A DECLARATIVE APPROACH TO
ONTOLOGY TRANSLATION WITH
KNOWLEDGE PRESERVATION

Author: Óscar Corcho
Advisor: Asunción Gómez-Pérez

April 2004
To my family and friends
I would like to thank Asunción Gómez-Pérez for her help and guidance during all these years. Since 1996, when we started working together, we have enjoyed with deep discussions and long struggles about many aspects of the Ontological Engineering field. Besides, Asun has been always able to find a time slot in her extremely busy schedule to help me to structure the thesis contents, to read the early drafts of this document, and to propose many interesting improvements to it so as to make the ideas expressed here easier to understand.

I am also in debt with current and old members of the Ontological Engineering Group at UPM, with whom I have had many interesting discussions during my and their work and have shared many good moments. Among all of them, I would like to give special thanks to Mariano Fernández-López, who made so many helpful comments to improve this thesis, and who gave me important clues about knowledge representation ontologies and about different options for representing translators declaratively; to Rafael González-Cabero, who was in charge of implementing the RDF(S), DAML+OIL and OWL import services for the WebODE workbench and helped me to evaluate the ontology translation method proposed in this thesis; to Ángel López-Cima, who has been patient enough to cope with my frequent programming errors in all the WebODE ontology translation systems developed; and to David Manzano, who made his large mathematical background available to me whenever I needed it.

I am particularly indebted to Rosario Plaza Arteche for her valuable help in checking and reviewing the grammar and spelling of the manuscript and improving its clarity and style.

Last but not least, I must thank my friends and family for believing in me, and understanding and helping me in my academic career. And among them, I want to thank Katy, for her love and support in everything I do, and for not complaining so much about my constant work during the last period of my thesis work.
Abstract

In Computer Science, an ontology is defined as a formal explicit specification of a shared conceptualisation. An ontology can be modelled in varied ontology tools, and implemented in varied ontology languages. Different ontology tools and languages are based on different knowledge representation formalisms and provide different ontology components to model ontologies.

The ontology translation problem (aka ontology interoperability problem) appears when we decide to reuse an ontology (or part of an ontology) with a tool or language that is different from those ones in which the ontology is available. If we force each ontology-based system developer, individually, to commit to the task of translating and incorporating to their systems the ontologies that they need, they will require a lot of effort and time to achieve their objective.

This thesis presents two contributions to the current state of the art on ontology translation among languages and/or tools:

- It proposes a new model for building and maintaining ontology translation systems, characterised by two main features. First, it identifies four layers where ontology translation decisions can be taken: lexical, syntax, semantic, and semiotic. This layered architecture is based on existing work in formal languages and the theory of signs. Second, it proposes to represent ontology translation decisions declaratively.

- It characterises existing ontology translation approaches from the perspectives of semantic and pragmatic preservation, that is, consequence and intended meaning preservation respectively. It also describes the lifecycle of ontologies in cyclic ontology translation processes, which are defined as successive translations where the initial source and final target formats coincide.
Resumen

Desde la perspectiva de la informática, una ontología se define como una especificación formal y explícita de una conceptualización compartida. Las ontologías se pueden modelar con distintos tipos de herramientas e implementar con distintos tipos de lenguajes de ontologías. Estas herramientas y lenguajes pueden estar basados en diferentes formalismos de representación de conocimientos y proporcionar distintos componentes para modelar ontologías.

El problema de la traducción o de interoperabilidad de ontologías aparece cuando se decide reutilizar una ontología (o parte de ella) con una herramienta o lenguaje distinto de aquellos en los que la ontología está disponible. Si se fuerza a cada desarrollador de sistemas basados en ontologías, de manera individual, a realizar la tarea de traducir e incorporar las ontologías que necesitan en sus sistemas, éstos necesitarán gran esfuerzo y tiempo para alcanzar sus objetivos.

Esta tesis presenta las siguientes dos contribuciones al estado del arte actual en traducción de ontologías entre lenguajes y/o herramientas:

- Se propone un nuevo modelo para la construcción y mantenimiento de sistemas de traducción de ontologías. Este modelo se caracteriza, en primer lugar, por ser multicapa: se identifican cuatro capas donde se pueden tomar decisiones de traducción: léxica, sintáctica, semántica y semiótica; esta arquitectura multicapa está basada en el trabajo existente sobre lenguajes formales y la teoría de signos. Asimismo, el modelo propone representar las decisiones de traducción de manera declarativa.

