms use a different method to construct the final domain and range constraints for P. The disjunction method computes the least specific subsumer of the set of classes in the domain and range. The least common named subsumer computes the least specific named concept that subsumes all of the terms in the constraint list. The vivification algorithm computes a heuristic guided summary of the concept list.

A domain or range constraint can be computed from the source ontology with any of these three methods. Using the computed domain and range statements can add previously unstated information to an ontology. The problem is that each of these approaches creates different semantics for the generated domain and range constraints. These differences have an impact on the quality of the resulting reasoning and on the performance of the reasoner.

Another additional difficulty with this approach is determining whether the computed domain and range constraints match the intentions of the ontology's authors. Therefore, there needs to be a way to manage the derivation of new facts from the knowledge base and to efficiently retract information from the knowledge base if later assertions clash with the generated domain and range.

1.1.4 Default Reasoning

Default reasoning is a form of non-monotonic reasoning, in which facts are known by default or are assumed due to a lack of information to the contrary. In this line of research, the domain and range constraints are computed and used in the reasoning process as if they were assertions made by the author. After those constraints are computed, information is subsequently entered into the knowledge base. These statements are the result of default rules and may clash with current or future facts causing the knowledge base to become inconsistent. In order to correct the inconsistency, there must be a way to distinguish between facts that were asserted, default, or inferred. If the inconsistency is caused by facts that were either default facts or inferred from default facts then which should be removed from the knowledge base. This is a form of default reasoning.

Most default reasoning is undecidable. In an undecidable logic, there are well-formed valid formulae that cannot be proven to be correct. A simple strategy to avoid undecidability while supporting very limited default reasoning will be described. The strategy will consist of two principle parts: tracking the derivation of facts and supporting contrac-

tion. The inference rules will be modified to handle the presence of default rules and will propagate the origin of a statement in the database through the inference process. The consistency criteria of the reasoner will be modified to handle the case of inconsistency due to default facts. When a default statement causes the inconsistency a contraction operation will remove the necessary facts to restore the knowledge base to a valid state.

1.1.5 Summary

Although domain and range constraints are important assertions when developing an ontology, they are frequently overlooked. Because of this, there is a great deal of latent information in an ontology that cannot be inferred by the reasoner. Generating default domain and range constraints can help restore this information. Construction of an algorithm to do this will help maintain reasoner performance and manage the size of the knowledge base. Default assertions can lead to problems with monotonicity which are addressed through tracking whether a fact is asserted by the ontology author or default. Modifying the reasoner's inference rules to use this tracking information can lead to an efficient form of default logic with contraction while preserving consistency and decidability.

1.2 Thesis Statement

The purpose of this research is to investigate methods for generating domain and range constraints from its defining ontology and to evaluate the quality of this generation. This work will also investigate the default reasoning necessary to support generated constraints. A specific focus will be on management of the default facts in the knowledge base including tracking default facts and efficient retraction operations to restore consistency.

The expected outcome of this research is an algorithmic framework to generate and evaluate domain and range constraints. Another outcome is to use this algorithm to com-

pare the generated constraints against the asserted constraints in existing ontologies to assess the subsumption relationships between them. A third outcome is an inference procedure that will enable limited default reasoning to be added to the existing OWL inference rules to support these operations while maintaining the completeness, correctness, and complexity results for traditional OWL reasoning tasks.

The intended audience for this research includes ontology developers, especially those developing large ontologies where these rules can be used to iteratively improve their ontologies by generating constraints and using those constraints as guidance for asserted constraints. Another audience will be researchers using information extraction to generate ontologies from non-semantic sources where specific domain and range information is not readily available. One final audience for this research includes those interested in ontology integration where presence of concise and meaningful domain and range constraints may help identify overlapping concepts between two ontologies.

1.3 Dissertation Outline

The remainder of this document is organized as follows. Chapter 2 provides an introduction to Knowledge Representation and Description Logics. Chapter 3 introduces the semantic web language OWL. Chapter 4 describes domain and range constraint generation algorithms and their implementation. Chapter 5 describes the modifications of OWL inference rules to support limited default reasoning. Chapter 6 provides results demonstrating the performance of experiment. Finally, Chapter 7 provides a final discussion and suggestions for future research.

Chapter 2

DESCRIPTION LOGICS

Knowledge representation is the field of Artificial Intelligence that focuses on the design of systems that are capable of expressing knowledge about a particular domain. Reasoning is concerned with the creation of systems that discover new information through an inference process. Together, knowledge representation and reasoning are two of the most crucial issues in the development of intelligent systems. Representing human knowledge and using it to solve problems has been at the heart of the Artificial Intelligence field since its beginnings with the Dartmouth Conference (McCarthy *et al.* 1955) where the original researchers were interested in how a ". . . computer can be programmed to use a language." Generally, there are two parts of this problem: how to represent knowledge and how to use that knowledge to reason about the state of the world.

There are numerous approaches to solving this problem. Some approaches are based on crisp logics with well defined semantics using first-order logic. First-order logic provides clear semantics and sound inference mechanisms. Systems of this type include PROLOG, Otter, and SPASS, (Colmerauer 1993), (Mccune & Wos 1997), (Weidenbach *et al.* 2002). Other approaches include probabilistic approaches. These systems frequently use Bayesian reasoning to extend the semantics of the reasoner to handle uncertainty. Systems of this type include Bayesian Networks (Pearl 1988), Probabilistic Logic (Nilsson 1986),

P-CLASSIC (Koller, Levy, & Pfeffer 1997) and BayesOWL (Ding 2005).

Description Logics are a branch of crisp logics that are are being employed in the construction of the Semantic Web with the definition of standard languages such as DAML+OIL and OWL (Section 3.2.2). Description Logics were selected for a number of reasons. First, they have a long history dating back to the frames systems in the 1970s (Baader & Nutt 2003). Second, they include well-researched and articulated languages such as KAON2 (Motik). Third, they provide clear semantics in the definition of classes, property relationships between individuals, and type assertions on those individuals. Fourth, there are existing web applications based on DL such as UNTANGLE and FINDUR; and the existence of DL languages designed for the web such as OIL (Horrocks, McGuiness, & Welty 2003). Finally, there are also a rich set of reasoners, such as Pellet and FACT (Parsia & Sirin 2004), (Horrocks 1999). The combination of expressive logic with clear semantics and efficient reasoning make Description Logics well suited for their work in the emerging Semantic Web.

This chapter includes an introduction to Description Logics in Section 2.1, an overview of types of Description Logics in Section 2.2, and the structure of a Description Logic knowledge base in Section 2.3. An overview of the reasoning process and types of tasks supported by a reasoner are discussed in Sections 2.4 and 2.5. Finally, an overview of Default Logics is discussed in Section 2.6.

2.1 Description Logics

Description Logics evolved from prior work in Semantic Nets and Frame based systems. The prior systems provided mechanisms to represent the generality or specificity of a particular domain through IS-A links; but they typically lacked the ability to express other types of relations. In comparison, Description Logics allow much more general expressions

of relations between concepts (Nardi & Brachman 2003).

2.1.1 Overview

Description Logics are built from a set of classes (concepts), relationships (roles or properties), and instances (individuals). Atomic concepts and roles are used to represent some principle concept in the domain or a binary relationship between them. More complex terms are built using operators such as *intersection*, *union*, *complement*, and *value restrictions* that combine atomic and complex terms together to define new classes. Figure 2.1 shows an example of a pair of concept descriptions in a Description Logic. The two definitions use the intersection operator to combine existing atomic classes to create new complex classes. The first represents the intersection of Person and Professor and represents all people who are professors. The second example defines a class where everyone is a Person and not Male and are Parent, which describes the concept of a mother.

$$\mathsf{Person} \sqcap \mathsf{Male}$$
$$\mathsf{Person} \sqcap \neg\mathsf{Male} \sqcap \mathsf{Parent}$$

FIG. 2.1. DL Example

Description Logics are a subset of First Order Logics (FOL). Like FOL, there is no inherent semantic meaning in the names of the symbols. The convention is to select a symbol whose name bears some relation to the concept it represents. In Figure 2.1, the symbol Person can represent anything. It is a recommended practice to select class names that reflect the meaning of what they represent.

Changing the name of the symbol does nothing to change the truth of the relationships between them. In the example of Figure 2.1, the symbol Person could just as easily

be named some randomly generated name, such as `GEN3233`. The semantic interpretation would be unaffected by this change. The semantic interpretation is defined by the organization of the classes and the relationships that connect them together. These relationships are used to make inferences about new relationships between the class definitions and to infer type assertions for individuals in the knowledge base. A pair of isomorphic graphics with different symbol names have identical semantics. The correspondence between symbol names and what they represent is primarily a tool to make it easier for human knowledge workers to maintain the knowledge base (Nardi & Brachman 2003).

2.1.2 Parts of a Description Logic

Classes (concepts) can be atomic, consisting of a single term, or can be be defined as a set of value restrictions involving other concepts or roles. Atomic classes are usually implicitly defined merely by referencing them in other class definitions or through type assertions. More complex classes can be created through class definition constructors.

Individuals are specific instances of classes. Individuals represent specific instances of some set of classes. Suppose there is a class `Car` that describes all cars, then an individual `YellowCar` represents a specific car. Each individual may have a number of class assertions that describe different aspects of the individual. For example, the individual `YellowCar` could be a member of the class `Car` and the class `JohnsThings`, the collection of things owned by John.

Properties are used to define relationships between individuals or between an individual an literal data such as a string or a number. A role assertion has an individual that is a subject and another that is the object. These individuals are often called fillers. The property assertion acts as the predicate that ties the subject to the object. Together properties and individuals define the semantic structure of a specific instance of a knowledge base (Nardi & Brachman 2003). For example, a property `hasDoors` could connect the individual

`YellowCar` to the value 4.

Different Description Logics may further extend the basic property constructor with additional definitions. One common set of extensions include domain and range constraints. Another common extension is a statement that a property has an inverse property. Properties can also be identified as being transitive, symmetric, and functional (Nardi & Brachman 2003).

Domain and Range Constraints Domain and range constraints are of particular interest to this research. The domain and range constraints assert the types of the fillers of a property. The subject filler is asserted to be a member of the class of the domain, and the object filler is asserted to be a member of the range. This use of constraints is different than in other applications, such as databases, where the domain and range constraints are interpreted as restrictions on the types of values that allowed to be used with that property.

For example, using traditional Object-Oriented Programming languages such as Java or C++, a function may be declared to accept an object of class `C`. In essence, the domain of this function is `C`. Calling the function with an object of any class that is not `C` or one of its sub-classes will result in either a compilation error or a run-time type exception. If the instance is a sub-class of `C`, it will be type-cast to type `C` and will, temporarily, lose all essence that is not part of class `C`.

The Description Logic interpretation of domain and range constraints is that they are sufficient conditions to add a type definition to the subject or object individuals that fill a property role assertion. Thus, for some property that has a domain of type `C`, any subject of that property becomes a member of the class `C`, or the ontology becomes inconsistent if that causes a class with other assertions.

Other Constructors Most Description Logics provide two class definitions constructors, a subclass constructor, $\sqsubseteq$, and an equivalence constructor, $\equiv$. The difference is in the interpretation of the relationships. The subclass constructor defines an inheritance relationship between the super- and sub-classes. The sub-class inherits the properties of the super-class and any instance of the sub-class necessarily has the properties of the super-class. The equivalence relationship is a stronger assertion and describes necessary and sufficient criteria for the two classes. In effect, it is sufficient to know that an individual is a member of one class to know that it is a member of the other class as well.

Classes are also constructed using restrictions. Restrictions are descriptions of sets of individuals that meet the criteria of the restriction. Common restriction types include existential restrictions, $\exists$, value restrictions, $\forall$, and cardinality restrictions, $\leq$, $=$, $\geq$.

Let `hasChild` be a property that represents that two individuals that are related to each other such that one is the parent and the other is the child. For example, $\texttt{hasChild}(\texttt{x}, \texttt{y})$ states that x is the parent of y.

An existential restriction on `Parent` could be $\exists \mathsf{hasChild}.Person$. This states that for each `Parent` there exists at least one individual that has type `Person` and is related to the `Parent` through the `hasChild` relationship.

The existential restriction does not mean that the y in the $\texttt{hasChild}(\texttt{x}, \texttt{y})$ assertion above is a `Person`. There could be other individuals that fill the object of the property. This assertion says that there must be at least one, possibly unknown, individual that fills this property.

A value restriction could be, $\forall \mathsf{hasChild}.Person$. This defines a set, or sub-class, that states that for all members of this class, they are subjects of the `hasChild` property, and any objects they are connected to through this role must be of type `Person`. This does not mean that there *are* any fillers of this property. It just states that *if* there are *then* they must be of type `Person`.

$$\texttt{ParentOfOne} \sqsubseteq \mathsf{Person} \sqcap \forall \mathsf{hasChild}.\mathsf{Person} \sqcap \exists \mathsf{hasChild}.\mathsf{Person} \sqcap = 1\mathsf{hasChild}$$

FIG. 2.2. Equivalent OWL Encodings

A cardinality restriction could be, $= 1\mathsf{hasChild}$. This states that there is exactly one one filler of this class. Cardinality restrictions are used to describe the sizes of the sets of fillers for a particular class.

Figure 2.2 combines these class constructors to define a new class, `ParentOfOne`. This class represents the subclass of all things that are in the set `Person`, and who only have children that are also in `Person`, and who have some child that is a `Person`, and who have exactly one child. This demonstrates the capability of a Description Logic to build complex terminologies from simple constructors.

2.2 Types of Description Logics

Brachman and Levesque first demonstrated that the expressiveness of the DL drove the runtime of reasoning and that for certain DL subsets, subsumption queries can be computed in polynomial time (Brachman & Levesque 1984). Their work showed that the computational complexity of the reasoner can be controlled through careful selection of the language constructs. They showed that for basic Description Logics, a reasoner could be constructed which operates in polynomial time, while adding that other more expressive constructors can move reasoning with the DL to worst-case NP time and space. This was one of the most important contributions to the field.

Because of the complexity results of Brachman and Leveseque, different Description Logics can define a set of constructors and operators to allow expression of particular types of relationships. For example, while all Description Logics allow the definition of subclass

Table 2.1. Description Logic notation and meaning.

Symbol	**Meaning**
$\mathcal{AL}$	Attributive Language
$\mathcal{U}$	Union of concepts
$\mathcal{E}$	Existential quantification of roles
$\mathcal{N}$	Numeric restrictions (cardinality constraints)
$\mathcal{C}$	Negation of concepts
$\mathcal{I}$	Inverse of roles
$\mathcal{R}$	Intersection of roles
$\mathcal{S}$	Stands for $\mathcal{ALC}_{\mathcal{R}^+}$, *i.e.* extension of $\mathcal{ALC}$ with transitive roles
$\mathcal{H}$	Hierarchy of roles
$\mathcal{O}$	Ability to define class by enumerating its instances
$\mathcal{Q}$	Qualified number restrictions
(D)	Support for primitive datatypes (*e.g. integer, string*)

relationships between classes, not every language supports functional properties or cyclic terminologies. The set of operators supported by a particular language determines the reasoning properties of the language. Careful selection of these operators allows a DL to maintain the necessary expressiveness for a given application domain while maintaining the most efficient inference procedures.

A Description Logics is commonly identified by the constructors that it supports. The naming convention uses a symbol for each of the constructors. The DL name is the concatenation of these symbols. Table 2.1 contains the set of common symbols and their meaning (Nardi & Brachman 2003), (Baader & Nutt 2003), (Horrocks, Patel-Schneider, & van Harmelen 2003). Description Logics are all subsets of the family of *Attributive Language*, noted by the standard symbol $\mathcal{AL}$. Each extension is noted by appending a symbol on $\mathcal{AL}$, connecting the DL name with its set of extensions. One set of DL extensions is quite common, and is identified as $\mathcal{S}$, which stands for the language $\mathcal{ALC}_{\mathcal{R}^+}$.

Using the symbols from Table 2.1, the Description Logic $\mathcal{ALUC}$ would support the base *attributive language*, *union of concepts*, and *cardinality constraints*. The language

$\mathcal{SHIQ}$ extends $\mathcal{ALC}_{\mathcal{R}^+}$ with role hierarchies, inverse roles, and qualified number restrictions.

These extensions add expressiveness through inclusion of different constructors that allow additional relationships to be defined. Different Description Logics have more in common than differences, despite the variance in linguistic combinations. Despite their differences, all Description Logics must provide a way to define terminological concepts and roles; assert instances of those concepts; store those assertions in a knowledge base (KB), and perform inferences using that KB (Nardi & Brachman 2003).

One noteworthy consequence of the dependence of reasoning complexity on expressivity is that complexity is independent of the actual language used to represent the constructors. A serialization of a Description Logic describes the language used to tell the knowledge base new facts. The serialization can be based on traditional logic formulae or be specific to an application such as FACT (Horrocks 1999). As long as the serialization has some form for each of the supported DL constructors it has no effect on the expressivity of the DL or the reasoning hardness. In many cases, an ontology expressed in one serialization can be translated to another serialization of the same Description Logic. This is an important consequence for the development of the Semantic Web languages. The semantics are determined by the DL and the reasoner, not the serialization.

2.3 The *T-Box* and *A-Box*

There are two principle components of a Description Logic knowledge base. The *T-Box* is composed of the intensional knowledge in the form of terminological descriptions of the domain. The *A-Box* is composed of the extensional, assertional knowledge specific to individuals (Nardi & Brachman 2003). The syntax of these assertions depends on the specific DL (and software) being used to implement the knowledge base (KB).

T-Box Defintion		
Professor	$\equiv$	Person $\sqcap$hasGradStudent $\leq 1 \sqcap \exists$hasStudent.Student
Student	$\sqsubseteq$	Person
hasStudent	$\leftrightarrow$	studentOf
hasGradStudent	$\sqsubseteq$	hasStudent
hasGradStudent	$\leftrightarrow$	isGradStudentOf
A-Box Defintion		
Professor(JONES)		
Student(ABEL)		
Student(BELL)		
isGradStudentOf(ABEL, JONES)		
isGradStudentOf(BELL, JONES)		

FIG. 2.3. DL knowledge base for a simplified University domain.

Figure 2.3 shows a knowledge base that defines a highly simplified university domain. The language uses Tarski-style logic sentences (Stanford Encyclopedia of Philosphy 2008). The *T-Box* explicitly defines two concepts: Professor, and Student. The atomic concept Person is implicitly defined as a result of being on the right hand side of the Professor and Student definitions. Four roles: hasStudent, studentOf, hasGradStudent, isGrad-StudentOf are also defined. The concept Professor is defined to have the necessary and sufficient properties of: a Person, having at least one student (hasStudent), and having 0 or 1 graduate students (hasGradStudent). The concept Student is defined as a subclass of Person. The role hasStudent is an atomic role and defined to have an inverse role of studentOf. The role hasGradStudent is a sub-class of the hasStudent relationship, and has an inverse role of isGradStudentOf. The *A-Box* makes five assertions: JONES is a Professor, ABEL and BELL are Students; and ABEL and BELL are graduate students of JONES. The seeming inconsistency of ABEL and BELL being students of JONES will be explored in Section 2.4.2.

Using the naming conventions of Table 2.1, the minimum DL that can express this ontology is $\mathcal{ALEN}^+$. This DL contains: *Attributive Language*, with *Existential Quantifi-*

cation, and *Cardinality Restrictions,* with *Inverse Roles*.

2.4 Basic Reasoning Tasks

The primary task of a Description Logic is to support the classification of objects within a hierarchy of concepts (Baader & Nutt 2007). The constructs of the DL are used to define concepts and roles and to make assertions regarding individuals within the domain. Automated reasoners are able to use the stated assertions and terminologic statements to infer new facts that were not explicitly specified during the KB construction. Reasoners typically answer two different types of queries: *subsumption* queries against the T-Box, and *instance checking* queries against the A-Box.

Subsumption A concept C is subsumed by a concept D with respect to T-Box T if $C^I \subseteq D^I$ holds for every model I of T. For example, we could use the KB in Figure 2.3 to verify that $\mathsf{Student} \sqsubseteq \mathsf{Person}$. Subsumption queries are used to answer other types of queries, including: *satisfiability*, *equivalence*, and *disjointness* (Baader & Nutt 2003).

Satisfiability Let C be a concept, and $\perp$ represent the null concept. A concept C is satisfiable (denoted $Sat(C)$) if C is not subsumed by the null concept:

$$Sat(C) \Longleftrightarrow \neg(C \sqsubseteq \perp)$$

■

Equivalence Let C and D be two concepts in the domain. The concepts are equivalent (denoted $C \equiv D$) if:

$$C \equiv D \Longleftrightarrow C \sqsubseteq D \text{and} D \sqsubseteq C$$

■

Disjointness Let C and D be two concepts in the domain. The concepts are disjoint (denoted $C \cap D = \emptyset$) if:

$$C \cap D = \emptyset \iff C \cap D \sqsubseteq \bot$$

■

Using these definitions, we are able to answer questions about the T-Box of the KB. For example, we can check if different concepts are satisfiable:

Example Check concept satisfiability by determining if the following is true.

$$Sat(\mathsf{Professor}) \iff \neg(\mathsf{Professor} \sqsubseteq \bot)$$

The satisfiability of class Professor is true so long as there can be at least one member of the class. Most reasoners implement this check by creating an anonymous individual and executing the reasoning algorithm until the reasoner is complete or there is an inconsistency created. In this case, with no further evidence the class Professor is satisfiable because there is no other evidence to show that it is not.

This set of definitions can also be extended to answer queries about instances in the A-Box. For example, $instanceOf(\mathsf{JONES}, \mathsf{Person})$ is true, because JONES is an instance of Professor, and Person subsumes Professor.

Using these definitions, the worst-case efficiency of the algorithms can be computed as the cost for subsumption checking and developers of knowledge-based systems can implement systems using only a subsumption operator. One of the most useful contributions of Description Logic is that the reasoner can be created using only a set of transformation rules which are dependent on the expressivity of the DL. Extending these rules enables the

same reasoner to work on more expressive knowledge bases. Due to the regularity of these rules, many heuristics and optimizations have been developed to exploit common patterns in ontologies to further improve the average case performance of the reasoner.