- Se caracterizan los enfoques de traducción de ontologías existentes desde las perspectivas de la preservación de la semántica y de la pragmática en la traducción. También se describe el ciclo de vida de las ontologías en procesos cíclicos de traducción, que se definen como una secuencia de traducciones sucesivas donde el formato inicial y el final coinciden.
# Table of Contents

*Acknowledgements* ........................................................................................................ iii

*Abstract* ................................................................................................................................ v

*Resumen* ........................................................................................................................... vii

*Table of Contents* ............................................................................................................ ix

*List of Figures* .................................................................................................................. xv

*List of Tables* ................................................................................................................... xix

**Chapter 1 Introduction and Overview** ........................................................................ 1

2.1 What is an ontology? ........................................................................................................ 7

2.2 Which are the main components of an ontology? ........................................................ 11

2.2.1 Modelling heavyweight ontologies using frames and first order logic ..................... 11

2.2.2 Modelling heavyweight ontologies using description logic ..................................... 12

2.2.3 Modelling lightweight ontologies using software engineering and database modelling techniques ......................................................................................................................... 14

2.3 Ontology languages......................................................................................................... 15

2.3.1 Ontology language evolution ..................................................................................... 15

2.3.2 KR language comparison frameworks ...................................................................... 19

2.3.2.1 Formal approaches for KR language comparison ................................................ 19

2.3.2.2 Semi-formal and informal approaches for KR language comparison .................. 21

2.3.3 Comparison of ontology languages ......................................................................... 23
4.4 Method for building an ontology translation system