2.4.1 Open World Assumption

The Open World Assumption is simply stated as anything that isn't asserted in the KB (or that cannot be inferred) is considered to be unknown. Most implementations of Description Logic reasoners choose the Open World Assumption (OWA). The Open World Assumption is a departure from traditional databases and some other knowledge-based languages like Prolog that make the Closed World Assumption. Any fact left unstated in the Close World Assumption model is assumed to be false. This is frequently described as *Negation as Failure*.

Suppose some traditional database contains a relation of parents and children. There is a row indicating that ALICE is the parent of BOB, and there are no other rows for ALICE. Under the Close World Assumption, selecting parents with more than one child would not include ALICE. This is due to the lack of additional entries for ALICE in the database. In this case, not knowing about any other children for ALICE is equivalent to the case where ALICE is known to have no more children.

Now consider the same example using the Open World Assumption. The knowledge base contains the assertion hasChild(ALICE,BOB). This assertion states that of all possible worlds, the only valid worlds are those in which ALICE has a child BOB. Any possible worlds in which ALICE has only BOB as her child are consistent, as are any worlds in which ALICE has more than one child. Thus, asking the KB if ALICE is $InstanceOf(= 1(hasChild.Person))$ returns false, indicating that there maybe worlds where ALICE has one child, but there may also be ones where she has more. Because it is not true in *all* worlds, then it is not true.

2.4.2 Unique Name Assumption

One other area where DLs depart from other more familiar systems is that they do not make the Unique Name Assumption (UNA). The Unique Name Assumption allows two individuals with different identifiers to be assumed to be two different individuals.

The definition of the university knowledge base in Figure 2.3 seems to contain a contradiction regarding the number of graduate students assigned to JONES. The knowledge base contains assertions that JONES is a Professor, which carries with it the necessary criteria that he have no more than one graduate student. The KB also asserted that ABEL and BELL satisfy the isGradStudentOf relation for JONES.

If there are no possible models of worlds where every fact is true then the KB is inconsistent and the reasoner stops. The ABox of Figure 2.3 does have a possible model that is consistent with the assertions in the knowledge base. Due to the lack of the Unique Name Assumption, it is possible that individuals ABEL and BELL are two different symbols for the same individual. Without UNA, the reasoner cannot automatically infer that ABEL $\neq$ BELL, and with the Open World Assumption, it cannot be concluded that this is or is not the case.

Description Logics frequently define operators to assert that specific concepts or individuals are not the same. For example, the university KB of Figure 2.3 can contain an assertion that ABEL is the same as BELL or that they are disjoint. Making extensive use of these operators, to explicitly close possible unique names, can degrade reasoner performance and cause quadratic growth in the size of the knowledge base. For a class of n individuals, to express that all individuals are distinct we need $n(n-1)/2 = O(n^2)$ disjoint statements.

2.5 Tableau Reasoners

Prior to the emergence of tableau algorithms, reasoners such as KL-ONE (Schmolze, Beranek, & Inc 1985) and KRYPTON (Brachman *et al.* 1985) relied on structural subsumption algorithms (Baader & Sattler 2001). Structural algorithms rely on computing normalized forms for each of the terms in the knowledge base. Subsumption checking is simply a comparison of the normalized forms of the two concepts (Horrocks 2003). Structural subsumption algorithms are generally decidable but frequently incomplete and difficult to extend to expressive DLs (Horrocks 2003). Tableau algorithms were originally proposed by Schmidt-Schuabß and Smolka in 1991 to address problems with structural subsumption algorithms (Schmidt-Schaubß& Smolka 1991).

Tableau algorithms operate by constructing a *tableau graph* where nodes represent the individuals of the graph, and directed edges indicate relationships between them (Horrocks 2003). The reasoner applies a set of expansion rules to the tableau. The algorithm terminates when there are no remaining expansions or when a clash is detected.

There are a number of optimizations and simplifications which can be applied to improve the performance of a tableau reasoner, including: acyclic definitions in the TBox, and an unfolding operation. The most important factor in the performance of the tableau reasoner is the set of consistency-preserving transformations that are available. The transformations that are selected represent the logical constructs of the Description Logic. Unlike structural subsumption algorithms, the tableau reasoner is easily extended through the additions to set of transformation rules (Horrocks 2003).

2.5.1 Acyclic TBox

Multiply Defined Class When a terminology contains multiple, partial definitions for a class, that class is said to be a multiply defined class. Let C, D be distinct class descriptions.

Let class A be defined with two statements, $A = C$ and $A = D$. Then class A is a multiply defined class (Baader & Sattler 2001).

Cyclic Definition When a terminology contains a definition that depends on itself, it is said to be a cyclic definition. For example, suppose there is a set of concept definitions in a TBox: $A_1 = C_1, A_2 = C_2, \ldots, A_n = C_n$ where A_i occurs in $C_{i-1}, (1 < i \leq n)$ and A_1 occurs in C_n. Then this is a cyclic definition. (Baader & Sattler 2001).

Acyclic Terminology A terminology (TBox) is an acyclic terminology if and only if it is a set of concept descriptions that neither contains multiple definitions of the same class or cyclic definitions.

Ensuring that a TBox is acyclic is important in order to guarantee completeness of the reasoner. Using an acyclic TBox allows the reasoner to use an unfolding operation to collapse the terminologic definitions into instance assertions and to perform all reasoning on those instance assertions. The result is a much more efficient reasoner and an avoidance of the hardness associated with allowing arbitrary TBox constructs.

2.5.2 Unfolding

Unfolding Unfolding is a process that replaces references to defined concepts with their definitions. If the TBox is acyclic, then all defined concepts can be unfolded into ABox assertions of atomic concepts (Horrocks 2003).

Using unfolding with acyclic TBoxes eliminates the need to do any reasoning with the TBox. By applying unfolding rules such as those shown in Figure 2.4, the statements in the TBox such as equivalence and subsumption can be applied to individuals in the ABox (Horrocks 2003). In that figure, $\mathcal{A}$ represents the ABox, $\mathcal{T}$ represents the TBox, A is an atomic, defined concepted, C is a defining concept.

$U_1-\text{rule}$ if 1. $\mathcal{A}$ contains $A(x)$ and $(A \equiv C) \in \mathcal{T}$
2. $C(x) \notin \mathcal{A}$
then $\mathcal{A} \rightarrow \mathcal{A} \cup \{C(x)\}$
$U_2-\text{rule}$ if 1. $\mathcal{A}$ contains $\neg A(x)$ and $(A \equiv C) \in \mathcal{T}$
2. $\neg C(x) \notin \mathcal{A}$
then $\mathcal{A} \rightarrow \mathcal{A} \cup \{\neg C(x)\}$
$U_3-\text{rule}$ if 1. $\mathcal{A}$ contains $A(x)$ and $(A \sqsubseteq C) \in \mathcal{T}$
2. $C(x) \notin \mathcal{A}$
then $\mathcal{A} \rightarrow \mathcal{A} \cup \{C(x)\}$.

FIG. 2.4. Unfolding Rules

2.5.3 Consistency Preserving Transformations

Given a KB with an acyclic TBox which has been unfolded using the previous unfolding transformations, then the tableau subsumption algorithm can proceed using only the ABox. The transformation rules for a tableau reasoner that implements the Description Logic $\mathcal{ALC}$ is shown in Figure 2.5 (Baader & Sattler 2001). The reasoner works by applying these rules and expanding the ABox until no further rules can be applied or until a clash is detected.

The greatest challenge of this algorithm is in the $\sqcup-\text{rule}$, where disjunction creates non-determinism with respect to which fact to add to the ABox. The non-determinism caused by this rule is usually handled through search with backtracking. When the rule is applied, it opens a set of ABoxes. For each ABox created in this step, reasoning continues until a clash occurs, then the reasoner backtracks and tries the next ABox in the set. Thus, disjunction is the primary source of complexity and performance degradation in these transformation rules and the cost of disjunction becomes a major motivation for this research.

The rules shown in Figure 2.5 are a starting point for a reasoner. More expressive Description Logics can be built by creating more expressive transformation rules. This

$\sqcap-$rule	if 1.	$\mathcal{A}$ contains $(C_1 \cap C_2)(x)$,
	2.	$C_1(x) \notin \mathcal{A}$ or $C_2(x) \notin \mathcal{A}$
	then	$\mathcal{A}' \rightarrow \mathcal{A} \cup \{C_1(x), C_2(x)\}$
$\sqcup-$rule	if 1.	$\mathcal{A}$ contains $(C_1 \cup C_2)(x)$,
	2.	but neither $C_1(x)$ nor $C_2(x)$
	then	$\mathcal{A}' \rightarrow \mathcal{A} \cup \{C_1(x)\}$
	and	$\mathcal{A}'' \rightarrow \mathcal{A} \cup \{C_2(x)\}$
$\exists-$rule	if 1.	$\mathcal{A}$ contains $(\exists R.z)(x)$
	2.	but no individual z such that $C(z)$ and $r(x, z)$ are in $\mathcal{A}$
	then	$\mathcal{A}' \rightarrow \mathcal{A} \cup \{C(z), r(x, z)\}$
$\forall-$rule	if 1.	$\mathcal{A}$ contains $(\forall R.C)(x)$ and $r(x, z)$
	2.	but not $C(z)$
	then	$\mathcal{A}' \rightarrow \mathcal{A} \cup \{C(z)\}$

FIG. 2.5. $\mathcal{ALC}$ Transformations

provides an efficient mechanism for the designer of a Description Logic to manage the trade-offs between expressibility and reasoning performance.

2.6 Default Logic

Default logic refers to a form of logic where facts that are not explicitly described in the knowledge base are held to be true by default. Consider the following thought experiment: suppose you were to model a class named ‘Tiger.’ What restrictions would you define? Where would you place it in a taxonomy? By default, most people assume that members of ‘Tiger’ have stripes, four legs, sharp claws, and eats people. Now, suppose you were given additional information, namely that ‘Tiger’ is superseded by ‘Leopard’ and is made by ‘Apple’. Now, our mental picture of the properties of ‘Tiger’ is suddenly changed to be relevant to an operating system. We must retract our previous beliefs about ‘Tiger’ and reshape them to the operating system sense of the word.

Default Logic plays an important role for this research. Default Logic describes the process used to represent these default facts and how inference can combine these results

together during reasoning. Default logic also must address the contraction of the knowledge base to restore the KB to a consistent state.

2.6.1 Monotonicity

Monotonicity of a logic means that satisfying a valid formula can only result in the addition of new information. A monotonic logic cannot invalidate prior information as a result of adding new facts to the knowledge base. If it is possible to add a new assertion to the knowledge base that invalidates a prior assertion, then the logic is non-monotonic (WWW Consortium 2004).

As an example of monotonicity, consider an example where, by default, all dogs chase cats. This default rule is a stand-in for asserted data. Until information is stated about the instances, this rule is true. In addition to this default rule, the knowledge base contains assertions that Fido is a dog and Fluffy is a cat. This will allow the reasoner to infer, by default, that Fido chases Fluffy. Later, the statement ‘Fido is a Poodle’ is made. This is still consistent with the knowledge base defaults so far. But what if the fact that Fluffy is a Tiger (the type with sharp claws) is added to the knowledge base. At this point, it is no longer likely that Fido chases Fluffy, or if so, not likely for very long. Due to monotonicity, the assertion that Fido chases Fluffy must be retracted. This is evidence of a non-monotonic logic. The rule that dogs chase cats was a default rule. This default is considered to be true until more information became available. With the addition of new information there are instances that are no longer consistent with the ontology - not all dogs chase cats. By including this default statement in the ontology the reasoner must now deal with the potential clashes caused as a result of adding new known information that contradicts default statements.

2.6.2 Contraction

Contraction is an operation for restoring consistency to the knowledge base by removing the facts that create the inconsistency (Antoniou & Williams 1998). This can be a non-trivial process when the the facts that caused the inconsistencies are part of a long chain of dependent facts. Simply removing a fact may leave other parts of the knowledge base unjustified. Contraction is frequently implemented as part of a larger belief revision system for default logics. The typical operation is to find some formula, ϕ that is part of the inconsistent ontology. After ϕ is found, the set of formulae that depend on ϕ are identified. Both ϕ and its dependences are retracted from the knowledge base. New statements are added to prevent ϕ from being reintroduced by the re-application of default rules (Antoniou & Williams 1998).

Contraction is generally implemented as part of a strategy for dealing with the non-monotonicity introduced through the inclusion of default logic. When a default fact must be removed from the knowledge base, the contraction operation is used to remove only the necessary facts to restore consistency in the knowledge base. The challenge is to keep the cost of contraction to a minimum and to remove a minimum number of facts.

Different forms of the contraction operation are described in the literature, such as (Antoniou & Williams 1998), (Colucci *et al.* 2004). These approaches are based on the need to carefully remove default facts from the knowledge base under the assumption that the cost of rebuilding them is higher than the cost to remove them.

Contraction is an important component of reasoning with default logics. It is used to manage the removal of facts to restore consistency to the knowledge base. Contraction allows the reasoner to effectively handle the issues of non-montonicity.

2.7 Relationship to Rule Languages

Description Logic languages, such as OWL, have been extended to include support for rule-based languages, such as SWRL (Ian Horrocks 2004). SWRL is a rule language that defines rules as pairs of antecedents and consequents. The rule is interpreted as any time the antecedent is true, then the consequent is true as well.

For example, a SWRL rule takes on the form, $\mathsf{Person}(?p) \sqcap \mathsf{hasDog}(?p, ?d) \rightarrow \mathsf{Dog}(?d)$. This rule is triggered whenever there are objects, p and d, where $\mathsf{p} \in \mathsf{Person}$, and p and s are, respectively, the subject and object of the predicate hasDog. When the rule is triggered, then the type assertion, Dog is added to individual d.

The previous example of a SWRL rule does not modify the terminological descriptions of the ontology and thus conforms to the Open World Assumption. New individuals may be added that fill the object of the hasDog relationship, but they will not be typed as a Dog unless the rule is re-executed. This rule demonstrates that SWRL rules may obey the Open World Assumption and monotonicity property of Description Logics.

SWRL does support more expressive extensions, such as procedural attachment, that do violate the monotonicity property. Procedural attachment is a type of rule that uses an external procedure to compute some value to be entered into the knowledge base. The computation is made with knowledge that is external to the representation in the ontology. The procedural attachment represents a portion of the terminological construction that exists in the world but is not expressed in the ontology. As such, this violates the Open World Assumption and leads to non-monotonicity (Ian Horrocks 2004).

2.8 Conclusion

Description Logics are a family of logics that are conducive to the development of the Semantic Web. Description Logics have a trade-off between expressivity and reasoning

complexity. This allows the construction of specialized Description Logics that can provide efficient reasoning over large ontologies at the cost of weaker expressivity. This is essential for the future development of the Semantic Web which will create a global knowledge base. This facet of Description Logic is is of interest to this research, especially the efficiency of the reasoner with disjunctions.

Default Logic is another family of logic that, unlike Description Logic, support reasoning over default rules. These logics play an important role in reasoning with data derived from default rules. Description Logics may be extended to include certain aspects of Default Logic. In the event that default rules conflict with the asserted information in the knowledge base then there are operators such as contraction that can be used to restore consistency.

Both forms of logic are of interest to this research which depends on using the terminological evidence in a Description Logic to infer default statements about other parts of the ontology. The default logic will provide useful operations to handle any inconsistencies that arise as a result of this default knowledge generation.

Chapter 3

SEMANTIC WEB

The World Wide Web has revolutionized the way we share information. The information published on the Web is organized for human comprehension and visual appeal, with information embedded in document formatting and natural language. To get order from this markup chaos, the Extensible Markup Language (XML) was developed to create a strict information markup language. XML has become the enabler of the Web 2.0 revolution and the interactive web. While XML gives structure and semantics to the data, it does not give the necessary structure for automatic, shared understanding of the information represented in the document.

The Semantic Web is being built from a collection of technologies that enable information to be exchanged between machines. At the core of these technologies is the next generation of mark-up languages that are based on formal logic languages. Automated reasoners are able to use the formal semantics and clearly encoded knowledge of these semantic languages to perform reasoning and semantic queries on a collection of documents. Using the Web's paradigm of linked documents, the Semantic Web will operate over the global knowledge base. As simple as this idea sounds, there are a number of challenges to answer before the greater benefits of this technology are fully exploited. This chapter presents a brief overview of the Semantic Web in Section 3.1, describes standard repre-

sentation languages Section 3.2, and the challenges facing Semantic Web development in Section 3.3.

3.1 Overview of the Semantic Web

For many, the "Internet" began in August 1991, when CERN introduced the World Wide Web project. The World Wide Web made the Internet user-friendly and accessible to the general population. The World Wide Web represents a tremendous leap forward in human communication. People use it to perform everyday tasks including communication, banking, and shopping. In January 2006 an estimated 1 billion people world-wide used the Internet. This figure represents 15.7% of the global population (Internet World Stats 2006).

The web was initially envisioned as a way for humans to communicate and as a platform for software agents to carry out tasks for their users. The information published on today's web is mainly for human consumption. Web pages contain large amounts of document formatting markup to control how the page is displayed on a screen. A large number of documents contain structural errors that can make it difficult for machines to parse the content of the page (Google 2006). Much of the information published on the web is actually content that was originally generated from a database. Even though the information originally was stored in highly normalized and structured databases the semantics are lost when the data is embedded in the formatting of the web page (Berners-Lee 1998).

The encoding of information into a browser markup makes it difficult for automated processes to extract meaningful information from a web page. Systems typically resort to customized text-extraction and mapping algorithms to find knowledge from web pages (Cimiano & Völker 2005), (Abiteboul *et al.* 1997) and (Mooney & Bunescu 2005). Unfortunately, these systems have great difficulty overcoming the problems created by irregularity of natural language. If the information is encoded in an unexpected manner the

information will be unusable. As a result, using exactly the same set of web pages, different extraction algorithms are likely to produce different descriptions of the world. Natural language processing is a hard task and is not likely to be solved in the near future (Russell & Norvig 2003).

Berners-Lee, the inventor of the World Wide Web, proposed the Semantic Web as a way of publishing information specifically for machine consumption. The Semantic Web uses markup languages, such as OWL, that express knowledge in a machine readable format. These languages define fixed semantics to describe concepts and their relationships to each other (Berners-Lee 1998). Automated reasoners rely on the fixed semantics and asserted knowledge to infer relationships between concepts. Use of languages designed for the Semantic Web enables document linking much like the World Wide Web. In the Semantic Web, linked documents allow the sharing of a common description of some part of the world. This common set of semantics make an ontology that will allow reasoners to work over large collections of documents spread throughout the world.

The Semantic Web has the potential to revolutionize the way humans use computers. Instead of using them to communicate with each other, the Semantic Web promises to usher in a new generation of intelligent agents. The expectation is these agents will be able search the internet for knowledge, not substrings; communicate with each other, not just other humans; and understand the intention of the task they are carrying out, not just the statement of it.

3.1.1 Growth of the Semantic Web

There has been considerable growth in the number of Semantic Web Documents (SWDs) published on the Semantic Web. In January, 2006, there are over three hundred thousand SWDs defining over four thousand ontologies, almost one hundred thousand classes, and over seven million triples in the Swoogle database (Swoogle 2006). In

comparison, there are over nine billion pages in Google's database (Google 2006).

One trend stands out in Swoogle's statistics. The number of documents in Swoogle's database showed almost no growth throughout 2005 (Swoogle 2006). This is contrary to the exponential growth in the number of registered users of the Protege tool, a popular ontology editor (Stanford Medical Informatics 2006). One plausible explanation is that many of the new ontologies are being used internally, or being embedded in information systems that aren't publicly accessible, that is, they are part of the 'deep web.'

The wide-spread adoption of the Semantic Web is delayed by a more fundamental factor: cost. Presently, highly trained knowledge engineers are employed to construct the high quality ontologies for use on the Semantic Web. This work requires training with logic and reasoning and domain knowledge. Some skeptics claim that in its current form, the Semantic Web "may work well for targeted vertical applications where there is a built-in economic incentive to support expensive mark-up work (such as biomedical information), such a labor-intensive platform will never scale to the Web as a whole" (Wright 2008).

There is a well-established need for tools to help automate this process. Automating the process of generating ontologies will help reduce the cost of publishing semantic data and help open the creation process to a large audience. This research describes a process that generates property constraints based on the terminology of a knowledge base. Future tools like this may help encourage acceptance of the Semantic Web and lead to a fulfillment of Berners-Lee's vision.

3.2 Semantic Web Languages

The World Wide Web Consortium (W3C) (W3C 2008) is an international consortium dedicated to maintaing the standards used on the World Wide Web. This body is responsible for maintaining standards for languages such as the Extensible Markup Language (XML)

```
<datebook_entry>
   <date>Mar 3, 2005, 10:30am EST</date>
   <title>Dentist Appointment</title>
   <descr>Go to the dentist for a check-up</descr>
</datebook_entry>
```

FIG. 3.1. Example of an XML date book entry.

and Resource Description Framework (RDF). The W3C also maintains the standard for the Web Ontology Language (OWL). OWL has become the *de facto* standard Semantic Web Language. OWL is based on an encoding that uses the RDF and XML language to markup data and constructors borrowed from Description Logics. This section explores the roots of the OWL language and provides an overview of its major constructs.

3.2.1 Base Languages

The Semantic Web is based on a pair of languages, the Extensible Markup Language (XML) and Resource Description Framework (RDF). They are used in a wide range of applications beyond the Semantic Web. For example, XML is used by a growing number of applications to interchange data, and it is used as a base language for the emerging XHTML markup language that is set to replace HTML.