4.4.1 Activity 1. Feasibility study

4.4.1.1 Task 1.1. Identify ontology translation system scope

4.4.1.2 Task 1.2. Analysis of current ontology translation systems

4.4.1.3 Task 1.3. Ontology translation system requirement definition

4.4.1.4 Task 1.4. Feasibility decision-making and recommendation

4.4.2 Activity 2. Analysis of source and target formats

4.4.2.1 Task 2.1. Describe source and target formats

4.4.2.2 Task 2.2. Determine expressiveness of source and target formats

4.4.2.3 Task 2.3. Compare knowledge models of source and target formats

4.4.2.4 Task 2.4. Describe and compare additional features of source and target formats

4.4.2.5 Task 2.5. Determine the scope of translation decisions

4.4.2.6 Task 2.6. Specify test plan

4.4.3 Activity 3. Design of the translation system

4.4.3.1 Task 3.1. Find and reuse similar translation systems

4.4.3.2 Task 3.2. Propose transformations at the pragmatic level

4.4.3.3 Task 3.3. Propose transformations at the semantic level

4.4.3.4 Task 3.4. Propose transformations at the syntax level

4.4.3.5 Task 3.5. Propose transformations at the lexical level

4.4.3.6 Task 3.6. Propose additional transformations

4.4.4 Activity 4. Implementation of the translation system

4.4.4.1 Task 4.1. Find translation functions to be reused

4.4.4.2 Task 4.2. Implement transformations in the pragmatic level

4.4.4.3 Task 4.3. Implement transformations in the semantic level

4.4.4.4 Task 4.4. Implement transformations in the syntax level

4.4.4.5 Task 4.5. Implement transformations in the lexical level

4.4.4.6 Task 4.6. Implement additional transformations

4.4.4.7 Task 4.7. Declarative specification processing and integration

4.4.4.8 Task 4.8. Test suite execution

Chapter 5 Semantic and Pragmatic Preservation in Ontology Translation

5.1 Transformation approaches and their implications in semantic and pragmatic preservation

5.2 Indirect ontology translation by means of common interchange formats
List of Figures

Figure 2.1. Overview of traditional ontology languages......................................................... 15
Figure 2.2. Ontology markup languages .................................................................................. 16
Figure 2.3. Ontology language evolution .................................................................................. 18
Figure 2.4. Informal comparison of the knowledge models of Protégé-2000 and OWL, using Venn diagrams [Knublauch, 2003] ........................................................................ 22
Figure 2.5. Layered structure of the Semantic Web [Berners-Lee, 1999] ............................... 26
Figure 2.6. Relationships among different characterisations of ontology translation problems, and with the syntax, semantic and pragmatic layers of the theory of signs ......................... 34
Figure 2.7. Comparison of the three architectures for ontology translation between n formats (adapted from [Uschold and Grüninger, 1996]) ................................................................. 36
Figure 4.1. Examples of transformations at the syntactic layer .............................................. 51
Figure 4.2. Examples of transformations at the semantic layer .............................................. 54
Figure 4.3. Examples of transformations at the pragmatic layer .......................................... 59
Figure 4.4. Example of translation decisions to be taken at several layers ........................... 63
Figure 4.5. Proposed development process of ontology translation system ......................... 88
Figure 4.6. Task decomposition of activity 1 (feasibility study) ........................................... 92
Figure 4.7. Task decomposition of activity 2 (analysis of source and target formats) .......... 101
Figure 4.8. Class taxonomy of the OWL DL KR ontology .................................................... 103
Figure 4.9. Task decomposition of activity 3 (design of ontology translation system) ......... 113
Figure 4.10. Translation decisions taken for transforming the scope of WebODE ad hoc relations .......................................................... 117

Figure 4.11. Classes and interfaces necessary for integrating the ontology translation system as a WebODE export service .......................................................... 123

Figure 4.12. Task decomposition of activity 4 (implementation of ontology translation system) .......................................................... 126

Figure 5.1. Semantic preservation transformation of the class Person to Protégé-2000 .... 137

Figure 5.2. Pragmatic preservation transformation of the class Person to Protégé-2000 .... 137

Figure 5.3. Different KIF representations of the same single-valued attribute ................ 142

Figure 5.4: Results of importing to the Ontolingua Server the KIF code generated from LOOM and Ontolingua .................................................................................. 143

Figure 5.5. Different syntaxes used to represent the same knowledge in RDF(S) ............. 145

Figure 5.6. Result of a direct translation from WebODE to Protégé-2000 without additional components or workarounds .......................................................... 154

Figure 5.7. WebODE KR ontology modelled in Protégé-2000 ........................................ 157

Figure 5.8. Result of a direct translation from WebODE to Protégé-2000 as an instantiation of the WebODE KR ontology .......................................................... 158

Figure 5.9. Partial WebODE KR ontology represented in Protégé-2000 mainly by means of metaclasses .................................................................................. 162

Figure 5.10. Result of a direct translation from WebODE to Protégé-2000 using the standard knowledge model of Protégé-2000 and part of the WebODE KR ontology .......................................................... 164

Figure 5.11. Result of a direct translation from WebODE to Protégé-2000 using the standard knowledge model of Protégé-2000 and part of the WebODE KR ontology, hiding classes and customising forms .......................................................... 165

Figure 5.12. Comparison of transformation approaches according to their semantic and pragmatic preservation properties .......................................................... 166

Figure I.1. KIF definition of Disjoint-Decomposition, Exhaustive-Decomposition, and Partition. .................................................................................. 207

Figure I.2. Definition of the class Travel in RDF(S) ........................................................ 215

Figure I.3. Definition of the instance AA7462Feb082002 in RDF(S) ........................ 218
Figure I.4. Definition of a claim in RDF. ................................................................. 219

Figure II.1. Edition of metaclasses with Protégé-2000. ............................. 246

Figure II.2. Edition of the concept Travel with Protégé-2000. ...................... 247

Figure II.3. Edition of the attribute arrival Date with Protégé-2000. ............... 248

Figure II.4. Form edition for the instances of the class Travel with Protege-2000. 249

Figure II.5. WebODE workbench’s service architecture. ............................... 251

Figure II.6. Edition of an instance attribute with the WebODE ontology editor. ... 253

Figure II.7. An ontology view in OntoDesigner. ............................................. 254

Figure II.8. Edition of a formal axiom with WAB. ........................................ 255

Figure II.9. Edition of a rule with WAB. ...................................................... 256

Figure II.10. A fragment of an intermediate representation generated by the WebODE ontology documentation service. ................................................. 256

Figure II.11. A fragment of the concept taxonomy of the travel ontology generated by the WebODE ontology documentation service. ................................................. 257

Figure II.12. WebODE inference engine ..................................................... 258

Figure II.13. Result of evaluating a concept taxonomy with ODEClean. ........... 259
List of Tables