Extensible Markup Language The Extensible Markup Language (XML) was developed as a data interchange language (World Wide Web Consortium 2005a). A sample appears in Figure 3.1. This sample describes an entry from a date book that includes the time and a brief description of the event. This representation is only useful if some program is coded to interpret the semantics of this entry. A typical system that uses XML to exchange data requires a manual mapping between the XML and that application.

Really Simple Syndication (RSS) is an XML based mark-up that allows the encoding

of news feeds (Wikipedia 2008). A developer must map source data to the RSS format. This allows applications to display those news feeds in a variety of fashions. For example, the RSS format specifies concepts such as feeds and articles. It also specifies components of each concept such as titles and descriptions.

XML does not satisfy the needs of the Semantic Web. XML does not enforce a consistent schema for assertion of common facts. The sample shown here asserts an entry from a Date Book. Humans can readily interpret the meaning of the entries in this example. To a software agent these are just symbols without any meaning. There is an unspecified relationship between a datebook entry, in the example, and an appointment in some other application. It is up to the developer to define the relationship between the two concepts.

The way RSS evolved is a good example of the lack of semantics in XML. RSS was designed to support news feeds. Presently, it is used to share many types of data, including video 'podcasts' and hurricane forecast graphics (Apple, Inc. 2008),(National Hurricane Center 2008). Apple's popular iTunes service uses a modified version of RSS (Apple, Inc. 2008). There are no built-in semantics to automatically infer relationships between the standard RSS types and the Apple extended types. If a developer chooses to include application support for Apple's extended RSS tags they must be coded into an application. If every vendor made their own version of RSS then an application developer would need to encode for all of the different standards. The lack of a clear framework to express relationships between extended types can void any benefit of using XML and standard schemata.

This does not mean that XML does not have a place in the future of the Web. The concept of wrapping data in XML as opposed to the past practice of using unstructured files represents a major advancement for machine representation of data. Using XML as a representation language allows complex data to be represented in a consistent manner, justifies development of XML schema, and promotes the development of standardized tools.

```
1 <?xml version="1.0"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3          xmlns:exterms="http://www.example.org/terms/">
4 <rdf:Description rdf:about="http://www.example.org/index.html">
5    <exterms:creation-date>August 16, 1999</exterms:creation-date>
6 </rdf:Description>
7 </rdf:RDF>
```

FIG. 3.2. Example of an RDF entry.

Resource Description Framework The Resource Description Framework (RDF) is based on the XML format, and is the basic building block of the Semantic Web. Resources are uniquely identified by a Uniform Resource Identifier(URI) and their relationships to other resources. Resources are described as a set of triples of *subject*, *property* or *predicate*, and a *value* or *object*.

Figure 3.2 shows a basic RDF entry. Line one specifies the version of XML compatibility. Lines 2-3 specify the RDF namespace alias to URI's, such that the prefix `rdf` maps to the URI `http://www.w3.org/...` and the prefix `exterms` maps to the URI `http://www.example.org`. Line four introduces a new resource with a subject of `http://www.example.org`, a predicate of `exterms:creation-date`, and a value of `August 16, 1999`. Note that `exterms` is really an alias for the full URI `http://www.example.org/terms`.

The Resource Description Framework has several advantages over the simpler XML language. Objects are described using simple sets of properties (i.e. the RDF triple). Resources are identified by their URI, allowing linking between resources across the web, and allowing resource descriptions to be re-used (World Wide Web Consortium 2005c). However RDF lacks a common vocabulary for describing key properties needed by inference procedures (e.g. isA). This is addressed in the Resource Description Framework Schema (RDFS) extension. Even with these extensions, RDFS is not expressive enough to capture

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">
  <owl:Ontology
       rdf:about="Describes simple car and driver relations."/>
  ...
```

FIG. 3.3. Example of a OWL-DL ontology header

all class relationships (World Wide Web Consortium 2001).

3.2.2 OWL

The Web Ontology Language (OWL) was recently recommended to the W3C for acceptance as a standard. OWL is based on RDF and DAML+OIL, another Semantic Web Language. There are three flavors of OWL: *Full*, *DL*, and *Lite*, forming a hierarchy, such that an ontology using OWL Lite is also in OWL DL, and an ontology using OWL DL is also in OWL Full (World Wide Web Consortium 2005b).

The three flavors of OWL support different degrees of expressiveness and reasoning properties. OWL Lite provides for a very basic classification hierarchy and simple constraints. OWL DL supports highly expressive ontologies and also admits complete and decidable reasoning. OWL Full supports the maximum expressiveness, but looses the efficient reasoning capabilities of OWL DL, since it is unlikely that there will ever be an efficient reasoner for OWL Full (World Wide Web Consortium 2005b). This research considers only OWL DL. It is the only one of the three versions that contains the necessary expressiveness for this project and supports efficient, complete, and decidable reasoning.

OWL-DL ontologies contain an optional RDF/RDFS style header and a set of *class*, *property*, and *individual* descriptions. A sample header appears in Figure 3.3. The

RDF/RDFS header contains a set of aliases that define the name space (and also link ontologies together). This header imports several different name spaces and associates each with an alias. It also contains an annotation which allows expression of ontological metadata, similar to comments in programming language source code. These annotations are meant for developers and are not used by reasoners.

3.2.3 OWL Classes

A class in OWL is described using formal descriptions that state requirements for an individual to be a member of the class. These requirements are specified as sets of properties that are either necessary and sufficient for membership (allowing class membership to be inferred), or simply necessary (allowing individual properties to be inferred based on class membership assertions). Classes are organized into an inheritance hierarchy (using `rdf:subClassOf`), such that one class that is a subclass of another class inherits the properties of the super class.

OWL-DL allows multiple inheritance, where a class is defined as a sub-class of multiple super-classes. Multiple inheritance allows extremely expressive class concepts to be developed, linking various branches of the class hierarchy tree. It also creates the potential to introduce inconsistencies into the knowledge base. For example, a sub-class may share two super-classes with mutually exclusive role restrictions. Ontology developers must be wary of unintended relationships and the possibility for inconsistencies caused by multiple inheritance.

Figure 3.4 shows a simple class definition of a class named *Vehicle*. Lines 2-9 assert that a *Vehicle* has the *hasDriver* property, which is satisfied if an individual is defined with an individual that is a subclass of the *Driver* class. Line 10 is important, as it asserts that a *Vehicle* is disjoint from other sibling classes, in particular, that a *Vehicle* cannot also be a *Driver*. Finally, line 11 asserts that a *Vehicle* is a sub-class of *owl:Thing*.

```
<owl:Class rdf:about="#Vehicle">
   <rdfs:subClassOf>
     <owl:Restriction>
       <owl:onProperty>
         <owl:ObjectProperty rdf:ID="hasDriver"/>
       </owl:onProperty>
       <owl:someValuesFrom rdf:resource="#Driver"/>
     </owl:Restriction>
   </rdfs:subClassOf>
   <owl:disjointWith rdf:resource="#Driver"/>
   <rdfs:subClassOf
       rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
</owl:Class>
```

FIG. 3.4. Example of an OWL-DL class definition of *Vehicle*

```
<owl:ObjectProperty rdf:ID="isDriverOf">
   <rdfs:domain rdf:resource="#Driver"/>
   <owl:inverseOf>
     <owl:ObjectProperty rdf:about="#hasDriver"/>
   </owl:inverseOf>
 </owl:ObjectProperty>
 <owl:ObjectProperty rdf:about="#hasDriver">
   <rdfs:range rdf:resource="#Driver"/>
   <owl:inverseOf rdf:resource="#isDriverOf"/>
 </owl:ObjectProperty>
```

FIG. 3.5. Example of an OWL-DL property definition of *hasDriver*

Defining a class without creating inconsistencies using OWL can be complex. For example, in the previous example, it is recommended that all sibling classes assert that they are mutually disjoint with each other. Consider the definition of another primitive class, *Boat* that also has the property *hasDriver*. Using this simple criteria, a reasoner would conclude that *Vehicle* and *Boat* are the same class. There are a number of design issues that apply to defining OWL classes, and this discussion is beyond the scope of this work. See (Horridge *et al.* 2004), (Stanford Medical Informatics, Stanford University School of Medicine 2006), (Noy & McGuiness 2005) for a more detailed discussion of these issues.

```
<Driver rdf:ID="Tom">
 <isDriverOf>
   <HondaPilot rdf:ID="TomsHondaPilot">
     <hasDriver rdf:resource="#Tom"/>
   </HondaPilot>
 </isDriverOf>
</Driver>
<owl:AllDifferent>
 <owl:distinctMembers rdf:parseType="Collection">
   <Driver rdf:about="#Tom"/>
 </owl:distinctMembers>
</owl:AllDifferent>
```

FIG. 3.6. Example of an OWL-DL instance definition of a *Driver*

OWL Properties Properties in OWL are binary relationships between a class and other classes or values. They may be defined as invertible, symmetric, and transitive. Figure 3.5 is an example of a definition of a simple property *hasDriver* and its inverse, *isDriverOf*.

Classes can be defined using complex combinations of properties, including:

- *owl:intersectionOf*,
- *owl:unionOf*, and
- *owl:complementOf*.

Asserting these relationships actually implies the existence of an anonymous class that carries the properties described by the property (Horridge *et al.* 2004).

In addition to these basic properties, OWL-DL also supports multiple inheritance of properties. OWL-DL allows the definition of a property as a sub-property of a parent, where the sub-property inherits the domain and range restrictions, as well as the inversion, symmetry, and inversion properties of its parent.

OWL Individuals OWL allows definition of *individuals* by defining their properties and assigning corresponding values. Individuals are consistent only when their values meet the defined constraints. In Figure 3.6, an individual with an ID of *Tom* is defined to be of class *Driver*. Lines 2-6 assert that *Tom* is a driver of another individual, *TomsHondaPilot*. Finally, lines 8-12 define a collection of individuals that are all mutually distinct from one another. This is necessary since OWL-DL does not make the Unique Name Assumption for individuals: objects with different names are not automatically assumed to be distinct (see Section 2.4.2). (World Wide Web Consortium 2005b)

Semantic Web Languages Summary The preceding description of Semantic Web languages represents a brief sample of the many different languages being developed or proposed as standards to the W3C. The Semantic Web is and will be a multilingual space, where developers may select the appropriate language for an application.

3.3 Semantic Web Development

Publishing information to either the traditional web or the Semantic Web seems to follow a similar path. First, a view of the data to be published must be selected. Then, a markup language must be selected. The data must encoded using that markup language. The final result must be placed on a web server and made accessible to a selected group of users.

The similarity between these two tasks vanishes quickly upon closer inspection. The intended audience completely differentiates these two tasks. When the audience is a human in the traditional web, data is frequently summarized and loses much of its internal structure. When the data is published for the Semantic Web, the data should be published in full context, complete with its relationships to other data elements. The markup language that is selected will either help create visually appealing web pages or effectively encode clear

semantic relationships between knowledge. The actual markup tools used will either help the user maintain consistent styles across pages of a website and check for link errors; or it will use the reasoner to help ensure that the ontology is consistent.

This section provides an overview of the current development tasks and tools that are being developed to support the emerging Semantic Web and make interoperability of reasoning tasks seamless. The development tasks, many of which are also active research areas, are: ontology generation, ontology linking and reuse; ontology integration, merging and mapping; and trust and provenance.

3.3.1 Generation

Ontology generation refers to the process by which domain knowledge is encoded into an ontological representation. The goal is to encode knowledge in such as way as to allow automated reasoning procedures to obtain meaning and useful results through symbolic manipulation (Embley 2004). Currently ontology generation is a manual process, requiring individuals with domain expertise and knowledge of description logics in order to develop ontologies that are applicable to the relevant domain and are correct and consistent representations of the domain with respect to the description logic framework being used.

3.3.2 Linking and Reuse

Reuse describes the process where existing ontologies are reused to publish new ontologies. The RDF name space mechanism allows a single ontology to be composed of existing ontologies, creating a global database of linked schemas. Linking to, and thus reusing existing ontologies, may reduce the development costs associated with ontology development, and ideally helps improve the semantic representation of the published information.

One example of a frequently used ontology is the Friend-of-a-Friend (FOAF) schema

(Brickley & Miller 2008). According to Swoogle, this is the most frequently referenced ontology schema in the Semantic Web (Ding *et al.* 2004). Tools such as *FOAF-a-Matic* simply prompt the user for information and generate an RDF document that imports the FOAF schema adds the appropriate values RDF entries based on the user's input (Dodds 2008). Using this tool, users with little or no experience with description logics are able to populate the form and generate an RDF document that describes their personal information.

Reusing ontologies has many challenges similar to reusing source code that can lead to errors (Noy 2005). These errors may include: ontology revisions, incorrect reuse, and finding appropriate ontologies.

3.3.3 Revisions

Revisions to existing ontologies may cause dependent ontologies to become inconsistent. The RDF, through its name space mechanism, allows an ontology to import external ontologies and define members using those external schema. Consider the following example of how revisions and reuse can create inconsistencies:

Example Suppose Alice published an ontology O_A, that contained a property P, defined with a minimum cardinality of one, $|P| \geq 1$. Bob found ontology O_A and imports it into his ontology O_B, along with some other set of ontologies O_{other}, such that $O_B = O_A \oplus O_{other}$, where the binary relation $\oplus$ represents the composition of two ontologies through a union operation. Later, Alice modifies O_A, and alters the cardinality constraints on P, such that $|P| \geq 2$. Any individuals in O_B that do not meet the new constraints will be inconsistent with O_A.

Currently, the best solution is to strongly enforce a *best practices* approach toward semantic revisions (World Wide Web Consortium 2005c). The suggested practice is to include a version number in the URI of the schema (for example, see the previous FOAF

URI). In so doing, a developer can create new versions of existing resources without breaking any linked documents.

3.3.4 Integration, Merge, and Mapping

Integration is commonly used to describe three different tasks including: building ontologies by reusing existing ontologies (we called this linking / reusing - see Section 3.3.2), when building an ontology by merging existing ontologies (not linking), and building an application using one or more different ontologies (Pinto 1999). We refer to integration as the process of defining a new ontology by conglomerating, in whole or in part, a set of different ontologies (possibly from different domains) into one new ontology (Pinto & Martins 2001).

There are five different strategies for solving this problem for developing integrated ontologies (Ding 2005):

One Centralized Global Ontology This strategy would impose a global schema over the Semantic Web. This removes the loosely federated development processes of the current semantic web and forces development of new ontologies to some central authority. While it is tempting to speculate that this would remove semantic ambiguity, it is unlikely that this would be the case. This is clearly unsupportable without considerable investment of resources and is in disagreement with the stated design goals of Semantic Web.

Merging Ontologies Merging derives a new ontology from a set of candidate ontologies. Generally, a candidate is selected based on some heuristic, such as linguistic / natural language processing, syntactic analysis, or some hybrid system. Classic examples include Chimera (Zhu *et al.* 1999), and PROMPT (Noy & Musen 2001).

Mapping Ontologies Mapping ontologies uses a process to map between entries in two ontologies based on a manual, semi-automatic, or fully automatic process. Automated processes often rely on NLP or statistical approaches to identify mappings in the structure or contents of the two ontologies. Example systems include Anchor-PROMPT (Noy & Musen 2001), BayesOWL (Pan 2005), and GLUE (Doan *et al.* 2002).

Ontology Translation Ontology translation takes two ontologies and attempts to translate the structure and individuals from one to the other. OntoMerge is an example system that uses a set of refactoring rules to translate from one ontology to another. (Dou, Mcdermott, & Qi 2002).

Runtime Ontology Resolution Each of the previous strategies happens during ontology creation. Runtime resolution is designed to detect and handle conflicting information that is identified during the reasoning process.

These tasks each depend on the evidence in the ontology and make particular use of finding relationships that have the same or similar semantics. Consider the problem of mapping two ontologies to each other. The mapper must be able to find similar concepts, properties, and individuals and add assertions to map them onto each other. The lack of well-defined assertions, or worse, unequally defined assertions will be problematic.

Suppose two ontologies are to be mapped to each other. The first ontology contains a property, `ownsCar`, which has a domain of `Person` and a range of `Car`. The second ontology contains a property, `hasTitleTo`, which does not have a domain and range at all.

What conditions must be met for the mapping agent to join these two properties? In order for the mapping agent to even consider these two for mapping there must be evidence that they are related. These two properties have different applicability. One clearly applies to people and cars, while the other is not related to any specific concepts (no domain or

range). The mapping agent will external external knowledge that car ownership is identical to having the title to the vehicle.

To continue this thought experiment, consider the situation where the property `hasTitleTo` has a domain and range that the mapper identifies as being identical to the `ownsCar` property's domain and range. Now there is strong internal evidence that these properties are similar.

Well defined property constraints are important for the successful operation of many of these algorithms. Further, the success of these algorithms is important for the successful growth of the Semantic Web. Therefore well defined property constraints are important for the successful growth of the Semantic Web.

3.4 Conclusion

The Semantic Web is an evolving research and development enterprise. Developed for the representation and reasoning about information, the Semantic Web is a major departure from today's visual web. By marking up information for machines, the goal of the Semantic Web is no longer how to render information on a screen, but to make sites contain relevant, complete, and accurate information.

New languages, extensions, reasoners, and tools are continually being developed. Interest in Semantic Web technology is spreading beyond academic interest, and systems are beginning to be deployed with semantic technologies at their core.

However there are significant challenges that must be addressed before wide-spread acceptance of the technology can occur. Some of these areas, including generation, reuse, and mapping, put demands on the quality of the structure of the ontologies that are to be used by those processes.

Ontology generation techniques may be able to bridge the divide between the large

amount of data and semantic knowledge. Generation techniques may also be able to improve existing ontologies to either increase the amount of knowledge that may be inferred from the ontology, or improve the performance of the reasoner to infer those facts.

Without consistent, well-defined, and structured ontologies to reuse or map to, then the quality of the mapping and reuse will suffer. Ontologies that are missing information or are not fully developed to maximize the amount of information that can be inferred from them are going to be less useful to semantic services.

Chapter 4

DOMAIN AND RANGE CONSTRAINT GENERATION

Domain and range constraint generation is the process of automatically creating constraints for the properties of an ontology based on the evidence contained in that ontology. The strongest evidence for domain and range constraints comes from the terminological statements, specifically in the role restrictions used in class definitions. Individual role assertions are problematic for a number of reasons that will be explored in this work.

Constraint generation is an important task for a number of reasons. Proper constraints on a property act as type assertions that add valuable information about the individuals that participate in those roles. Well defined constraints can help define the intention of a particular property, which is especially useful for manual or automated ontology tasks. Despite these benefits, empirical evidence gathered from a large collection of ontologies shows that an overwhelming number of properties are unconstrained. Generating property constraints may be able to fill in the missing constraints or be used as a tool to suggest new or verify existing constraints.

There are many different ways to approach constraint generation. In particular, the approaches described here will rely only on the terminological evidence contained in the definition of class restrictions. Three different approaches: disjunction, least-common named subsumer, and vivification will be explored.

In certain circumstances, constraint generation can lead to the inference of new information which was previously unavailable in the ontology. In other circumstances, the use of the constraint generation will provide direct evidence of concepts which the reasoner would discover through inference. In this case, the direct assertions contained in the constraints will reduce the work required by the reasoner to generate it. The determining factor depends on the relationships between the terms in the inheritance hierarchy. The most extreme cases of generated constraints will be created that are long chains of disjunctions of terms or else the top concept. In the best cases, constraints can be determined that accurately model the domain and range of the property with the least number of useless disjunctions.

Section 4.1 provides a detailed overview of domain and range constraints and explains why they are an important feature of a description logic. Section 4.2 presents an overview of the generation process and available sources of information. Section 4.3 describes the disjunction algorithm, Section 4.4 presents the least-common named subsumer algorithm, and Section 4.5 presents the vivification algorithm. Finally, concluding remarks about constraint generation appear in Section 4.6.

4.1 Domain and Range Constraints

OWL properties are based on roles in Description Logic (Section 2.1.2). Properties are interpreted as a mapping between instances of two classes or a class and a datatype. A property P, written in OWL as $ObjectProperty(P)$ or $DatatypeProperty(P)$ defines a mapping $P \subseteq O \times O$ or $P \subseteq O \times LV$ where O is some class and LV represents a literal value (World Wide Web Consortium 2005b). When defining the terminology (TBox) for an ontology, a property is described in terms of its relationship to other properties (a property may be a sub-property of another property), or used in a role restriction in the defi-

nition of another class. In the assertional section of the ontology (ABox), property relationships are associated with specific instances, for example, $hasName(JOHN, `JOHN')$, or $teaches(SMITH, ABEL)$.

Property descriptions are translated into generalized concept inclusion (GCI) axioms. A GCI defines a subset of the universe and is used to compute set membership. The property definition itself does not directly translate into a GCI, but its domain and range constraints do. Restricting the domain of a property to concept C is translated into a GCI of $\exists R.\top \sqsubseteq C$, and restricting the range of a property to concept D is translated into a GCI of $\top \sqsubseteq \forall R.D$ (Tsarkov & Horrocks 2004).

Asserting that some unconstrained property relates two individuals but does not provide any further direct evidence about either the subject or object of the property. The only information given is that the individual is either a `Thing` or a literal value and it shares a relationship through the property to another individual.

OWL extends both object and datatype properties with constructors to optionally define domain and range constraints. An object property definition which includes a domain and range is written in OWL as:

$$\texttt{ObjectProperty}(P\ domain(d_1) \ldots domain(d_n) \ldots range(r_1) \ldots range(r_n))$$

and is interpreted as a mapping:

$$P \sqsubseteq (O \times O) \sqcap (d_1 \times O) \sqcap \ldots (d_n \times O) \sqcap \ldots \sqcap (O \times r_1) \ldots (O \times r_n)$$

A similar OWL syntax and interpretation exists for a datatype properties as well (Grau & Motik 2008).

When an object property connects two individuals, one individual is the subject and

the other is the object. The subject is interpreted as a subclass of each of the domains defined for the property and the object is interpreted as a subclass of the ranges. If there are multiple domain or range definitions then they are treated as a single intersection of each of the types. In the case of a datatype property, the property connects an individual and a literal value. The subject continues to be a subclass of the domains and the literal values are matched to the specific range of the property.

Treating multiple constraints as an intersection is counter-intuitive. Suppose a property, `hasLegs` is defined with a domain of `Animal` and `Furniture`. Any subject of this property will be asserted to be *both* `Animal` and `Furniture`. Ontology developers frequently create a single constraint that is a disjunction of a number of concepts. This is such a common practice that tools like Protege do this by default.

4.1.1 Unconstrained Properties

Unconstrained properties are of little value when references to them occur only in direct assertions between individuals. The presence of an assertion states that a relationship between the two individuals exists. Since there are no domain or range statements and no other reference to the property, there are no further type assertions that can be inferred. The meaning of the relationship is subject to the external interpretation of what the symbolic property name represents.

Domain and range constraints on properties can provide valuable information to ontology developers and reasoners. Constraints are not strictly necessary according to the OWL specification. Including a constraint definition provides a tighter set of semantics for the property and stronger typing of those individuals that are fillers of the property.

Domain and range constraints provide important information regarding the intentions of the original author of the property to a developer reusing the ontology. One of the fundamental design goals of the semantic web is the reuse of ontologies. Specifying the

domain and range constraints for a property helps communicate exactly how this property is supposed to be used. Based on these constraints an engineer can decide if the given property is applicable to classes in their ontology.

The accompanying class assertions of stated domains and ranges add valuable axioms to the reasoner. In some cases, this helps reduce the work of the reasoner by asserting instance information directly that would otherwise need to be inferred. When the reasoner encounters a property assertion on a pair of individuals, the subject individual immediately becomes a subclass of the domain and the object becomes a subclass of range of the property.

There are many reasons why these constraints may be left out. Some of these reasons are:

- Information is unknown
- Faulty model of the world
- Artifact of ontology generator
- User error
- Intention of developer
- To avoid conflicts with reuse of the ontology

One reason the constraints may not be specified is that the information may simply not be known to the ontology developer. The developer may only have partial information about a domain and is therefore unable to fill in the domain and range with necessary precision. This may happen if the developer is unfamiliar with the domain he is modeling and unable to accurately describe appropriate constraints, or the constraints may not even exist in the ontology.

One signal that there are faulty semantics in an ontology is that the ontology lacks concepts that adequately describe key concepts in the ontology. For example, the ontology lacks concepts that describe the domain and range for a property. For example, suppose the ontology consists of a property `hasLegs`. The developer uses this property to describe two disjoint concepts: the number of legs of a `Dog` and a `Table`. The domain of the property `hasLegs` becomes `Dog` $\sqcap$ `Table` because OWL combines multiple statements as an intersection of those classes. Because these are disjoint classes, the intersection is unsatisfiable, and the ontology is inconsistent. Although this is a rather overly simplified and egregious example of the problem, it is clear that the inability to select a concept for the domain hints at a larger problem with the model of the world. In this example `hasLegs` should probably be split up into different properties.

Ontology generators take data from some non-semantic source such as a relational database and restructure the data into an OWL ontology (Velardi, Fabriani, & Missikoff 2001), (Modica 2002). These systems typically do well at organizing the data into classes, individuals, and properties based on the relations of the original schema. They frequently have difficulty accurately constructing the domain and range concept descriptions from these sources (Modica, Gal, & Jamil 2001). In Section 2.1.2 the differences in interpretation of domain and range constraints between Description Logics and RDBMS was introduced. Using the domain and range *type restriction* in an RDBMS may not be equivalent to the *type definition* semantics of Description Logics.

The lack of a domain and range constraint could simply be due to user error. Using tools like Protege, one must first create the classes for the domain and range and then create the properties. In a complicated, large ontology, it is very easy for the developer to add the properties and not return later to fill in the constraints.

The final reason for the lack of constraints is that it could be the intention of the developer. The developer may be using a property as a very broad property assertion and

does not intend it to be used to automatically create class inclusion axioms with a particular property assignment. There are good reasons why a developer may intentionally choose to leave a property with under-specified constraints. Two such reasons are that there is not sufficient expressivity in the chosen language to support the intended constraints and that the ontology is expected to be used by other developers and there is no way to extend constraints in an imported ontology.

The present version of OWL does not support domain to range mapping. In the definition provided above, the domain and range can be specified as a collection interpreted as the intersection of the members. A specific domain and range cannot be specified as a pair. Consider the `hasLegs` example above. If the domain and range could be paired, then the property could contain a pairing of `Biped` has range 2, while domain of `Quadruped` has range 4.

OWL is designed to be a language for the Semantic Web. A critical part of this language is the support for linking to other documents in the Semantic Web. With the present version of OWL it is easy to import another ontology and integrate and extend its classes, properties, and individuals. One thing that cannot presently be done is to modify the domain and range constraints of a property. There is not a mechanism to extend the definition of an imported property's constraints to include classes in the importing ontology. To do so may result in a non-monotonic operation where facts that were dependent on the domain and range constraints will need to be retracted as a result of adding additional definition to the constraints.

There are many reasons constraints may be left off a particular property. Several of these reasons, as previously shown, were unintentional omissions, while others are intentional and key to the representation of the ontology. The problem, with respect to this work, is that there is no way to tell the difference.

4.2 Domain and Range Generation

What remains to be explored is whether anything can be stated about the domain and range constraints that may apply to a given property. The way in which a property is used provides some hint of what constraints may apply to that property. There are two sources for this information: the class descriptions, specifically the role restrictions in the TBox and the instance assertions in the ABox.

4.2.1 Generating Constraints from Individual Assertions

Instance assertions are property and type definitions applied to individuals in the ABox. They are problematic for determining what constraints may apply. The domain and range constraints must be extracted from the set of class membership assertions for each of the individuals in the knowledge base that are related by a property. This creates a two-fold problem: the property assertions on individuals may be missing, and the set of concepts may be overlapping.

First, there must be individuals in the ABox that are related by a property. If there are individuals, there may be a number of different type assertions for the individual, some directly asserted, while others are inferred. Because of the Open World Assumption, we cannot conclude that the set of type assertions is complete. Because of the Unique Name Assumption, there may be two or more individuals that represent the same real instance and the set of type assertions may be distributed across each individual. With a complete set of type assertions and property memberships for a single individual, there needs to be a method to determine which types of the individual are related to any one particular property assertion. This is an open and on-going area of research.

The example shown in Figure 4.1 demonstrates the difficulty with using instance assertions to generate property constraints. Using the assertional data, what can be stated