Table 2.1. Common DL constructors and their correspondence with language names. .......... 13
Table 2.2: Semi-formal comparison of ontology language expressiveness. ....................... 23
Table 2.3: Ontology tools’ KR paradigm and underlying KR languages (updated from [Gómez-Pérez et al., 2003]). .................................................................................................................. 30
Table 2.4: Informal comparison of ontology tools’ knowledge model expressiveness (adapted from [Gómez-Pérez et al., 2003]). ................................................................. 31
Table 2.5: Ontology tools’ interoperability (adapted from [Gómez-Pérez et al., 2003]). .......... 32
Table 4.1. Examples of transformations at the lexical layer. ............................................. 50
Table 4.2. List of activities and tasks of the method for developing ontology translation systems. ................................................................................................................. 88
Table 4.3. Task products, techniques, and participants of activity 1 (feasibility study) ....... 92
Table 4.4. Checklist for the feasibility decision making (based on the CommonKADS worksheet OM-5 [Schreiber et al., 2000]). ................................................................. 97
Table 4.5. Checklist for the feasibility decision making of the WebODE export service to OWL DL ..................................................................................................................... 98
Table 4.6. Task products, techniques, and participants of activity 2 (analysis of source and target formats) ............................................................................................................ 101
Table 4.7. Property descriptions of the OWL DL KR ontology. ........................................ 103
Table 4.8. Results of applying the evaluation framework described in [Corcho and Gómez-Pérez, 2000] to the knowledge models of WebODE and OWL DL .......................... 105
Table 4.9. Correspondences between WebODE and OWL DL components. .......................... 109

Table 4.10. Task products, techniques, and participants of activity 3 (design of ontology translation system) ........................................................................................................ 113

Table 4.11. Semantic transformation of WebODE partitions to OWL DL.......................... 119

Table 4.12. WebODE and OWL DL datatype correspondences................................. 120

Table 4.13. Task products, techniques, and participants of activity 4 (implementation of ontology translation system) ................................................................. 126

Table 6.1. Summary of the results obtained in the indirect translation between WebODE and Protégé-2000, with RDF(S) as a knowledge interchange format......................... 173

Table 6.2. Summary of the results obtained in the indirect translation between WebODE and Protégé-2000, with OWL DL as a knowledge interchange format.............................. 175

Table 6.3. Summary of the results obtained in the direct translation between WebODE and Protégé-2000, without additional components for semantic preservation .................. 176

Table 6.4. Summary of the results obtained in the direct translation between WebODE and Protégé-2000, instantiating the WebODE KR ontology represented in Protégé-2000 ..... 178

Table 6.5. Summary of the results obtained in the direct translation between WebODE and Protégé-2000, with additional components for semantic preservation .................. 180
Chapter 1

Introduction and Overview

Ontologies are widely used in the areas of Knowledge Engineering, Artificial Intelligence and Computer Science, in applications related to knowledge management, natural language processing, e-commerce, intelligent information integration, information retrieval, database design and integration, bio-informatics, education, and in new emerging fields like the Semantic Web and the Semantic Grid.

The word “Ontology” is taken from Philosophy, where it means a systematic explanation of being. It was in the last decade when ontologies became relevant as a solution to the knowledge acquisition bottleneck in the development of knowledge-based systems. More specifically, in 1991, when the DARPA Knowledge Sharing Effort [Neches et al., 1991, page 37] envisioned a new way to build knowledge-based systems, based on assembling reusable knowledge components instead of constructing new knowledge bases from scratch. With this new approach, systems would be able to share their declarative knowledge, modelled by means of ontologies, and their problem-solving techniques and reasoning services; and system developers would only need to worry about creating the specialized knowledge and reasoners new to the specific task of their system.

Since then, several definitions for the word “ontology” have been proposed in the literature, many of which are collected and explained in [Guarino and Giaretta, 1995] and [Gómez-Pérez et al., 2003]. One of the most cited definitions is proposed by Studer and colleagues [Studer et al., 1998], based on previous definitions of Gruber [Gruber, 1993a] and Borst [Borst, 1997]: “An ontology is a formal, explicit specification of a shared conceptualization”. Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the
ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group.