```
1  Class: Person
2  Class: Vegetarian, subClassOf: Person, ...
3  Class: Driver, subClassOf: Person, drives some Car, ...
4
5  ObjectProperty: drives
6
7  Individual: Fred
8          Types: Driver, Vegetarian
9
10 Individual: George
11         Facts: drives(SportsCar)
12
13 Individual: SportsCar
```

FIG. 4.1. Individuals may have many classes. Using instance assertions is problematic.

about the domain and range constraints for `drives`? There are two individuals, `Fred` and `George`. The individual `Fred` is a member of a number of different classes, but does not have an explicit property assertion involving `drives`, so `Fred` contributes no information for the constraints. `George` does have an explicit property assertion on `drives`, but neither `George` nor `SportsCar` have any class memberships; the reasoner cannot infer any memberships aside from `Thing`.

4.2.2 Generating Constraints from Terminological Descriptions

Information stored in the terminological assertions (the TBox) is more useful for circumscribing the domain and range for a property. Figure 4.2 shows a simple ontology defining three classes: `Man`, `Woman`, and `FighterPilot`; and two object properties: `hasGender` and `drives`.

Lemma 4.2.1. *Let P be a property in some ontology, and $C_1, C_2, \ldots C_n$ be defined classes in the ontology which are subclasses of a role-restriction, $C_i \subseteq P.D_i$ involving property P, where D_i is the object of role restriction. The domain of P must subsume the set $C' = C_1 \sqcup C_2 \sqcup \ldots \sqcup C_n$. The range of P must subsume the set of objects, $D' = D_1 \sqcup D_2 \sqcup \ldots \sqcup D_n$.*

```
1 Class: Man EquivalentTo Person and hasGender male
2 Class: Woman EquivalentTo Person and not (hasGender male)
3 Class: FighterPilot EquivalentTo Officer and hasGender male and drives some Airplane
4
5 ObjectProperty: hasGender
6 ObjectPropety:  drives
```

FIG. 4.2. Terminological Assertions for Constraint Generation

Proof. Let δ and ρ be the domain and range of some property P, respectively. Let $C' = C_1 \sqcup C_2 \sqcup \ldots \sqcup C_n$ represent the set of classes which are subsumed by role-restrictions involving property P. Either the C' is the domain for P and $\delta \subseteq C'$, or it is not. Assume $\delta \nsubseteq C'$. This implies that for some concept $C_i \in C'$, $C_i \nsubseteq P.D_i$, that is that a concept is not a subset of its own definition. This implies that $C_i \neq C_i$, which is a contradiction. Therefore, the domain of a property P must subsume the union of the classes which are subclasses of a role-restriction involving P. A similar proof using ρ and D' will show the same results for the range of a property. □

Lemma 4.2.1 states that the domain for a property must subsume the set of all classes defined in terms of a role restriction on that property. For the example shown in Figure 4.2 this implies that the domain for `hasGender` is a class that must subsume `Man`, `Woman`, and `FigherPilot`. Lemma 4.2.1 and Open World Semantics do not preclude other classes that are not yet represented from being subsumed by the domain. By momentarily closing the world, this lemma allows us to construct a constraint from the present state of the terminology. The Lemma does not require the generated constraint to be minimal and does not enforce any future restriction on the relationship of the classes involved in the constraint description.

Lemma 4.2.1 provides a simple method of constructing a domain and range constraint for each property in a given ontology. These are not the only constraints possible.

Lemma 4.2.2 shows that there will be either exactly one trivial constraint, or there will be, in the worst case, an exponential number of constraints.

Lemma 4.2.2. *When constructing a constraint from the available evidence in an ontology, for any property in that ontology, there is either a single, trivial constraint definition, or there are an exponential number of possible constraints based on the combinations of classes in the ontology and connectives used in the species of OWL.*

Proof. For the first case, for a single, trivial constraint definition, such as `Thing` or `Nothing`. The property's domain and range either subsumes everything or the constraint is inconsistent. For the second case, a constraint that cannot be applied to the entire ontology, the number possible domains is determined by the number of combinations of classes that the constraint can be applied to. If there are n classes (including all direct and indirect super-classes) which are defined in terms of the property, with the standard set of connectives: $\langle \cup, \cap, and \neg \rangle$, then there are 3^n ways to combine the terms to describe the constraint. Neither will all of these combinations be valid, nor will they be unique. □

There are three classes that are defined in terms of role-restrictions on `hasGender` of the ontology shown in Figure 4.2. Lemma 4.2.1 shows that one domain could be `Man` ⊔ `Woman` ⊔ `FigherPilot`. This is not the only possible domain. Lemma 4.2.2 shows that there may be other domain descriptions. For example, another domain could be `Person` ⊔ `FighterPilot`, which generalizes the descriptions for `Man`⊔`Woman`. The domain could also be `Person` ⊓ `FighterPilot`, or even `Person` ⊓ ¬`FigherPilot`. The point that is being made by Lemma 4.2.2 is that exhaustively checking all of these possible combinations is intractable for large ontologies.

When generating domain or range constraints for a property, there is either one, many, or an unknown number of possible constraints. Since the goal is of improve the semantics of the model, the trivial case of `Thing` can be removed from further consideration. In each

of the subsequent ontology generation methods, the Open World Assumption is suspended during the generation process. The implication of doing this is to give up the monotonicity property of inference and shift the reasoning process to a form of default reasoning. This will be explored in more detail in Chapter 5.

Three different constraint generation approaches will be discussed in the following sections, including:

- Disjunction,
- Least Common Named Subsumer, and
- Vivified Subsumer.

4.3 Disjunction Approach

One method of generating constraints relies on the creation a the disjunction of all dependent classes. This generation method is fast and efficient. It tends to have very weak reasoning results and may even create serious performance problems for reasoners.

An algorithm for generating constraints by disjunction is shown in Figure 4.3. First, the algorithm enumerates all of the properties of an ontology. For each property, the algorithm generates a list of role restrictions. For each role restriction, the subject and object of a role restriction is added to the domain and range lists respectively. After each role restriction is processed, the Least Common Subsumer (LCS) is computed. Because OWL supports the disjunction of concepts, the LCS of a concept description is the disjunction of its subsumers (Baader & Nutt 2003). Thus, a disjunction list of concepts is added to the property's definition as a domain or range constraint.

This algorithm runs in time that is polynomial to the number of properties and restrictions. Let p be the number of properties, and r be the number of restrictions, then the

growth-class can be characterized as $O(pr)$ to maintain the content of the lists.

```
GENERATE-DISJUNCTION(O)
 1  P ← ENUMERATE-PROPERTIES(O)
 2  for p ∈ P
 3       do
 4          R ← ENUMERATE-RESTRICTIONS(O, p)
 5          δ ← {}
 6          ρ ← {}
 7          for r ∈ R
 8               do
 9                  if subject[r] ⊈ δ
                           ▷ Add subject of r to domain
10                    then
11                         δ = δ ∪ subject[r]
12                  if object[r] ⊈ ρ
                           ▷ Add subject of r to range
13                    then
14                         ρ = ρ ∪ object[r]
15          domain[p] ← owl : UnionOf(δ)
16          range[p] ← owl : UnionOf(ρ)
17
```

FIG. 4.3. Disjunction Generator

4.3.1 Disjunction Examples

Using the disjunction approach may generate useful constraints. The problem is that disjunctive statements are frequently of little use for the reasoner. Knowing that some class subsumes the disjunction of a set of classes does not allow the reasoner to draw many more conclusions. There are times where the disjunction can provide the necessary information to draw new conclusions, such as that shown in Figure 4.4.

```
1  Class: A SubClassOf: Thing, P some C
2  Class: B SubClassOf: Thing, P some C
3  Class: C SubClassOf: Thing
4
5
6  ObjectProperty: P
7          Domain: Thing        Range: Thing
8
9  Individual: J
10 Individual: I
11         Facts: P(I,J)
```

FIG. 4.4. Example Ontology. Example where disjunction of constraints may be useful.

This example shows an ontology with three classes in a simple hierarchy. There is a single object property `P`, which has no domain or range constraint. Classes `A` and `B` are both defined as subclasses of `Thing`, and are defined with a role restriction that there is some individual who is the object of `P`. Two individual instances are also defined, `I` is the subject of property `P` with `J` as an object.

A reasoner will (correctly) fail to find any additional facts about this simple ontology. The situation changes after the application of the disjunction constraint generator. For property `P`, the algorithm will construct a domain of $\texttt{A} \cup \texttt{B}$, and a range of `C`. A reasoner will now classify `J` as an instance of class `C`. The reasoner still does not classify `I` as a member of any named class, it belongs to a sub-class of the disjunction $\texttt{A} \cup \texttt{B}$. If there were an assertion that `I` belonged to a class that was disjoint from either `A` or `B`, then the reasoner would be able to close the disjunction.

4.3.2 Disjunction Discussion

The symbols used in this example were chosen to be vague. Adding the domain and range constraints clearly changed the model of the ontology. The generator cannot know whether this change is a meaningful modification to the ontology or not. Can humans do

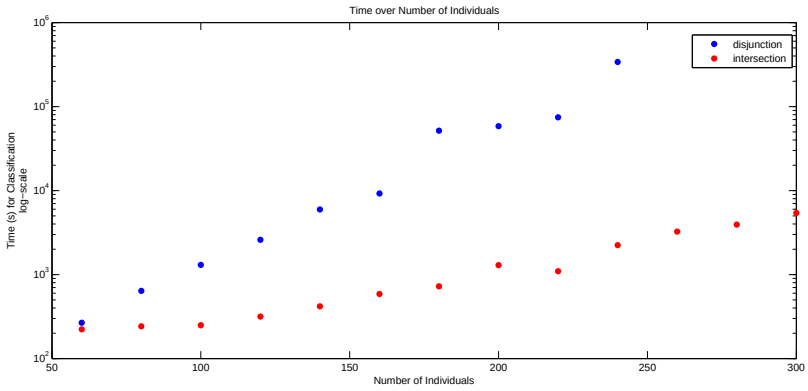

FIG. 4.5. Reasoner comparison of disjunction and intersection.

better by looking at the symbols?

Suppose `P` represents the concept "*teaches*," `C` represents "*student*," and `A` and `B` represent "*professor*" and "*adjunct*." The modification of the domain and range would be meaningful in this case. With the domain and range constraints, `I` is inferred to be some form of a teacher and `J` is inferred to be a student. This is not the only interpretation of this isomorphism of symbols.

Another problem with this method of ontology generation results from the way tableau reasoners handle disjunction. Tableau reasoners were described in Section 2.5. Disjunction introduces a type of non-determinism that is handled using the $\rightarrow \cup\mathbf{rule}$, which creates new potential models from each term in the disjunction. The non-determinism is resolved through search and backtracking through each of these models. The resources required for the reasoner increases exponentially with the size and number of disjunctions encountered by the reasoner. This is demonstrated in Figure 4.5.

The statistics in this figure were generated using a stochastic simulation with parameters chosen to generate reasonable sizes for the model. For each data point, a number of classes N_C, object properties $N_P = \frac{N_C}{2}$, and individuals $N_I = 2N_C$ where created. For each property, the domain constraint was set to a disjunction or intersection of

a random number of randomly selected classes. The expected size of the domain lists, $E(|Dom(Pi)|) = \frac{1}{3}\sqrt{N_C}$. Individuals were assigned a random number of property assertions with probability $P_{PI} = .1N_P$. The results were computed for $50 \leq N_C \leq 300$. The y-axis is a logarithmic scale. The time required for reasoning over the models for both types increased exponentially (linear on a log-axis graph), but the time required by the disjunction was several orders of magnitude greater than for the intersection.

This illustrates a problem with this generation strategy. As the number of role-restrictions involving a property increases, the number of terms in the disjunction will increase linearly. As the number of properties with large disjunct constraints grows, the time and space resources for the reasoner will grow exponentially.

In conclusion, the disjunction method of generation is the fastest and simplest generation method described here. It generates disjunction constraints from each set of role restrictions on a property. These disjunction clauses may add little additional information. The current generation of tableau reasoners have difficulty with long chains of disjunctions and their performance degrades exponentially.

4.4 Least Common Named Subsumer

The Least Common Named Subsumer (LCNS) approach to generating constraints for a property operates by finding the named concept that is the least common subsumer for the set of role-restrictions. The key difference between this approach and the disjunction generator of Section 4.3 is that the disjunction generator allowed an unnamed common subsumer while this approach selects a named subsumer. The requirement to select a named subsumer is an attempt to improve on the results of the previous disjunction algorithm by removing the disjunctions and the non-determinism they create in the reasoner.

There is a trade-off from the “exact” LCS of the previous algorithm to the “named”

concept used by this approach. The LCS created in the disjunction algorithm is an exact representation of the least-common subsumer of the set of concepts. The named subsumer represents an approximation of the subsuming set of concepts. It should be noted that while the disjunction algorithm created an exact representation of the least-common subsumer it did not necessarily create the most exact definition of the constraint. For example, suppose there is a class, A with n direct subclasses. Suppose the LCS for a concept was $A_2 \sqcup A_2 \ldots A_n$, where for each $A_i \sqsubseteq A$. The class A_1 was missing. A more concise, and ignoring open world semantics, description would be $A \sqcap A_1$.

An algorithm which uses the LCNS to generate domain and range constraints is shown in Figure 4.4. The algorithm first determines whether the existing constraint subsumes the current role's subject or object. If it does not, the LCNS algorithm is used to find the least common named subsumer for this pair. The domain and range are updated after all roles are processed.

The reasoner is the determining factor in the performance of the GENERATE-LCNS algorithm. The algorithm calls on the reasoner to compute the full taxonomy of the ontology O. The algorithm makes repeated calls to the reasoner to subsumption relationships between concepts. Let N_c represent the number of classes in ontology O, N_{r_p} represent the number of role-restrictions with property p, and N_p represent the number of properties in O. Let $N_{\sqsubseteq}$ represent the number of calls to the reasoner's subsumption checking procedure. For each N_c classes, $C \in O$ there are three subsumption checks made on line 4 of the LCNS algorithm to compute whether C subsumes the concept descriptions. Equation 4.1 shows the total number of subsumption checks made by the algorithm.

$$N_{\sqsubseteq} = \sum_{i=1}^{N_p} \sum_{j=1}^{N_{r_p}} 3N_c \tag{4.1}$$

```
GENERATE-LCNS(O)
 1  P ← ENUMERATE-PROPERTIES(O)
 2  for p ∈ P
 3       do
 4          R ← ENUMERATE-RESTRICTIONS(O, p)
 5          δ ← ⊥
 6          ρ ← ⊥
 7          for r ∈ R
 8               do
 9                  if subject[r] ⊈ δ
                           ▷ Get current LCNS
10                    then
11                         δ ← LCNS(O, δ, subject[r])
12                  if object[r] ⊈ ρ
                           ▷ Get current LCNS
13                    then
14                         ρ ← LCNS(O, ρ, object[r])
15          domain[p] ← δ
16          range[p] ← ρ
17
```

FIG. 4.6. Least Common Named Subsumer Generator

There will be a polynomial number of calls made to the reasoner's subsumption checking algorithm, and there will be one call made to the reasoner's classification algorithm. Let $R_{\sqsubseteq}$ and $R_{classify}$ represent the complexity for the reasoner's subsumption checking and classification procedures respectively. The total run-time of the algorithm will be in $O(R_{classify} + N_{\sqsubseteq}R_{\sqsubseteq})$. As discussed in Section 2.5, the time required by the reasoner will depend on the expressivity of the description logic being used.

LCNS Examples Figure 4.4 described a simple ontology. The disjunction algorithm created a domain for property `P` of `A⊔B` and the reasoner was not able to infer any additional

```
LCNS(O, A, B)
1  L ← ⊤
2  for C ∈ classes(O)
3       do
4          if A ⊑ C  &  B ⊑ C  &  C ⊑ L
5            then
                  ▷ C subsumes both A and B and is more specific than current LCNS
6                 L ← C
7
8  return L
```

FIG. 4.7. Compute Least Common Named Subsumer

type assertions for individual J. The LCNS algorithm will find the least-common named subsumer of A ⊔ B. In this simple ontology, there is only one named concept that subsumes both A and B namely: Thing. Surprisingly, for this simple example there are no differences in the types inferred for individuals I and J.

Figure 4.8 shows a more complex class hierarchy. The class A is a super-class to classes $\langle$B1, B2, . . . , Bn$\rangle$. Using the disjunction algorithm of Section 4.3 would create an n element disjunction of B1∪B2∪. . .∪Bn. If the domain of P were to be set to this disjunction the reasoner would provide little additional information about individual I except that it was one of those classes. As the size of these disjunctions increases the time and memory costs of the reasoner will increase. In spite of this long disjunction the reasoner is not able to infer any additional type assertions for individual I.

The situation is different with the LCNS-based generator. The range of P will continue to be C, but the domain of P will become A. Using this new fact the reasoner will conclude that individual I is an instance of class A. Because the disjunction is replaced with a simple type assertion the performance of the reasoner will be improved over the previous example.

```
Class: A  SubClassOf: Thing
Class: B1 SubClassOf: A, P some C
Class: B2 SubClassOf: A, P some C
Class: B3 SubClassOf: A, P some C
    ...
Class: Bn SubClassOf: A, P some C
Class: C  SubClassOf: Thing
Class: D  EquivalentTo: P some C

ObjectProperty: P
        Domain: Thing        Range: Thing

Individual: J
Individual: I
        Facts: P(I,J)
```

FIG. 4.8. Example Ontology. Example where LCNS constraints may be useful.

4.4.1 LCNS Discussion

The LCNS-based generator defines the domain and range constraints to be the least-common named subsumer. The LCNS is the most specific named class that subsumes a set of classes. The LCNS of a concept description can be computed by an algorithm that makes a polynomial bounded number of calls to a reasoner. The overall complexity depends on the reasoning costs and expressivity of the Description Logic. The LCNS approach represents a trade-off in specificity and run-time performance from the disjunction-based approach. For many practical applications the LCNS-based approach will not provide much improvement over the current default domain approach.

Figure 4.4 demonstrated the trade-off in specificity. Because the domain of `P` was $\texttt{A} \cup \texttt{B}$, the least-common *named* subsumer of the domain of `P` was `Thing`. In the ontology of Figure 4.8, the LCNS-based approach assigned the domain of `P` to be simply class `A`. The domain of `P` will be a class that is not defined in terms of a role-restriction involving property `P`. Because `A` is more general than the sub-classes that are defined in terms of a role-restriction on `P` there is a possibility that some sub-class of `A` does not include a

restriction on P. The LCNS algorithm selected a class definition that was more general than the individual defining properties. In effect, it traded specificity for performance. The specificity as compared to the more accurate statement of the LCS of the defining properties. The performance is measured in the time and resources required be the reasoner to compute additional inferred facts about the individuals who fill property P.

The LCNS algorithm has a problem in how it generalizes across top-level branches of the inheritance tree. When the least-common subsumer must include two branches of the inheritance tree whose only common ancestor is Thing, then the only common named subsumer is Thing. The disjunction approach effectively handles this situation by including both sets of classes in the disjunction. If the length of this disjunction is small then the disjunction algorithm may be more informative than the LCNS in this situation.

In certain instances the LCNS approach to generating domain and range constraints represent an improvement over the disjunction approach. The LCNS is able to summarize concept descriptions by finding the named concept that subsumes some set of classes. The result is a concept description made of exactly one concept name. There is a trade-off in specificity and performance. The major drawback is that the LCNS may over-generalize and discard too much information. There are times when the LCNS is preferable, and others when the disjunction approach is preferable. These times may occur in the same ontology.

4.5 Vivified Subsumer

The vivification approach is proposed as an alternative to the disjunction and LCNS based approaches. The approach is designed to avoid the long strings of disjunctive clauses while avoiding the over-generalization of the LCNS approach. The concept of vivification describes removing disjunctions with the least common subsumer of the disjunction (Cohen

& Hirsh 1992). For example, $\texttt{Pianist} \cup \texttt{Organist}$ could be vivified into the least common subsumer of $\texttt{Keyboard} - \texttt{Player}$.

The vivification approach proposed used here takes a disjunction and, using the class structure of the TBox, summarizes the term into a definition that replaces subsets of disjunctive terms with a common direct super-class. The goal is to arrive at a description of a domain or range that includes most of the original detail while avoiding the long and often meaningless disjunctive strings.

The original algorithm, proposed by Cohen, requires complete subsumption of the children by the parent to trigger an absorption of children. The algorithm presented here allows for partial subsumption of the children. The degree to which this algorithm will accept partial subsumption is controlled by the parameter at runtime.

The vivification approach initially generates disjunctive strings for a constraint and then replaces subsets of concepts with subsuming super-concepts until the constraint is vivified. The process of replacing a subset of concepts will be called *absorption* which is defined in the following pair of definitions.

Absorption A subset of named concepts in a disjunction may be replaced if they share a common and direct named super-class; and they must meet the *absorption criteria*. If any concept which is part of the subset to be absorbed has more than one direct super-class it is not removed from the disjunction unless each of its super-classes is subsumed by a member of the disjunction list.

Absorption Criterion Let L be a disjunctive list of concepts to be summarized, and $A \subseteq L$ be a subset of L to be absorbed by concept description D. Let $m = |A|$ and n be the number of direct subclasses of D. If $m \geq \beta n$ then the absorption is accepted, or it is rejected otherwise. The parameter β represents a hyper-parameter, with values $0 < \beta \leq 1$, to control the degree of concept inclusion necessary to accept an absorption.

The absorption criterion represents a heuristic to govern how aggressive the vivification process will be in summarizing concept descriptions. The criterion is based on the proportion of sub-classes that are actually included in the list to be summarized. The intuition of the heuristic is based on the fact that information may be incompletely specified in a large ontology. If a sufficiently large proportion of sub-classes is present in a list to be summarized then it is likely that a description which includes only the super-class will be sufficient for reasoning.

```
ABSORB(L, A, C, β)
    ▷ L is disjunction list, A ⊆ L, classes to be summarized, 0 ≥ β ≥ 1 is criteria
1   m ← size[A]
    ▷ All elements in A share same super-class
2   B ← FIND-DIRECT-SUBCLASSES(C)
3   n ← size[B]
4   if m ≥ βn
5      then
6           L' = L − A ∪ C
7      else
8           L' = L

9   return L'
```

FIG. 4.9. Absorption Pseudocode

4.5.1 Vivification Algorithm

Figure 4.9 shows the pseudo-code for the absorption process. Let L represent the list of properties to be vivified. Let $A \subseteq L$ be a list of terms that share a direct super-class C, and are to be conditionally absorbed. Let β represent the absorption criteria. Given

L, A, C and β algorithm ABSORB will determine if the absorption criteria is met. The algorithm computes D, the set of direct-subclasses of C. The absorption criteria is met if $|C|/|D| \geq \beta$. If the absorption criteria is met, then the absorption is performed and $L \leftarrow L - A \cup C$. If the absorption criteria is not met then L is left unchanged.

Figure 4.10 shows the pseudo-code for a process that will vivify a given list of concepts. Let L represent the list of properties to be vivified and β represent the absorption criteria. The algorithm proceeds by building a mapping of classes that are in L and their direct sub-classes (stored in arrays C and S respectively). This mapping is used to select a member, $A \in L$ which should be considered for absorption. After A is selected, the ABSORPTION algorithm is called (see Figure 4.9) to perform the absorption. If the absorption occurs then the arrays C and S are modified to include the new absorbing class C and L is updated to reflect the new state of the concept description.

The final procedure is the actual vivification-based constraint generator. The code is largely the same as the disjunction-based generator. The candidate domain and range lists are built up from the disjunction of the role restrictions like the previous approaches. The main difference occurs in the final two lines where VIVIFY-CONCEPT is used to perform the summarization.

4.5.2 Vivification Performance

The vivification algorithm presented uses the taxonomy of the knowledge base to determine direct sub-class relationships. It does not make any subsumption checks as the LCNS algorithm did. The results of vivification do depend on whether the taxonomy was classified prior to invocation of the algorithm or not. If the reasoner is not used to classify the taxonomy then any inferred sub-class assertions will not be available to the algorithm. This may result in differences in absorption acceptance between classified and unclassified taxonomies. Despite these differences, the algorithm itself does not require the use of the

```
VIVIFY-CONCEPT(L, β)
 1  i ← 1
 2  for c ∈ L
 3       do
 4          for s ∈ direct-superclasses[c]
 5               do
 6                  C[i] ← c
 7                  S[i] ← s
 8                  i ← i + 1
 9
10
11  done ← false
12  while done = false
       do
13          done ← true
14          A ← SELECT-NEXT-SUBSET(C, S)
15          L′ ← ABSORB(L, A, β)
16          if L′ ≠ L
17            then
18                 DELETESUBSET(A, C, S)
19                 for s ∈ direct-superclasses[S]
                        do
20                    C[i + 1] ← S
21                    S[i + 1] ← s
22                    done ← false
```

FIG. 4.10. Vivify Concept pseudocode

```
GENERATE-VIVIFICATION(O)
 1  P ← ENUMERATE-PROPERTIES(O)
 2  for p ∈ P
 3       do
 4          R ← ENUMERATE-RESTRICTIONS(O, p)
 5          δ ← ⊥
 6          ρ ← ⊥
 7          for r ∈ R
 8               do
 9                  if subject[r] ⊈ δ
                           ▷ Add subject of r to domain
10                    then
11                         δ = δ ∪ subject[r]
12                  if object[r] ⊈ ρ
                           ▷ Add subject of r to range
13                    then
14                         ρ = ρ ∪ object[r]
15          domain[p] ← VIVIFY-CONCEPT(owl : UnionOf(δ))
16          range[p] ← VIVIFY-CONCEPT(owl : UnionOf(ρ))
17
```

FIG. 4.11. Vivification Based Generator

reasoner, either before or after to operate.

The performance of the algorithm depends on the number of classes, properties, and restrictions. The `Generate-Vivification` algorithm creates a list of restrictions on that property, and invokes the `Vivify-Concept` algorithm twice for each property.

Theorem 4.5.1. *The vivification algorithm runs in polynomial time,* $O(N_p \max(N_r, {N_c}^2))$ *where* N_c*,* N_r*,* N_p *represent the number of classes, restrictions and properties in an ontology.*

Proof. Let N_c, N_p, N_r represent the total number of classes, properties and restrictions in an ontology respectively. The runtime of algorithm `Absorb` is in $O(N_c)$ due to the cost of finding the direct-subclasses of C. The algorithm `Vivify-Concept` makes at most N_c calls to `Absorb`. This can be seen from the case where `Select-Next-Subset` always selects one class from the ontology and replaces it with a single class C. If C is already in the list L, it is not added again. Therefore, the most times this will be called is bound above by the number of classes, N_c. Thus, `Vivify-Concept` is in $O({N_c}^2)$. The algorithm `Generate-Vivification` has two parts, the first part builds the list of restrictions and the second calls the vivification algorithm on the domain and range lists. In some degenerate ontology there could be a disproportionate amount of restrictions than properties. Because of this, `Generate-Vivification` is in $O(N_p \max(N_r, {N_c}^2))$ □

4.5.3 Vivification Discussion

Figure 4.12 demonstrates the ability of the vivification algorithm to balance specificity for performance. The disjunction approach generates a domain for `P` of $\texttt{B1} \cup \texttt{B2} \cup \texttt{D1} \cup \texttt{D2}$. This accurately reflects the fact that the four classes each have a restriction on P. The LCNS algorithm generates a domain for `P` of `Thing`. This is due to the disjointness of classes `A` and `C` and the fact that the disjunction cross over this disjoint branch of the inheritance tree.

```
Class: A  SubClassOf: Thing
        DisjointWith C
Class: B1 SubClassOf: A, P some E
Class: B2 SubClassOf: A, P some E
Class: B3 SubClassOf: A

Class: C  SubClassOf: Thing
        DisjointWith A
Class: D1 SubClassOf: C, P some E
Class: D2 SubClassOf: C, P some E
Class: D3 SubClassOf: C

ObjectProperty: P
        Domain: Thing        Range: Thing

Individual: J
Individual: I
        Facts: P(I,J)
```

FIG. 4.12. Example Ontology. Example where vivification of constraints may be useful.

The vivification generates a summary domain of $\mathtt{A} \cup \mathtt{C}$. This domain statement is more concise than the disjunction result and it preserves at least some of the information lost in the over-generalization of the LCNS result.

4.6 Conclusion

This chapter introduced the concept of constraint generation for properties in OWL. It is important to generate property constraints when they are missing to and validate the integrity of existing constraints. As described in Section 4.1, there is more information for constraint generation in the terminological definitions, specifically class restrictions, than in instance assertions of the ontology. This chapter introduced several algorithms to generate property constraints from the terminological definition, namely: disjunction, least-common named subsumer, and vivification.

A simple disjunction of concepts can be created for each of the domain and range constraints from the restrictions used to define the classes in the ontology. Long chains of

disjunctions lead to weak generalization, inability to create useful information, and inefficient reasoning. The Least-Common Named Subsumer attempts to address this weakness by selecting classes defined in the ontology. This approach tends to over-generalize. The over-generalization of the LCNS approach does avoid the cost of reasoning with disjunctions but it also fails to add useful information. The vivification algorithm was developed as a trade-off between the other two approaches.

Domain and range constraints can be generated from the terminological statements in the ontology. The structure and quality of the generated statements depends on the completeness of the restrictions present in the terminology and in the algorithm selected. The assertions inferred from the addition of these constraints depends on the ability to create constraints that are not so specific as to become long disjunctive chains of statements that prevent the reasoner for picking any one of them. At the same time, a competing goal is to ensure the constraints are not so generic as to become `owl:Thing` or other top-level concepts that are likely already present in the assertion list for the individuals which participate in the properties.

Chapter 5

DEFAULT REASONING EXTENSIONS FOR OWL

Description Logics, which were introduced in Chapter 2, are a type of logic that trades expressibility for decidability, completeness, and monotonicity. The memory and time requirements of a reasoner can be controlled through careful selection of the types of concepts that can be expressed in a particular Description Logic. Default rules are a type of concept that is not included in most Description Logics. Allowing default rules to be part of the reasoning process often leads to non-monotonicity. Default facts may not always hold with specific instances. It is the process of reconciling these clashes that frequently lead to undecidability and complications reconciling the knowledge base.

In many cases algorithmic knowledge generation is a type of default reasoning which invokes the problems of non-monotonicity. The algorithms described in Chapter 4 is an example of a knowledge generator. In effect the computed domain and range constraints are equivalent to default statements about the constraints. These default statements are consistent with the current set of facts in the knowledge base but they may not hold given future facts. The order in which the ontology is built may alter the constraints that are created because the default statements depend on the state of the knowledge base when they are invoked. As future facts are added to the knowledge base they may conflict with earlier default constraints. These conflicting statements must be reconciled to return the

knowledge base to a consistent state.

This chapter presents a technique to manage the retraction of facts derived from default assertions. The defaultness of a statement is propagated to any derived facts. The result of this operation is that the modified DL reasoner will be able to keep track of which statements that were added or inferred from default knowledge. Using this knowledge, the reasoner can contract the knowledge base to restore consistency. The result is a reasoner that provides limited default reasoning, maintains decidability, completeness, and efficiency.

The remainder of this chapter presents a strategy for modifying the existing OWL reasoner. Section 5.2 presents modifications to the OWL reasoner to implement the contraction operation. Section 5.3 shows the correctness of this approach. Finally, Section 5.4 describes modifications to the OWl language to support this operation.

5.1 Introduction

The constraint generation operation described in Chapter 4 is not compatible with monotonic reasoning. The process of adding general class inclusion axioms (GCI), statements that describe class membership, from existing facts in the knowledge base is equivalent to a default rule regarding the domain and range of a property. The generator creates constraints based on the information that is present, at that time. However, new information can be asserted about the universe of discourse that contradicts the generated constraints. Other facts may be inferred from the generated constraints and added to the knowledge base, making it difficult to resolve conflicts.

Figure 5.1 shows an ontology with a simple class structure including classes `A`, `B`, and `C`. Each of these classes is a sub-class of `Thing`. There is one property `P`, without domain and range constraints. There are two individuals `I` and `J`, that are related through property

```
Class: A SubClassOf: Thing, P some B
Class: B SubClassOf: Thing
Class: C SubClassOf: Thing

ObjectProperty: P
        Domain: Thing        Range: Thing

Individual: J
Individual: I
        Facts: P(I,J)
```

FIG. 5.1. Example Ontology. Example where default reasoning creates contradiction.

```
Class: A SubClassOf: Thing, P some B
Class: B SubClassOf: Thing
Class: C SubClassOf: Thing

ObjectProperty: P
        Domain: A        Range: B

Individual: J
        Types: B
Individual: I
         Types: A
        Facts: P(I,J)
```

FIG. 5.2. Example Ontology. Example of Figure 5.1 after reasoning.

P. Class A is defined in terms of a property restriction on property P. For property P, any of the generation procedures described in Chapter 4 creates a domain of A, and a range of B. The reasoner would then conclude that individuals I and J were members of classes A an B respectively. The result of classification is shown in Figure 5.2.

Now, suppose new information is added to the knowledge base. The restriction $\mathtt{C} \sqsubseteq \mathtt{P}\mathrm{some}\mathtt{B}$ is added to the first ontology shown in Figure 5.1. No information is lost as a result of this process and the ontology remains consistent. Recall that the second ontology shown in Figure 5.2 was the result of applying a default rule. Adding the same statement to it will result in the reasoner concluding that class C is a subclass of A. This is because the domain for P states that every individual that fills the restriction $\mathtt{P}\mathrm{some}\mathtt{B}$ must be an A. If it is not

```
1  Class: A SubClassOf: Thing, P some B
2  Class: B SubClassOf: Thing
3  Class: C SubClassOf: Thing
4
5  ObjectProperty: P
6          Domain: A        Range: B
7
8  Individual: J
9          Types: B
10 Individual: I
11          Types: A
12         Facts: P(I,J)
```

FIG. 5.3. Example Ontology. Example of Figure 5.2 after new fact is added.

the case that C is a subclass of A then the old default domain and range must be revoked and so must the inferred facts that individuals I and J are instances of A and B respectively.

Recomputing the default domain and range for property P results in the domain being $A \sqcup C$. Invoking the reasoner on the modified knowledge base will not add any new facts about individual I. The previous type assertions had to be removed because the default facts that allowed for their inference was also removed. Now because of the additional facts added to the knowledge base there is a reduction in the known facts about the knowledge base.

In each of these cases, the application of the constraint generation algorithm changed the semantics of the ontology. Because most Description Logics support monotonic reasoning, there is no defined operation to retract facts which depend on default beliefs. The result is that knowledge that was inferred from default beliefs, which were later revised, remains in the knowledge base. If the order of operations were altered, as this example demonstrates, the results of reasoning with default rules could be very different.

The preceding automated process is different from a manual revision to the ontology. Suppose the author of the ontology makes a revision to the ontology that changed the semantics of the world. There is a presumption of rationality in the actions of the ontology's

developers. If the ontology is a description of some world model, then that ontology is expected to be consistent with that world model. If there are changes that are inconsistent with that model then the developer will make the necessary modifications to return it to a consistent state.

The source of a change in an ontology makes a big difference with respect to monotonicity. If a change is initiated by a human and there is a presumption of rationality to that change, then it is fair to assume that the human will apply changes that are consistent with the world being modeled. When there are changes the human can make appropriate decisions to correct any errors that may occur. When a change is initiated as a result of a conflicted default rule then an algorithm must make assumptions about the world without being able to observe the full state of that world. The algorithm cannot possibly know more about the world than what is currently asserted in the knowledge base. As a result the reasoner rules will be created and applied that are consistent with the present state but which may conflict with statements that will be entered in the future.

5.1.1 Managing Non-Monotonicity

The assertions contained in a description logic knowledge base restrict the possible states of are universally true in all possible worlds. The inclusion of default rules and reasoning created assertions describing the present state of the world and which may not be consistent in all future worlds. There are now two different types of assertions with differing values of commitment associated with them. One set are based on external evidence and asserted by the ontology's developers and the other set of assertions are believed true based on the internal evidence in the ontology.

Using default rules to derive new knowledge results in a set of at least two plausible worlds. One where the set of facts derived from the default rules holds and one where it does not. When the modeling language allows open, incremental descriptions of the world

then there exists the potential for future assertions to clash with previous default beliefs. Non-monotonic systems address these potential clashes.

The constraint generation procedures defined in Chapter 4 create constraints that are believed to be correct and accurate. These constraints may immediately clash with existing facts in the knowledge base or may clash with future facts that either have not yet been derived or entered. There is a difference between a clash of generated knowledge and asserted knowledge.

If there is a clash between generated and asserted knowledge then the generated knowledge is immediately assumed to be inconsistent. If the generated fact has not yet been added to the knowledge base, then the operation is considered a failure and the state of the knowledge base is unchanged. If a new asserted fact conflicts with an existing generated statement, then that statement and any statements inferred from it must be removed from the knowledge base. This is the basis of a contraction operation was introduced in Section 2.6.

5.2 Modifications to the Reasoner

Section 2.1 introduced the concept that Description Logics, including OWL, are based on tableau reasoners. Tableau reasoners presently use two basic types of production rules to reason and answer queries: unfolding and transformation rules (see Section 2.5). Unfolding replaces concept references with their definitions. Transformation rules derive new facts from existing statements in the knowledge base. These processes are defined for the traditional monotonic Description Logics and do not support the desired contraction operations described in the previous section.

A contraction will be initiated to resolve some clash in a knowledge base. In order to complete a contraction request, the reasoner must be able to remove any default statements

and any statements derived from those default statements from the knowledge base. If the clash remains after a contraction then it remains solely with asserted statements and is beyond the scope of the default reasoning presented here. After contraction, the default generation procedure may be reapplied to recreate the rules.

Definition Default Descriptor is a descriptor which indicates that a statement, C^d, is either added as a result of a default rule, or is derived from a default statement. A class C, is equivalent to its default assertion C^d, $C \equiv C^d$. The descriptor only notes the origin of the statement and does not alter the semantics of the description. The default descriptor will propagate through inference, such that for any derived class E, which was derived in whole or in part from a default class, will be added to the knowledge base with the default descriptor, E^d.

A default descriptor is attached to any statement that is added to the database as a result of a default process. The default descriptor is used to differentiate statements added by a default rule from those that are asserted through the normal OWL process. This is just the first step to supporting a contraction. In order to implement a proper contraction operator those statements that were inferred from the removed statement must be retracted.

5.2.1 Default Descriptor Propagation

Without modification to the reasoner, the inference procedure will use default statements and asserted statements equally. The result is that inferred facts will be added to the knowledge base. These new facts may be derived from either default or asserted statements. In some cases, there may be long chains of dependencies linking the default facts to their final statements. The unmodified reasoner will not track the origin of those statements that were derived from default statements from those that were not. Contracting the

$\rightarrow_{\sqcap}$-rule If $\mathcal{A}$ contains $(C_1 \sqcap C_2)(x)$, but it does not contain both $C_1(x)$ and $C_2(x)$, then $\mathcal{A}' = \mathcal{A} \cup \{C_1(x), C_2(x)\}$.

FIG. 5.4. A tableau reasoner transformation rule.

default statements without also pruning the dependent statements will defeat the purpose of contraction and leave unsupported facts in the knowledge base.

The first proposed modification to the reasoner is to include awareness of the default descriptor into the inference procedures. As facts are inferred the default descriptor will propagate to inferred facts. Figure 5.4 shows one of the transformation rules commonly used to build a tableau reasoner. These consistency preserving transformations typically have an antecedent and a consequent. In this rule, the antecedent is 'If $\mathcal{A}$ contains $(C_1 \sqcap C_2)(x)$ but does not contain both $C_1(x)$ and $C_2(x)$,' and the consequent is 'Then add $C_1(x)$ and $C_2(x)$ to $\mathcal{A}$.'

Default Propagation Rule During inference the reasoner applies an transformation rule to the tableau. If any part of the antecedent has a default descriptor then the default descriptor will propagate to the consequent when it is added to the tableau.

5.2.2 Existential Verification

The tableau transformation rules define the criteria that must be met before the rule can be applied to augment the knowledge base. Figure 5.4 shows an example of a transformation rule for a tableau reasoner. A complete version for the Description Logic $\mathcal{ALC}$ appears in Figures 2.4 and 2.5. The existential check that determines whether an assertion is present in the knowledge base is a common criteria in every one of the rules. The existential check will also be used when constructing a union list of concepts to prevent duplicate concepts from being entered into the knowledge base.

Without modification to the existential verification step the reasoner will treat default statements and asserted statements equally. For example, suppose the knowledge base contains a concept description $Person^d$ which was derived from a default rule. During inference the reasoner infers $Person$, a non-default concept description. Default statements represent information that has not yet been entered into the knowledge base. In this case, the information that was previously only known by default is inferred by the reasoner. The reasoner should replace the weaker default fact with the stronger non-default fact.

Concept Strength Let C and D be two concept descriptions which describe identical classes, then one of the following concept strength relationships must hold:

- $C \succ D$ if C is not default and D is,
- $C \doteq D$ if both C and D have the same default descriptor,
- $C \prec D$ if C is default but D is not.

Using the definition of concept strength suggests that a weaker concept should be replaced by a stronger concept. Using the example above, the strength relationship between the two classes is: $Person \succ Person^d$. Modifying the existing check of the reasoner to obey the *Contains Rule* will control the activation of the transformation rules. Using the modified rule will prevent the activation of a rule when the existential check in the antecedent of the transformation would replace a stronger concept.

Contains Rule The $contains(X)$ predicate will be modified to return true if and only if one of the following conditions is true:

- the knowledge base does not contain X at all, or
- the knowledge base contains a class, Y, which describes the same class as X, and $X \prec Y$.

If a rule is triggered and a new concept is inferred then the concept must be added to the knowledge base. If the existential check of the transformation rule ignored weaker members of the knowledge base then the new, stronger inferred members must be added to the knowledge base without creating duplicate statements. The goal is to avoid creating situations where the same concept is present as a default and non-default fact. To preserve uniqueness the union operator will be modified using the union rule.

Union Rule The $union$ operator will be modified to replace an existing term in the knowledge base if the term being added describes the same class and is stronger than the term in the knowledge base.

For example, suppose $KB = A(x) \cup B^d(x) \cup (C(x) \cap B(x))$. The reasoner selects the expression $(C(x) \cup B(x))$ for transformation. The reasoner checks if KB $contains(C(x))$, it does not, so $KB = KB \cup C(x)$. Next the reasoner checks if KB $contains(B(x))$. In this case, it does, but $B(x) \succ B^d(x)$, so $B(X)$ will be added to the KB using the modified *union* operator, replacing $B^d(x)$.

The Default Propagation Rule, Contains Rule, and Union Rule are the only necessary modifications to the tableau reasoner to support this limited version of default reasoning. These rules allow the reasoner to infer default facts and propagate the defaultness of those facts through the inference process. They also favor non-default facts that are otherwise equivalent to existing default facts. In doing so the knowledge base will tend toward elimination of default facts.

5.2.3 Contraction Triggering

If the knowledge base enters an inconsistent state during reasoning then the unmodified reasoner simply notes the cause and indicates the inconsistent state to the caller. The

presence of default statements in the knowledge base implies that there are now two possible types of clashes in the knowledge base, clashes that depend on default statements, and those that do not.

Lemma 5.2.1. *If a knowledge base becomes inconsistent and the clash occurs between two non-default statements then the clash is a* true clash *and is not caused by default facts.*

Proof. Statements in the knowledge base are either default or not. The inference procedure propagates default descriptors to all statements that are derived from other default statements. Default statements are replaced with equivalent non-default statements when they are asserted or inferred. Every statement that is inferred from a default statement is also a default statement. If a statement is not a default statement it is either in the set of direct assertions in the knowledge base or it was inferred from other non-default statements. If the inference rules are consistency preserving (see Section 2.5.3) then the clash is a legitimate clash and not due to the presence of default statements. □

If the clash is a *true clash* then there are errors in the non-default facts and reconciliation is beyond the scope of this work. If the clash is due to default statements, then the default reasoner should restore consistency. Lemma 5.2.2 shows that contraction will restore consistency to the knowledge base.

Lemma 5.2.2. *A knowledge base containing a clash that is not a* true clash *can be returned to a consistent state by retracting all default statements.*

Proof. Given a knowledge base that contains a clash that is not a *true clash*, then all default statements can be revoked. The statements to be removed can be identified using default descriptors. Because the clash is not a *true clash* the statements that caused the clash are no longer present in the knowledge base. The knowledge base must be consistent. □

```
1  Class: A SubClassOf: Thing, P some B
2  Class: B SubClassOf: Thing
3  Class: C disjointWith(A)
4
5  ObjectProperty: P
6          Domain: Thing        Range: Thing
7
8  Individual: J
9  Individual:
10         Facts: P(I,J)
```

FIG. 5.5. Example Ontology.

Contraction can be triggered by the reasoner based on the internal evidence of the cause of a clash. A clash is detected by the reasoner when the it generates an inconsistent statement. For example, suppose the Abox includes a statement, $A(x)$ for some individual x. If the reasoner generates a statement, $/A(x)$, then there is a clash in the knowledge base. Lemma 5.2.1 states that the reasoner can stop and report the cause of the inconsistency when the cause are two non-default statements. When the clash is caused by default statements the reasoner can invoke the contraction operation to retract all default facts. Lemma 5.2.2 states that the resulting knowledge base will consist of only default facts and will be consistent.

5.2.4 Rebuilding Default Knowledge

In the event of a clash, the reasoner can invoke a contraction operation in order to restore consistency. Contraction results in removal of all default statements. The next step for the reasoner is to re-apply the generation of default statements. The rebuilding operation will reevaluate the default rules based on the new information, which may add a new set of default statements. The inference process can then restart from the beginning. The resulting reasoning process is either consistent or it is not. If the resulting knowledge base is consistent then the reasoner is complete.

If the resulting knowledge base is not consistent then there are incompatibilities with the default rules and the stated facts in the knowledge base. Resolving these types of errors requires knowledge that is external to the ontology being modeled. An intelligent decision on how to reconcile the rule / knowledge incompatibility is required and likely requires an external view of the world being modeled. Most importantly, the work of the reasoner is finished. There is no point in attempting to repeat contraction or reasoning.

Suppose the constraint generation algorithm did not check for consistency as it generates constraints. Figure 5.5 shows a simple ontology that is initially consistent. Using the vivification procedure on property `P` creates a domain and range of `A` and `B` respectively. Later, the statement that `I is − a C` is added to the knowledge base. Invoking the reasoner on the modified ontology results in a conflict. Individual `I` is a type of `C` which is disjoint from class `A`. The default domain of `P` is `A`. By default, individual `I` is a member of both `A` and `C`. This conflict depends on default facts so the knowledge base is contracted. The domain and range for property `P` are restored to `Thing`, and the knowledge base is returned to a consistent state. Because there were no changes to the TBox, which the constraint generation procedure uses, a reapplication of the domain and range rules restores the original default domain and range statements. The ontology remains in an inconsistent state.

The inconsistency demonstrated in this example is difficult to reconcile. In this case, there is a type assertion on an individual in the ABox that conflicts with the default constraints generated from the class restrictions in the TBox. There is insufficient evidence in the ontology to detect whether the inconsistency is caused by a faulty TBox description, namely that `A` is disjoint from `C`; or whether there is a fault in the type assertions for individual `I`; or whether there is missing information in the TBox that would cause the constraint generator to build a different set of constraints.

5.3 Reasoner Correctness

The modifications described in Section 5.2 will preserve the completeness and soundness of the reasoner. The modified existential predicate and concatenation operator do not alter the fundamental operation of the tableau algorithm. The modifications do not alter the generation of the tableau for type assertions over individuals.

The following proof of completion of the unmodified tableau algorithm is shown in Theorem 5.3.1 and comes from Baader and Nutt (Baader & Nutt 2007). The theorem shows that there is a finite sequence of transformation rules that can be applied to a knowledge base. The reasoner can terminate when there are no additional statements that can be derived and reasoning is complete.

5.3.1 Completeness

Theorem 5.3.1. *Let A be an ABox contained in S_i for some $i \geq 1$.*

- *For every individual $x \neq x_0$ in A, there is a unique sequence: $R_1, R_2, \ldots, R_l$ where $(l \geq 1)$ of role names and a unique sequence $x_1, x_2, \ldots, x_l - 1$ of individual names such that $\{R_(x_0, x_1), R_2(x_1, x_2), \ldots, R(x_{l-1}, x)\} \subseteq A$. In this case, we say that x occurs on level l of A.*

- *If $C(X) \in A$ for an individual name x on level l, then the maximal role depth of C (i.e., the maximal nesting of constructors involving roles) is bounded by the maximal role depth C_0 minus l. Consequently, the level of any individual in A is bounded by the maximal role depth of C_0.*

- *If $C(x) \in A$, then C is a subdescription of C_0. Consequently, the number of different concept assertions on x is bounded by the size of C_0.*

- *The number of different role successors of x in A (i.e., individuals y such that $R(x, y) \in A$ for a role name R) is bounded by the sum of the numbers occurring in at-least restrictions in C_0 plus the number of different existential restrictions in C_0.*

Theorem 5.3.1 can be extended to the modified reasoner by simple extension. First, if the knowledge base contains no default statements, then the modifications to the reasoner are not invoked and the results are unchanged. Since the default descriptor only notes the origin of the statement and carries no other semantic meaning, the results of reasoning will remain the same, which gives the following theorem:

Theorem 5.3.2. *Completeness of Default Reasoner. Let $C^d(x) \in A$ be a default concept description. If $C^d(x)$ is not replaced by any non-default reasoning process, then $C^d(x)$ is treated like any other concept description in Theorem 5.3.1. If $C^d(x)$ is replaced by a stronger, non-default concept, $C(x)$, then all occurrences of $C^d(x)$ are replaced with $C(x)$. No additional assertions are added as a result of the replacement. Since $C^d(x) \equiv C(x)$, there will be no possible, additional transformation rules which result from the substitution.*

Theorem 5.3.3. *Completeness of Default Reasoner With Contraction. A reasoner that supports default inference with contraction and knowledge rengeneration will be complete if the default knowledge generation is a finite process and the default inference procedure is complete.*

Proof. Theorem 5.3.2 shows that the inference procedure will be complete. If the inference procedure terminates in a consistent knowledge base then there is no contraction and the completeness results are unchanged. If the knowledge base is inconsistent then it must be contracted. There are finite number of default statements that can be removed from the knowledge base. Lemma 5.3.1 shows that inference to verify consistency of the non-default knowledge base is also complete. If the knowledge regeneration process is a finite process

then the knowledge base can be repopulated with default facts in a finite time. The final invocation of the default reasoner is complete. Therefore, the whole default inference with contraction and knowledge regeneration is complete. □

5.3.2 Reasoner Soundness

The soundness of these results are derived from the soundness of the tableau transformation rules that are used. Baader and Nutt proved that the soundness of a tableau reasoner is derived from soundness of the transformation rules (Baader & Nutt 2003).

Because of the inclusion of disjunction there is an element of non-determinism in the reasoner. The result of a transformation of an ABox is a finite set of ABoxes. The reasoner deals with these sets and non-determinism by searching through the individual ABoxes. The original ABox is consistent if and only if one of the generated ABoxes is also consistent. Let $\mathcal{S} = A_1, \ldots, A_k$ be such a set. Then the set $\mathcal{S}$ is consistent if and only if there is some i, $1 \leq i \leq k$, such that $\mathcal{A}_i$ is consistent. The application of a transformation rule to an $\mathcal{A}$ in $\mathcal{S}$ generates by one, two, or finitely many ABoxes (Baader & Nutt 2003).

Lemma 5.3.4 (Baader's Soundness Lemma)**.** *Assume that $\mathcal{S}'$ is obtained from the finite set of ABoxes $\mathcal{S}$ by application of a transformation rule. Then $\mathcal{S}$ is consistent if and only if $\mathcal{S}'$ is consistent.*

The default reasoner described here uses the same transformations that are described by Baader. To show that the default reasoner is sound requires establishing whether the modified contains and union operations alter the soundness of the transformation.

Theorem 5.3.5. *Assume the transformation rules defined for a non-default Description Logic are truth-preserving. Assume the ABox $\mathcal{S}'$ is obtained from a finite set of ABoxes $\mathcal{S}$ by application of a transformation rule including the modified contains and union operations. Then $\mathcal{S}$ is consistent if and only if $\mathcal{S}'$.*

Proof. The modified contains operator does not change the truth preserving property of the transformation rule. The antecedent of a transformation rule depends on the the non-existence of a particular item. If the item is not in the ABox, then both the original and modified contains operation will return false. If the item is in the ABox and has equal *strength*, then both the original and modified contains operation will return true. The difference is when the item is present and stronger than the item in the ABox. In this case, the contains operator will return false allowing the transformation rule to be applied. The difference between the two contains operators is corrected by the modified union operator. This operator will ensure that the weaker concept is overwritten with the stronger concept. The truth of the transformation is thus unaffected. The generated ABox, $\mathcal{S}'$ is consistent, therefore the original ABox, $\mathcal{S}$ is also consistent. □

The default reasoner is shown to be complete and sound. The strength of these results is that neither do they depend on a particular set of transformation rules nor do they depend on the default statements that are available to be generated. Thus this method can be extended to future Description Logics that require support for default reasoning where the contraction operation depends only on the presence of an indication and transformation rules that propagate the defaultness through reasoning.

5.4 Modifications to OWL

The primary application for this technique is the semantic web language OWL. OWL does not have a construct to represent the origin of a statement. In order to represent the default descriptor in OWL, the language must be modified to identify the default descriptor. There are several approaches to this problem that are worth exploring, namely: the class inheritance mechanism, the annotation property mechanism, and a defined class construction mechanism.

Class Inheritance Mechanism One possible way to represent default information is to create a new class `OWL : Default`, and declare each default concept description to be a sub-classs of `OWL : Default`. The problem with this approach is that the reasoner will require modifications to perform the strength-based rewrite rules described above. When a non-default assertion is added to the knowledge base the reasoner will not replace the existing default class description. The replacement operation requires a subsumption check by the reasoner, which must be carried out while the reasoner is building the taxonomy of the knowledge base, which will have to be carried out recursively. This will cause a serious degradation in the performance of the reasoner because it must try to connect every new inferred statement to the default class definition. One final reason for not using the inheritance mechanism is that the default nature of an assertion is not part of the world being described. The defaultness of an assertion is really metadata about that assertion. Representing such metadata as a class description is a fundamental alteration to the semantics of the model being developed.

Annotation Properties Another approach is to rely on the existing class and object annotation property mechanism. Either by direct use of the existing syntax or a new parallel syntax. This method is very attractive since it would avoid requiring a change to a standard language. The problem with using the existing annotation property is that it would allow multiple assertions about defaultness to be made since there is not a limitation on the number of content of annotations made about a class in OWL. There would need to be a well-defined but non-standardized agreement detailing how the property would be used. The inference rules would need to be modified to look for the default annotations in the terminology and to create new default annotations when necessary. Finally as an annotation property the reasoner would need to interrogate these properties, which are otherwise out of bounds to the reasoner, to answer questions about defaultness.

5.4.1 Default Constructor

The proposed solution is to create a modification to OWL that parallels the deprecation flag given to a class, object, or individual property. Presently, an OWL statement can be marked as *deprecated* which is a hint to other developers that this description has been replaced with an updated version. A similar flag can be added to the OWL language, such as *default* to indicate that the description is a default statement.

This has the advantage that the default descriptor can be removed easily, resulting in the statement becoming part of the asserted knowledge base. Another advantage is that the overhead is reduced to a single statement in the knowledge base (or a triple) and readily available during parsing. Finally the OWL language could stipulate that there will be only one modifier on any given class which is an improvement over the potential for multiple annotations in the previous method.

The main drawback is that this approach would require a modification to the language. This requires submitting the change for a future revision of OWL. This would also be an upward compatible change since old language parsers would invalidate any ontology that included the constructor. This could be a major problem if backward compatibility is a concern.

5.5 Reasoner Implementation

A tableau reasoner that implements the Description Logic, ALC, was created to demonstrate a reference implementation of the default reasoner described in in this chapter. This reasoner is meant as a proof of concept to demonstrate the basic functionality that must be implemented by the reasoner. It is not meant to be a production-ready reasoner that works with the suite of OWL languages. This approach was selected as opposed to modifying an existing reasoner such as Pellet.

TBox	
A	$\sqsubseteq (\forall R.B) \sqcap B \sqcap (\exists S.C)$
R	role
S	role
S	has-domain D (*)
ABox	
$I1$	type-of A
$I1$	type-of B (*)

FIG. 5.6. KB Before Reasoning

Reasoners such as Pellet are complex software packages involving many thousands of lines of code and many compilation units. These tools are meant to support a variety of purposes and include support for more than just the basic reasoning services, such as SWRL rules. Modifying this reasoner to include support for this research would require a significant amount of effort that is only tangential to this work.

The reasoner was implemented to take constructs similar to the OWL class constructors, with class definitions, object property definitions, and instance assertions. The constructs take an optional *default indicator*. The default indicator is tracked as meta-data with each construct. Each of the consistency preserving transformations is implemented as described in Section 5.2. Reasoning preserves and propagates the default indicator throughout the reasoning process.

5.5.1 Reasoning Results

The example shown in Figure 5.6 shows a simple ontology. There is a single defined class, A, and two properties, R and S. A single individual, $I1$ is asserted to have two types: A and B. The notation '(*)' shows that a particular statement is asserted default or derived from another default statement. Thus, the assertion that $I1$ is a type of B is a default statement.

The ontology shown in Figure 5.7 shows the ontology after reasoning. The first action taken by the reasoner is to apply the unfolding rule, which adds the definition of A to the list of facts known about $I1$. Since this is a *non-default* statement, the result of the unfolding is also *non-default*. The $\sqcap-\mathrm{rule}$ is applied next, resulting in each of the clauses of the intersection being added to the list of facts known about $I1$. Normally, the reasoner would not add or replace the assertion that $I1$ is a type of B, because it is already present in the knowledge base. The default reasoner will add the non-default fact, $I1$ type of B, and replace the existing default fact. This occurs because of the modifications to the reasoner's rules with respect to the *contains* predicate.

The next interesting action taken by the reasoner happens when domain operation is applied to $I1$. When the $\exists-\mathrm{rule}$ is applied, the reasoner generates an anonymous individual, $genid1$ to satisfy the rule; and at the same time applies any domain and range assertions associated with the rule. Here, the property, S has a domain of D, which is specified as a default domain. The reasoner adds four statements for this one rule:

1. $genid1$ is created as an anonymous individual to satisfy the $\exists-\mathrm{rule}$.
2. $I1$ is related to $genid1$ through property S to satisfy the $\exists-\mathrm{rule}$,
3. $genid1$ is a type of C due to the $\exists-\mathrm{rule}$
4. $I1$ is a type of D, due to the domain of property S.

This last statement is interesting because the domain was asserted to be a default statement. Because of this, the type assertion that $I1$ s a type of D becomes a default statement. This example shows that the reasoner is able to properly derive facts while maintaining the default descriptor to the inferred facts. This example also shows that the reasoner is able to properly replace any default-fact when an appropriate non-default fact is inferred from the KB.

TBox	
A	$\sqsubseteq (\forall R.B) \sqcap B \sqcap (\exists S.C)$
R	role
S	role
S	has-domain D (*)
ABox	
I1	type-of A
I1	type-of $(\forall R.B) \sqcap B \sqcap (\exists S.C)$
I1	type-of $\forall R.B$
I1	type-of B
I1	type-of $\exists S.C$
genid1	type-of C
I1	S genid1
I1	type-of D (*)

FIG. 5.7. KB After Reasoning

5.6 Contraction

The semantics of the ontology in the previous example are identical to one that does not contain default statements. However, this may not be the case as future information is applied to the knowledge base. In the event that some future assertion causes the reasoner to detect a clash with a default statement, or one derived from a default statement, the reasoner must take action to restore consistency to the knowledge base. At a minimum, contraction must remove the statements which cause the clashes (or cause information to be lost). Following the procedure described in Section 5.2.3, the contraction operation implemented here simply removes all default assertions from the knowledge base, leaving only those statements that were asserted by the ontology's author. The result of the contraction operation is shown in Figure 5.8.

The contraction removed the result of the default domain assertion but left all other remaining statements. The contraction does not necessarily take the knowledge base back to a pre-inference state. The example clearly shows the remnants of the previous inference

TBox	
A	$\sqsubseteq (\forall R.B) \sqcap B \sqcap (\exists S.C)$
R	role
S	role
ABox	
I1	type-of A
I1	type-of $(\forall R.B) \sqcap B \sqcap (\exists S.C)$
I1	type-of $\forall R.B$
I1	type-of B
I1	type-of $\exists S.C$
genid1	type-of C
I1	S genid1

FIG. 5.8. KB After Contraction

in the knowledge base. The contraction only removes the default statements.

Without the default statements, the knowledge base should once again be consistent. The processes which were used to build the default facts can again be applied. For example, the domain and range constraint generation process could be applied to the ontology again.

5.7 Conclusion

The use of default rules to create new statements can lead to unexpected clashes in the knowledge base. These clashes may not be evident until a reasoner is invoked on such a knowledge base. A clash in default statements should be handled differently than a clash based on asserted statements. In the latter, this is considered a true clash, and the knowledge base is described as inconsistent. Conversely, if the clash occurs as a consequence of the default statements then this indicates that the problem arises with the default statements. In order to restore the knowledge base to a consistent state, the knowledge base must be rolled-back or contracted to a valid, consistent state.

As implemented here, contraction describes the process to remove default statements from the knowledge base. The contraction operation must not only remove the statements

added directly by the default rules, but also any statements that were inferred by the reasoner. In order to support this process, the defaultness of the statement must be tracked with the statement. The default descriptor was introduced to show the origin of a statement as being the result of a default statement or the result of inference using a default statement. Contraction can remove any statement tainted by this process.

The inference procedures of the reasoner must be modified to propagate the default descriptor when a default statement is used to produce a new statement. The reasoner must also replace a weaker, default statement with a stronger, non-default statement. A set of minor alterations to the traditional tableau reasoner, specifically the contains and union operations, implement the necessary changes to the reasoner to support propagation of the default descriptor. Further, it was shown that this modification can be implemented without changing the soundness or completeness of the tableau reasoner.

A reasoner was implemented to support a very limited set of default reasoning. A fact derived from another default statement will, itself be default. At the same time, any fact that is non-default will replace an equivalent default fact in the knowledge base. This reasoner also supports a minimal version of contraction which removes all default facts (asserted or inferred) to restore the knowledge base to a consistent state.

This reasoner was not developed as a full default logic reasoning engine. Instead, the intent was to provide the minimal set of features necessary to track default descriptors through the reasoning process. This is more of an engineer's approach than a logicians. The result is a simple reasoning model that maintains the characteristics of the description logic it is built upon, such as completeness, decidability, and efficiency.

The OWL specification does not currently allow a default descriptor to be stored with the concept hierarchy. A simple and efficient modification to the OWL specification to tag each concept with a default descriptor is proposed. If this strategy were accepted to the OWL standard, the OWL language would have a standard notation to track the origin of a

statement and to allow the reasoner to support the contraction operation.

Chapter 6

CONSTRAINT GENERATION ON REAL ONTOLOGIES

This chapter explores the application of the different generation algorithms on a collection of real-world ontologies. In Chapter 4 the algorithms were applied to a few synthetic examples. This chapter will apply the algorithms to a large collection of real world results. It is easy to construct synthetic examples that demonstrate some desired property. This experiment will demonstrate that the generation techniques described here can be applied on real-world ontologies to generate domain constraints.

Section 6.1 presents an overview of the collection of documents and testing environment used for this experiment. Details about how constraints were generated for this experiment are described in Section 6.2. A comparison of the algorithms is presented in Section 6.3. Real world examples are explored in Section 6.4. Runtime performance statistics are discussed in Section 6.5.

6.1 Swoogle

The Swoogle project, created by the eBiquity group at the University of Maryland-Baltimore County, results in a collection of semantic web documents (Ding *et al.* 2004). A snapshot of the Swoogle documents was used for this research. The Swoogle snapshot con-

Table 6.1. Swoogle Statistics

Count	Description
2,236,147	Total Number of Semantic Web Documents
395,584,533	Total Number of Triples

Table 6.2. Swoolge OWL Counts

Count	Document Type
2,236,147	Total SWD's including RDF, FOAF, and OWL
133,920	OWL Documents
11,662	OWL SWD's with properties
7,080	Valid OWL Ontologies

sists of 2,236,147 semantic web documents harvested between January 17, 2005 through April 28, 2007. A summary of these statistics appears in Table 6.1.

OWL documents make up a relatively small subset of the total number of documents in the Swoogle repository. A document was determined to be an OWL document if it imported a namespace abbreviated 'owl.' The set of documents was further reduced to only those documents that contain properties, since they would be of little use to this study otherwise. A summary appears in Table 6.2.

Only 7,080 of the 11,662 OWL ontologies were found to be valid. Some documents could not be loaded by the ontology tools, and can become invalid due to version drift, communications error, or structural errors in the ontology itself. Other documents became invalid as a result of invoking the reasoner because they were inconsistent.

6.2 Building Constraints

Three different algorithms to constraint generation were described in Chapter 4: disjunction, least-common named subsumer (LCNS), and vivification. A program, *JPDomainGenerator* was created to apply each of the three algorithms to a given ontology and to compare the results. In order to assess the performance of the three algorithms, the

process was applied to each of the valid ontologies in the Swoogle database.

The generator was implemented in Java using the Protege API and the Pellet reasoner. The latest 'beta' release of Protege 3.4 was used for the experiment. This version was selected due to its improved support to invoke the reasoner. Prior to this version, the Pellet reasoner was available only through the DIG interface. In this 'beta' release, the Pellet reasoner is available directly through the Protege and OWL APIs. This results in more efficient reasoning and a single process to manage.

The generator operates by loading the ontology into a Protege model, invoking the reasoner to classify the taxonomy and infer new instance types. For each property in the ontology and for each generation type, the domain and range is constructed. The reasoner is used to compute the subsumption relationship between the original property's asserted domain and range, and each of the constraints created by the three generation methods. The detailed results are stored in an XML database for later analysis.

Running *JPDomainGenerator* could take as little as a few seconds to retrieve the ontology from the *SwoogleCache*, invoke the reasoner, and generate and compare the constraints. In some cases, the reasoner would demand an unsatisfiable amount of memory or CPU time. A total of 2GB of heap space and 30 minutes of time was allocated to *JPDomainGenerator* to process each document. If either of these were exceeded then the task would be killed and an error reported in the database.

6.2.1 Test Environment

Initially, the tests were conducted on a pair of computers, one a server for the Swoogle meta-data and cache files; and another to run the JPDomainGenerator. Even though the experiment ran on only 7,000 ontologies, it required multiple weeks to complete. The test environment was shifted to a loosely coupled cluster consisting of 26 8-core PowerMac workstations, with a total of 208 processors and 104GB of RAM. Each ontology was se-

quentially evaluated on one core, with the cluster executing up to 208 evaluations simultaneously.

Using this test environment, all 7,080 OWL ontologies were processed in approximately seven hours. The speed-up is due to the independence of the constraint generation, and the ability of the central server to serve the cached SWD's to over 200 concurrent processors.

6.3 Results

This section describes the results of generating domain and range constraints over the properties in the Swoogle database. For each of the three types of ontology generation algorithms a domain and range was computed. The subsumption relationship between the original and generated constraints was computed and recorded. The results for the generation of domains is shown in Table 6.3, and for ranges in Table 6.4.

In each of the tables, when a constraint is not specified in the ontology it is treated as a special case, that of being unspecified, even though it is equivalent to `owl:Thing`. This makes it possible for an ontology designer to explicitly assert that the domain and range is `owl:Thing`.

6.3.1 Domains

The first row of Table 6.3 on page 107,'*Original Equals Generated*' shows the number of properties in all of the ontologies where the original specified domain constraint is equal, with respect to subsumption checking, to the generated constraint. Only those properties whose constraints are explicitly stated to be something other than `owl:Thing` are included in this category.

In all cases, the finding that the generated constraint matches the original specified

Table 6.3. Domain Comparison: Original to Generated Types

	Relationship	Disjunction		LCNS		Vivified	
		# props	%	# props	%	# props	%
1	Original Equals Generated	801	2.8	833	2.9	808	2.8
2	Original More Specific Than Generated	7	0.0	7	0.0	63	0.2
3	Original More General Than Generated	141	0.5	103	0.4	74	0.3
4	Original $\top$, Generated $\top$	800	2.8	1111	3.8	807	2.8
5	Original $\top$, Generated More Specific	2427	8.4	2112	7.3	2412	8.4
6	Generated $\top$, Original More Specific	27	0.1	71	0.2	25	0.1
7	Property Unused, Original Specified	3201	11.1	3204	11.1	3190	11.0
8	Property Unused, Domain Unspecified	21385	74.0	21406	74.1	21267	73.6
9	Processor Failed	64	0.2	46	0.2	201	0.7
10	Reasoner Failed	49	0.2	9	0.0	53	0.2
Total		28902		28902		28902	

constraint reflects that this algorithm correctly inferred the constraints for the property. The fact that both LCNS and Vivification were able to generate more equal constraints than disjunction is a novel outcome. For example, in one real-world ontology, the specified domain for a property has_pathological_type is defined by the author as Diseases (Advanced Computation Laboratory - Cancer Research UK). The disjunction generator found the set of diseases which are defined in terms of a restriction on this property, namely: $\langle$ Breast_Cancer $\sqcup$ Cancers $\sqcup$ Adenocarcinoma_of_the_Breast $\sqcup \ldots \rangle$. Both the LCNS and Vivification algorithms summarized this disjunction to Diseases, the same as the original specified domain.

The second row of Table 6.3, '*Original More Specific Than Generated*' shows a small number of properties where the original was more specific than the generated one. This result was another novel outcome of this approach. In this case, it shows that the author specified a domain or range that was more specific than the generator's constraint which was inferred from its usage. Stated another way, the author incorrectly constricted the constraint beyond those classes which use it. For example, in another real-world ontology, the author specified the domain for a property $\texttt{minute} - \texttt{of}$ to be the intersection of $\texttt{time} - \texttt{point}$ and $\texttt{calendar} - \texttt{date}$ (Knowledge Media Institute, The Open University). All three generation algorithms constructed a domain of $\texttt{time} - \texttt{point}$. The class $\texttt{time} - \texttt{point}$ includes a restriction: $(\leq\ 1\ \texttt{Thing}\ )$. The class $\texttt{calendar} - \texttt{date}$ is a subclass of $\texttt{time} - \texttt{point}$, and includes its own restriction: $(\leq\ 1\ \texttt{Thing}\ )$, it is not inherited. In this case, the intersection of $\texttt{time} - \texttt{point}$ and $\texttt{calendar} - \texttt{date}$ is the the class $\texttt{calendar} - \texttt{date}$. Any instance of class $\texttt{time} - \texttt{point}$ would become an instance of type $\texttt{calendar} - \texttt{date}$. This result indicates that the disjunction method of constraint generation may be used to detect inconsistent ontology design. The same cannot be said for the other two methods because they may each over-generalize the constraints.

There is a significant difference between the disjunction and LCNS algorithms, each

reporting seven properties in this category; and the Vivification algorithm which reports sixty-three. This is a by-product of the summarization performed by the vivification approach. Where the disjunction of concepts can easily cross different branches in the inheritance tree, the vivification algorithm will summarize these to a set of common top level concepts and any more specific concepts that could not be absorbed. The result is that the Vivification approach will tend to generalize in more cases than the other two methods.

The next row, '*Original More General Than Generated*' is a positive to neutral outcome. The present usage of the ontology contains a property where the original, asserted constraint is more general than its present usage suggests. There are many reasons for this. This may be intentional: the author elected to leave open future possibilities; or this may be unintentional: the author incorrectly specified the constraint to be overly general. There is insufficient evidence to reliably identify which is the case. If the property were specifically left general to support future work, then the combination of the constraint generation process and default reasoning proposed here may help close the semantic gap for the reasoner while leaving open future modifications.

The fourth row, '*Original* $\top$, *Generated* $\top$' is another form of the case that both the generated and specified constraints are equal. In this particular case, they are both equal to `Thing`, which is a group that was excluded in the first row of this table. This is a neutral result and reflects the incomplete or under-specified nature of some real-world ontologies. One interesting outcome of this case is the demonstration of the tendency of the LCNS to summarize to $\top$. Here, about three hundred more properties were summarized to the top concept. This will happen when the constraint must include portions of the inheritance tree which cross branches at the first level. The least-common named subsumer is the top concept. For these approximately three hundred properties $(1111 - 800 = 311)$, this is a negative result for LCNS.

The fifth row, '*Original* $\top$, *Generated More Specific*' shows strong results for the

generation algorithm. These are properties which the ontology author used in a restriction, but did not specify a domain constraint. Because the original constraint was unspecified there is little to compare between it and the generated constraint. It is interesting that the numerical difference between the LCNS and disjunction methods is $(2427 - 2112 = 315)$, which is almost identical to the 311 difference on the previous row. This illustrates how the LCNS tends to over-generalize to the top concept. The main difference between the disjunction and LCNS is that the LCNS tends to summarize to `Thing`. The vivification approach is almost identical to the disjunction approach in the quality of generalization. This is strong empirical evidence of the strength of the vivification algorithm over the other two approaches.

The sixth row, '*Generated* $\top$, *Original More Specific*' can be a neutral or negative result for all three algorithms. For a given property, the ontology's author described constraints on a property, but the constraint generation algorithm created a constraint of $\top$. This is another case of that described in the third row of this table. The same pattern emerges, with the LCNS tending to generate concepts at `Thing` where the other two tend to be more specific; but all three generate results in this column.

In some cases, the generator creates a constraint that is equivalent the original, non-top concept. After invoking the reasoner, the reasoner finds that the generated constraint is equivalent to the top concept. This peculiar situation may indicate an inconsistency in the ontology, namely that a defined concept is found to encompass the whole universe. This is typically not an intended outcome when designing an ontology. In one real example, an ontology's author described a universe consisting of `Person` and its various subclasses (Stanford Medical Informatics). The property in question, `hasAunt` is defined by two classes, `Niece` and `Nephew`. These two classes are defined to be equivalent to `Woman` and `Man`, respectively, and `Person` transitively. The domain was specified by the author as `Person` and based on the defined class restrictions the domain was generated to be `Person`.

The reasoner concluded that the property was equivalent to `Thing`, even though it was specified as non-`Thing`. The problem in this case was that there was a logical error in the definition of the `Person` class such that it included the disjunction of a concept and its negation.

The next two cases, appearing on lines seven and eight, are neutral results for property generation. These two lines count properties that are defined in an ontology but are not used in any class definitions. In the present state of definition of the Semantic Web, it is clear that the majority of property assertions are made without domain constraints and are not tied to any particular class definitions. This illustrates a clear problem in ascribing any semantic meaning to the property or the individuals connected by it. In a graph-theoretic interpretation of these properties, they represent an arc between a pair of individuals with a label that represents some concept that connects them. It is likely that the ontology author fell to the GENSYM fallacy and assumes that the semantic meaning of the property is derived from the name of the property - it is not.

The final two lines of Table 6.3 represent processing errors. The first represents errors of the generator itself. For example, the generator tried to generate a concept that clashed with the ontology, or there was an unspecified programming error, or in some rare cases, ontologies that used data type properties as if they were object properties. The second category of errors is generated by the reasoner itself. When invoking the reasoner, the reasoner is allocated a fixed amount of time and memory to perform (one half-hour of time and 1.5 GB of memory). If the reasoner fails to complete within these resources then it is terminated and counted as an error.

6.3.2 Ranges

Table 6.4 on page 112 shows the same type of results as Table 6.3, but for range constraints on properties. Based on these two tables, the results for range constraints are similar

Table 6.4. Range Comparison: Original to Generated Types

	Relationship	Disjunction		LCNS		Vivified	
		# props	%	# props	%	# props	%
1	Original Equals Generated	231	0.8	248	0.9	255	0.9
2	Original More Specific Than Generated	6	0.0	6	0.0	17	0.1
3	Original More General Than Generated	172	0.6	147	0.5	138	0.5
4	Original $\top$, Generated $\top$	647	2.2	930	3.2	657	2.3
5	Original $\top$, Generated More Specific	2113	7.3	1839	6.4	2097	7.3
6	Generated $\top$, Original More Specific	361	1.2	392	1.4	365	1.3
7	Property Unused, Original Specified	3403	11.8	3428	11.9	3416	11.8
8	Property Unused, Range Unspecified	21824	75.5	21834	75.5	20959	72.5
9	Processor Failed	102	102	63	0.2	955	3.3
10	Reasoner Failed	43	43	15	0.1	43	0.1
Total		28902		28902		28902	

to those for domain constraints. This stems largely from the fact that the set of subjects for a property generalize differently than the set of objects for a property. For example, consider a fictitious assertion, `knowsStuff`, which may have a domain of `Person` but a range of `Thing`. Unsurprisingly, the generation of a domain will not be a top concept, while the range will be. In spite of the numerical differences, the same qualitative relationships hold between the different categories of generation results.

6.3.3 Results Summary

Several of the rows in Tables 6.3 and 6.4 are irrelevant to the performance of the generation method. They are either errors or include properties which are not used. One additional view of the data is shown in Table 6.5. This table shows the same statistics as in the previous two tables; but with the irrelevant data removed.

This table shows that any of the three generation methods can replace non-existent constraints. Each of the three algorithms demonstrated different capability to generalize a constraint at an appropriate level. The conclusion that the LCNS tends to over-generalize those constraints is made even more clear on this table when comparing rows three and four for the three generation methods. Of the three generation methods, this data shows that for both domain and range generation, the disjunction and vivification approaches perform similarly.

6.4 Results of Application in Different Domains

6.4.1 Plant Anatomy

One example that highlights the difference in generalization between the different algorithms can be seen in Figure 6.1. This example comes from the Swoogle repository (Mungall) and is part of an ontology that describes the plant anatomy domain. The ontol-

Table 6.5. Domain and Range Comparison: Original to Generated Types

Relationship	**Domain**					
	Disjunction		LCNS		Vivified	
	# props	%	# props	%	# props	%
Original Equals Generated	801	19.1	833	19.7	808	19.6
Original More General Than Generated	141	3.4	103	2.4	74	1.8
Original $\top$, Generated $\top$	800	19.1	1111	26.3	807	19.5
Original $\top$, Generated More Specific	2427	57.8	2112	49.9	2414	58.5
Generated $\top$, Original More Specific	27	0.6	71	1.7	25	0.6
Total	**4,196**		**4,230**		**4,128**	

Relationship	**Range**					
	Disjunction		LCNS		Vivified	
	# props	%	# props	%	# props	%
Original Equals Generated	231	6.6	248	7.0	255	7.3
Original More General Than Generated	172	4.9	147	4.1	138	3.9
Original $\top$, Generated $\top$	647	18.4	930	26.2	657	18.7
Original $\top$, Generated More Specific	2113	60.0	1839	51.7	2097	59.7
Generated $\top$, Original More Specific	361	10.2	392	11.0	365	10.4
Total	**3,524**		**3,556**		**3,512**	

```
Property: sensu

Original Domain:
Thing

Disjoint Domain:
PO_0006456,PO_0006482,PO_0006460,PO_0006384,PO_0006478,PO_0006318,PO_0006468,
PO_0006450,PO_0006357,PO_0006447,PO_0006383,PO_0006461,PO_0006494,PO_0000041,
PO_0006444,PO_0006451,PO_0006476,PO_0006495,PO_0006462,PO_0006481,PO_0006485,
PO_0006474,PO_0006471,PO_0006499,PO_0006441,PO_0006448,PO_0006477,PO_0006487,
PO_0006455,PO_0006465,PO_0006508,PO_0006443,PO_0006489,PO_0006446,PO_0006507,
PO_0006457,PO_0006470,PO_0006445,PO_0006459,PO_0006467,PO_0006329,PO_0006469,
PO_0006466,PO_0006483,PO_0006458,PO_0006472,PO_0006449,PO_0006473,PO_0006484,
PO_0006454,PO_0006486,PO_0006475,PO_0006464,PO_0006497,PO_0006442,PO_0006453,
PO_0006496,PO_0006463,PO_0006506,PO_0006452,PO_0006493,PO_0006490,PO_0006498,
PO_0006491,PO_0006480,PO_0006500,PO_0006492,PO_0006479

LCNS Domain:
Thing

Vivified Domain:
PO_0009074,PO_0020006,PO_0020101,PO_0009067,PO_0020019,PO_0009066,PO_0020048,
PO_0009073,PO_0009046,PO_0009062,PO_0006204,PO_0009070,PO_0009027,PO_0020026,
PO_0020003,PO_0009014,PO_0009013
```

FIG. 6.1. Constraint Generalization. Example where constraint generation can improve reasoning performance

Table 6.6. Performance Time for Plant Anatomy Processing

Generator	Time (s)
Disjunction	59.8
LCNS	90.0
Vivification	50.3

ogy contains 734 classes, 8 object properties, and 1,068 sub-class axioms.

Table 6.6 timing statistics for each of the three methods. The time reported here is the total time, in seconds, required to load, build constraint for all properties, assign constraints to properties, re-classify the terminology and compute inferred types for all individuals. This follows the same approach described in Section 6.3.

The disjunction method created the most specific and also the longest descriptions compared to the other methods. For example, for the property, `senu`, which is jargon

from the plant anatomy domain, it created a property domain that was the disjunction of 68 different classes. Despite the fact that this method is the most efficient to compute constraints, its overall performance was 19% slower than the vivification method. This is mainly due to the classification costs associated with the large disjunctive sentences present after constraint generation.

The vivification method demonstrated remarkable performance in this example. Its ability to summarize disjunctive sentences enabled it to summarize the long disjunctions into shorter, more concise descriptions which helped improve the performance of the reasoner. For example, for the `sensu` property described above, the vivification approach created a more general concept description consisting of 17 classes in disjunction. This was a 74% reduction in the length of the domain description. As a result of efficient generation and more concise concept descriptions, vivification was the fastest method of the three compared here.

The LCNS algorithm performed quite poorly in this case. The LCNS performed poorly due to the very large number of subsumption queries that were required to summarize the large and complex classes in this particular example. This is largely due to the large number of the concepts and the deepness of the inheritance tree which required more subsumption checks to compute the LCNS. The LCNS constraint for the `sensu` property was over-generalized to `Thing`.

This result clearly shows the vivification method can efficiently create concise concept descriptions, even with a reasonably complex ontology. This result also demonstrates that there is a significant performance benefit to breaking down long disjunctions in real-world ontologies. Finally, it shows that the vivification algorithm out performed the other two methods.

6.4.2 Machine Translated Data

The next example shows some of the challenges working with machine generated ontologies. This example appears to be a machine generated ontology, based on the chosen class names and the poor structure of terminological relations. This ontology is nearly completely flat - there are two subclasses of `owl : Thing`: Person and Thing. There are no class restrictions in this ontology. There are 22 object properties and 44 individuals. While there is a significant amount of data in this ontology, there is very little knowledge encoded in it.

For each of the 22 object properties, the domain and range constraints of this ontology are `owl : Thing`. There is simply not enough information to support the generation process within the taxonomy. This degenerate case is illustrative of the case where a semantic web document is simply a collection of data encoded in OWL rather than a rich taxonomy describing knowledge. In this case, it is unlikely that with additional external information (e.g. a mapping to another ontology) that any generation method will be able to do much better. Because of the lack of any evidence about the use of the properties in the terminology, the algorithms used here were unable to generate a constraint for any of the classes in this ontology.

6.5 Performance Comparison

This section compares the run-time performance of the three algorithms. Run-time performance is important for a number of applications, such as future search engines. Tables 6.7 and 6.8 show a comparison of the run-time for each of the generation algorithms.

Table 6.7 shows run-time performance statistics of each of the generation algorithms. A random sample of 100 ontologies was selected for this analysis. The same ontologies were used for each sample. For each group, the time, in seconds, to load and classify the

Table 6.7. Performance of Generation Algorithms

Algorithm	Statistic	Seconds
None	min	3.80
	max	14.62
	average	5.29
	std. dev	1.71
Disjunction	min	3.85
	max	23.37
	average	5.43
	std. dev	2.29
LCNS	min	3.79
	max	23.92
	average	5.34
	std. dev	2.20
Vivification	min	3.79
	max	22.56
	average	5.29
	std. dev	2.08

Table 6.8. Normalized Performance of Generation Algorithms

Method	Average	Std. Dev
Disjunction	0.22	1.16
LCNS	0.14	0.14
Vivification	0.08	.82

ontology, then generate the constraints, and then classify the modified ontology is reported as a minimum, maximum, average, and standard deviation. The first group, labeled 'None,' reports base-line performance statistics, and omits the constraint generation step; but still uses the reasoner.

This table suggests that the performance of the vivification algorithm is better (lower is better) than the other two. In fact, based on these statistics, the performance of the vivification algorithm is nearest the performance without generation. Standard hypothesis testing shows that due to the high standard deviation, the four results are not significantly different.

The cost to load an ontology and invoke the reasoner on it is approximately constant, and acts as a linear scaling of the data values. Subtracting this fixed cost leads to the results shown in Table 6.8. This table shows the costs, in seconds, of generating the constraints and invoking the reasoner on the result. In this case, the difference between the algorithms is statistically significant. The vivification approach is faster than the disjunction with 92.6% confidence, and is faster than the LCNS with 76.4% confidence.

6.6 Conclusion

A significant collection of real-world ontologies collected from Swoogle were used as the basis for this work. The normal mechanism for retrieving semantic web documents proved unsatisfactory for this work due to a number of different types of errors, especially due to communications and revisions. A 'Semantic Web in a Box' test environment was created to improve the speed of data collection over this large collection of documents. One outcome of this research is that this testing environment can be used for future studies and may serve as a model for future semantic web agents which need to quickly perform tasks for their masters.

Comparing the domain and range constraints generated by the three different algorithms shows significant differences between them. The disjunction algorithm is the simplest of the three in that it does not perform any generalization of the constraint. These results show that while disjunctive constraints provide the most specificity, they also tend to reduce the performance of the reasoner. These results also show that the LCNS approach was relatively poor at creating useful constraints as it tends to over-generalize, in most cases to the top concept. The vivification algorithm demonstrated generalization performance that was nearly identical to the disjunction algorithm. Because concepts were summarized, the long chains of disjunction were broken down by this algorithm reducing

the time and memory required by the reasoner.

Not every ontology lends itself to this technique. Each of these algorithms depends on the availability of a taxonomy which is built with class restrictions over the set of properties. In the extreme case where there is simply a set of classes without any restrictions then these algorithms are unable to generate any constraints at all. As the complexity of the taxonomy and availability of restrictions increases the evidence for constraint generation increases as well. For small ontologies with short chains of disjunctions, both the disjunction and vivification algorithms are appropriate choices. As the size of the disjunctive chains increase, the vivification algorithm is the best all around choice. The LCNS algorithm demonstrated such poor performance its use is discouraged in favor of the other two algorithms.

Chapter 7

CONCLUSION

The research set out to provide a solution to the problem of missing domain and range constraints in OWL ontologies. It was shown that this problem is pervasive: the vast majority of properties of real-world ontologies in present use do not include any domain and range constraints. Several reasons why these constraints are not specified were presented. In many cases, the information is available but not directly encoded in the ontology.

This chapter provides a summary of the major results of this generation process. A summary of the major results appears in Section 7.1 and and overview of possible future work appears in Section 7.2.

7.1 Major Results

Given an ontology, an approach to create constraints was described. This approach is based on inferring the usage of a property from classes defined in terms of restrictions based on the properties. This approach quickly turned to methods of generating constraints. The most direct way to generate a domain constraint from a list of restrictions on a property is to construct a disjunction of the concepts. This research further investigated the tendency of disjunctive constraints to be overly specific and tend towards long chains of disjunctions. This research also demonstrated how these long disjunctions would negatively impact the

performance of the reasoner.

Two methods of summarizing the properties were described: the least-common named subsumer and vivification methods. The least-common named subsumer builds a concept description of the least-common (or least general) named class in the ontology that subsumes every member of the domain or range. The vivification concept creates summaries by allowing partial subsumption checking while constructing the list.

A comparison of the constraints generated by the three algorithms showed that the LCNS algorithm tended to over-generalize and create constraints that were equivalent to the top-concept or else were more general than the other two methods. The same results showed that the vivification algorithm produces constraints that are closer to the specificity of the disjunction algorithm while still creating useful summaries of common super-classes in the constraint. This is beneficial to preserve as much of the available information as possible while breaking the long disjunctive chains that impacted the reasoner's performance.

A time-based comparison showed that the vivification algorithm was statistically faster than the other approaches. This is due to the amortization of the generation costs over the improved reasoner cost of the final model. Thus, for certain ontologies, the vivification algorithm is superior in generalization and in performance.

The problem with applying any of the three algorithms is that it is equivalent to making default assertions and introduces non-monotonicity and potential undecidability into the ontology. A method to address a very limited form of default reasoning was introduced which tracked the origin of a particular statement to be either default or non-default. A set of modifications to the reasoner were described which propagated the defaultness of a statement through any statements which were derived in whole or part from another default statement; and which allowed a non-default statement to replace a default statement. In the event of a clash in the knowledge base or to handle retractions due to non-monotonicity, a retraction operation was built into the reasoner which allowed the default facts (and those

facts derived from them) to be removed from the knowledge base. Because of the limited scope of this modification, the resulting reasoner was demonstrated to be complete, decidable, and efficient.

7.2 Future Work

This research raises some additional questions that can be explored in future work: such as role of instance data, domain and range pairings, use of class names for constructing constraints, comparing ontologies, and improving search results.

One of the primary questions is whether individual instance data can be used to improve the results of the constraint generation process. At the present, there appears to be a significant amount of variability in the amount of type assertions that can be extracted from the instance data. As such, it is not clear that there is sufficient information to support generation of additional constraints (or even to verify the correctness of constraints) from the instantiation data. Assertional evidence may also be useful during contraction, such as in the situation discussed in Section 5.2.4, when there is insufficient evidence in the TBox to properly infer default constructs.

Another open question is how the new OWL 1.1 constraint pattern, where domain and range statements are paired can be combined with this generation method. Specifically, will vivification yield satisfactory results when the language allows multiple domain and range pairings. Even if it does, the method in which the vivified concept is created would need to be modified to take advantage of the new linguistic support.

One other area that warrants some exploration is the use of the class names and external meta-ontologies in constructing the domain and range constraints. The current work ignores the name of the symbols used. However, there are some systems that make use of the symbol names to extend additional meaning into the ontology. This approach, often

used by ontology mapping tools, may allow additional constraints to be generated even when not supported by the usage within the ontology.

One final future application of this research is in the domain of semantic search engines. Presently, search engines such as Google try to aggressively magnify the amount of information from the given documents in the search base. They need to do this to overcome differences in information representation in the various web documents. It is clear from this research that different ontologies have different degrees of completeness and specificity. Further, accurately mapping and linking documents will require a rich ontology with clear semantics. Using this approach, property constraints can be generated during the mapping or linking phases and improve the performance of the mapper and resulting reasoner. Further, this approach can be used by the search agent to simplify and generalize long disjunction chains to gain additional reasoning performance and reduce memory costs.

7.3 Conclusion

The research results presented here strongly support the original research thesis. It is possible to generate domain and range constraints for properties. These constraints may be used to find new information in existing ontologies, and it can be done efficiently. With a few modifications to the reasoner the issues of default reasoning and non-monotonicity can be addressed as well. This research is an exciting beginning for a number of different application areas and future research areas. These methods may be extended into additional areas and future Semantic Web languages.

REFERENCES

[1] Abiteboul, S.; Quass, D.; McHugh, J.; Widom, J.; and Wiener, J. L. 1997. The LOREL query language for semistructured data. In *International Journal on Digital Libraries* (2001) 68–88. ISSN 0302-9743.

[2] Advanced Computation Laboratory - Cancer Research UK. Cancer ontology. Swoogle ID 4645687.

[3] Antoniou, G., and Williams, M. 1998. Revising default theories. In *Proceedings of Tenth IEEE International Conference on Tools with Artificial Intelligence*, 423–430. Taipei, Taiwan: IEEE Press.

[4] Apple, Inc. 2008. iTunes RSS. URL: `http://www.apple.com/rss/`.

[5] Baader, F., and Nutt, W. 2003. *The Description Logic handbook*. New York, NY: Cambridge University Press. chapter Basic Description Logics, 41–74, 273.

[6] Baader, F., and Nutt, W. 2007. *Description Logic Handbook*. Cambridge University Press. chapter Basic Description Logics, 47–104.

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

[8] Berners-Lee, T. 1998. Semantic web road map. WWW. From: `http://www.w3.org/DesignIssues/Semantic.html`.

[9] Brachman, R. J., and Levesque, H. J. 1984. The tractability of subsumption in frame-based description languages. In *AAAI*, 34–37.

[10] Brachman, R. J.; Pigman, V.; Hector, G.; and Levesque, J. 1985. An essential hybrid reasoning system: knowledge and symbol level accounts of krypton. In *In Proceedings of the 9th International Joint Conference on Artificial Intelligence*, 532–539. Morgan Kaufmann.

[11] Brickley, D., and Miller, L. 2008. Friend-of-a-friend (foaf) namespace. URL: `http://www.foaf-project.org`. Website.

[12] Cimiano, P., and Völker, J. 2005. Text2onto. In Montoyo, A.; Muñoz, R.; and Métais, E., eds., *NLDB*, volume 3513 of *Lecture Notes in Computer Science*, 227–238. Springer.

[13] Cohen, W., B. A., and Hirsh, H. 1992. Computing least common subsumers in description logics. *Proceedings of the National Conference on Artificial Intelligence - AAAI* 754–760.

[14] Colmerauer, A. 1993. The birth of prolog. In *III, CACM Vol.33, No7*, 37–52.

[15] Colucci, S.; Noia, T. D.; Sciascio, E. D.; Mongiello, M.; and Donini, F. M. 2004. Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace. In *ICEC '04: Proceedings of the 6th international conference on Electronic commerce*, 41–50. New York, NY, USA: ACM.

[16] Ding, L.; Finin, T.; Joshi, A.; Pan, R.; Cost, R. S.; Peng, Y.; Reddivari, P.; Doshi, V.; and Sachs, J. 2004. *Swoogle: a search and metadata engine for the semantic web*. New York, NY, USA: ACM. 652–659.

[17] Ding, Z. 2005. *BayesOWL: a probabilistic framework for uncertainty in semantic web,*. Ph.D. Dissertation, University of Maryland, Baltimore County.

[18] Doan, A.; Madhavan, J.; Domingos, P.; and Halevy, A. 2002. Learning to map

between ontologies on the semantic web. In *WWW '02: Proceedings of the eleventh international conference on World Wide Web*, 662–673. ACM Press.

[19] Dodds, L. 2008. FOAF-A-Matic. `http://www.ldodds.com/foaf/foaf-a-matic`.

[20] Dou, D.; Mcdermott, D.; and Qi, P. 2002. Ontology translation by ontology merging.

[21] Embley, D. W. 2004. Toward semantic understanding: an approach based on information extraction ontologies. In *CRPIT '27: Proceedings of the fifteenth conference on Australasian database*, 3–12. Australian Computer Society, Inc.

[22] Google. 2006. Web authoring statistics. [Online; accessed 26-January-2006].

[23] Grau, B. C., and Motik, B. 2008. Owl 1.1 web ontology language model-theoretic semantics. Technical report, World Wide Web Consortium.

[24] Horridge, M.; Knublauch, H.; Rector, A.; Stevens, R.; and Wroe, C. 2004. A practical guide to building OWL ontologies using the Protege-OWL plugin and CO-ODE tools edition 1.0. `http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf`. [Online; accessed 27-January-2006].

[25] Horrocks, I.; McGuiness, D. L.; and Welty, A. C. 2003. *The Description Logic handbook*. New York, NY: Cambridge University Press. chapter Digital Libraries and Web-Based Information Systems, 427–460.

[26] Horrocks, I.; Patel-Schneider, P. F.; and van Harmelen, F. 2003. From SHIQ and RDF to OWL: the making of a web ontology language. *Journal of Web Semantics* 1(1).

[27] Horrocks, I. 1999. Fact and ifact. In *In Proceedings of the International Workshop on Description Logics (DL99*, 133–135.

[28] Horrocks, I. 2003. *The Description Logic handbook*. New York, NY: Cambridge University Press. chapter Implementation and Optimization Techniques, 329–377.

[29] Ian Horrocks, Peter F. Patel-Schneider, H. B. S. T. B. G. M. D. 2004. Swrl: A semantic web rule language combining owl and ruleml. URL.

[30] Internet World Stats. 2006. World internet usage statistics and population stats. http://www.internetworldstats.com/stats.htm.

[31] Knowledge Media Institute, The Open University. Untitled. Swoogle ID 598598.

[32] Koller, D.; Levy, A. Y.; and Pfeffer, A. 1997. P-CLASSIC: a tractable probablistic description logic. In *AAAI/IAAI*, 390–397.

[33] McCarthy, J.; Minsky, M. L.; Rochester, N.; and Shannon, C. E. 1955. A proposal for the dartmouth summer research project on Artificial Intelligence. `http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html`. [Online; accessed 07-February-2006].

[34] Mccune, W., and Wos, L. 1997. Otter: The cade-13 competition incarnations. *Journal of Automated Reasoning* 18:211–220.

[35] Modica, G.; Gal, A.; and Jamil, H. M. 2001. *The use of machine-generated ontologies in dynamic information seeking*, Volume 2172 of Batini (2001). 433–448. ISSN 0302-9743.

[36] Modica, G. 2002. *A framework for automatic ontology generation from autonomous web applications*. Ph.D. Dissertation, Mississippi State University.

[37] Mooney, R. J., and Bunescu, R. 2005. Mining knowledge from text using information extraction. *SIGKDD Explor. Newsl.* 7(1):3–10.

[38] Motik, B. KAON2.

[39] Mungall, C. Plant anatomy ontology. Semantic Web.

[40] Nardi, D., and Brachman, R. J. 2003. *The Description Logic handbook*. New York, NY: Cambridge University Press. chapter An introduction to Description Logics, 1–40.

[41] National Hurricane Center. 2008. Atlantic Tropical Storm Outlook. URL: `http://www.nhc.noaa.gov/index-at.xml`.

[42] Nilsson, N. J. 1986. Probabilistic logic. Technical Report 1.

[43] Noy, N., and McGuiness, D. L. 2005. *Ontology development 101: a guide to creating your first ontology*. Stanford University. [On-Line; Accessed 2005].

[44] Noy, N., and Musen, M. 2001. Anchor-PROMPT: using non-local context for semantic matching. In *Workshop on Ontologies and Information Sharing at the Seventeenth International Joint Conference on Artificial Intelligence*. Seattle, WA: IJCAI.

[45] Noy, N. 2005. Order from chaos. *Queue* 3(8):42–49.

[46] Pan, R., D. Z. Y.-Y. P. Y. 2005. A Bayesian network approach to ontology mapping. ISWC2005.

[47] Parsia, B., and Sirin, E. 2004. Pellet: An owl dl reasoner. *In Third International Semantic Web Conference - Poster* 2003.

[48] Pearl, J. 1988. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. San mateo, California: Morgan Kaufmann Publishers, inc. Second Printing.

[49] Pinto, H. S., and Martins, J. P. 2001. A methodology for ontology integration. In *K-CAP 2001: Proceedings of the international conference on Knowledge capture*, 131–138. ACM Press.

[50] Pinto, H. 1999. Some issues on ontology integration. In IJCAI-99 Workshop on Ontologies and Problem-Solving Methods (KRR5).

[51] Russell, S., and Norvig, P. 2003. *Artificial intelligence: a modern approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition.

[52] Schmidt-Schaubß, M., and Smolka, G. 1991. Attributive concept descriptions with complements. *Artif. Intell.* 48(1):1–26.

[53] Schmolze, J. G.; Beranek, B.; and Inc, N. 1985. An overview of the kl-one knowledge representation system. *Cognitive Science* 9:171–216.

[54] Stanford Encyclopedia of Philosphy. 2008. Tarksi's Truth Definitions. URL: `http://plato.stanford.edu/entries/tarski-truth`.

[55] Stanford Medical Informatics, S. U. S. o. M. Family-swrl ontology. Swoogle id 4720353.

[56] Stanford Medical Informatics, Stanford University School of Medicine. 2006. Protege documentation. `http://protege.stanford.edu/plugins/owl/documentation.html`. [Online; accessed 26-January-2006].

[57] Stanford Medical Informatics, S. U. S. o. M. 2006. Protege registration statistics. [Online; accessed 26-January-2006].

[58] SWAD-Europe. 2008. Swad-europe faq. URL: `http://www.w3.org/2001/sw/Europe/`.

[59] Swoogle. 2006. Swoogle Statistics. [Online; accessed 26-January-2006].

[60] Tsarkov, D., and Horrocks, I. 2004. Efficient reasoning with range and domain constraints. In Haarslev, V., and Mller, R., eds., *Proceedings of the 2004 International Workshop on Description Logics (DL2004)*, 41–51.

[61] Velardi, P.; Fabriani, P.; and Missikoff, M. 2001. Using text processing techniques to automatically enrich a domain ontology. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, 270–284. New York, NY, USA: ACM Press.

[62] W3C. 2008. World wide web consortium (w3c). URL: `http://www.w3c.org`.

[63] Weidenbach, C.; Brahm, U.; Hillenbr, T.; Keen, E.; and Theobald, C. 2002. Spass version 2.0. In *In Proc. CADE-18*, 275–279. Springer.

[64] Wikipedia. 2008. Rss entry. URL: `http://en.wikipedia.org/wiki/RSS_(file_format)`.

[65] World Wide Web Consortium. 2001. DAML+OIL March 2001 reference description. [Online; accessed 27-January-2006].

[66] World Wide Web Consortium. 2005a. Extensible markup language standard specification. [Online; accessed 26-January-2006].

[67] World Wide Web Consortium. 2005b. OWL web ontololgy language overview. [Online; accessed 26-January-2006].

[68] World Wide Web Consortium. 2005c. Resource description framework standard. [Online; accessed 26-January-2006].

[69] Wright, A. 2008. Searching the deep web. *Commun. ACM* 51(10):14–15.

[70] WWW Consortium. 2004. RDF Semantics. `http://www.w3.org/TR/rdf-mt`. Tecnical Report.

[71] Zhu, X.; Gauch, S.; Gerhard, L.; Kral, N.; and Pretschner, A. 1999. Ontology-based web site mapping for information exploration. In *CIKM*, 188–194.