Ontologies can be implemented in varied ontology languages, which are usually divided in two groups: classical and ontology markup languages. Among the classical languages used for ontology construction we can cite (in alphabetical order): CycL [Lenat and Guha, 1990], FLogic [Kifer et al., 1995], KIF [Genesereth and Fikes, 1992], LOOM [MacGregor, 1991], OCML [Motta, 1999], and Ontolingua [Gruber, 1992]. Among the ontology markup languages, used in the context of the Semantic Web, we can cite: DAML+OIL [Horrocks and van Harmelen, 2001], OIL [Horrocks et al., 2000], OWL [Dean and Schreiber, 2004], RDF [Lassila and Swick, 1999], RDF Schema [Brickley and Guha, 2004], SHOE [Luke and Hefflin, 2000], and XOL [Karp et al., 1999]. Each of these languages has its own syntax, its own expressiveness, and its own reasoning capabilities, provided by different inference engine. Languages are also based on different knowledge representation paradigms and combinations of them (frames, first order logic, description logic, semantic networks, topic maps, conceptual graphs, etc.).

A similar situation applies to ontology tools: several ontology editors and ontology management systems can be used to develop ontologies. Among them we can cite (in alphabetical order): KAON [Maedche et al., 2003], OilEd [Bechhofer et al., 2001], OntoEdit [Sure et al., 2002], the Ontolingua Server [Farquhar et al., 1997], OntoSaurus [Swartout et al., 1997], Protégé-2000 [Noy et al., 2000], WebODE [Arpírez et al., 2003], and WebOnto [Domingue, 1998]. As in the case of languages, the knowledge models underlying these tools have their own expressiveness and reasoning capabilities, since they are also based on different knowledge representation paradigms and combinations of them. Besides, ontology tools usually export ontologies to one or several ontology languages and imports ontologies coded in different ontology languages.

There are important connections and implications between the knowledge modelling components used to build an ontology in such languages and tools, and the knowledge representation paradigms used to represent formally such components. With frames and first order logic, the knowledge components commonly used to build ontologies are [Gruber, 1993a]: classes, relations, functions, formal axioms, and instances. With description logics, they are [Baader et al., 2003]: concepts, roles, and individuals. With semantic networks, they are: nodes and arcs between nodes.

The ontology translation problem [Gruber, 1993a] appears when we decide to reuse an ontology (or part of an ontology) with a tool or language that is different from those ones in which the ontology is available. If we force each ontology-based system developer,
individually, to commit to the task of translating and incorporating to their systems the ontologies that they need, they will require both a lot of effort and a lot of time to achieve their objectives [Swartout et al., 1997]. Therefore, ontology reuse in different contexts will be highly boosted as long as we provide ontology translation services among those languages and/or tools.

If we refer to the exchange of ontologies between ontology tools, the ontology translation problem is also known as the interoperability problem. In fact, interoperability is defined as "the ability of two or more systems or components to exchange information in a heterogeneous network and use of that information" [IEEE, 1999, page 2]. In the Ontological Engineering field, the term interoperability refers to the ability of different ontology tools to exchange their ontologies.

Many ontology translation systems can be found in the literature. They are mainly aimed at importing ontologies implemented in a specific ontology language to an ontology tool, or at exporting ontologies modelled with an ontology tool to an ontology language. A smaller number of ontology translation systems are aimed at transforming ontologies between ontology languages or between ontology tools.

Since ontology tools and languages have different expressiveness and reasoning capabilities, translations between them are not straightforward nor easily reusable. They normally require to take many decisions at different levels, which range from low layers (i.e., how to transform a concept name identifier from one format to the another) to higher layers (i.e., how to transform a ternary relation among concepts to a format that only allows representing binary relations between concepts).

In this thesis we demonstrate that current ontology translation systems do not usually take into account such a layered structure of translation decisions. Besides, in these systems translation decisions are usually hidden inside their programming code. Both aspects make it difficult to understand how ontology translation systems work.

Furthermore, the wide variety of expressiveness and reasoning capabilities of tools and languages makes rather difficult to exchange ontologies between ontology languages and/or tools without losing knowledge from the original ontologies (aka, with semantic preservation) or without changing their intended meaning (aka, with pragmatic preservation). Not all the ontology components that can be represented in a source format can be easily transformed into a target one, and not all the results of the transformations performed are easy to understand either by human users or by software systems.
This thesis presents our work in ontology translation among languages and/or tools, which has two main contributions:

- It proposes a new **model for building and maintaining ontology translation systems**, characterised by two main features. First, it identifies four layers where ontology translation decisions can be taken: lexical, syntax, semantic, and pragmatic. This **layered architecture** is based on existing work in formal languages and the theory of signs (semiotics) [Morris, 1938]. Second, it proposes to represent ontology translation decisions **declaratively** in the language ODEDialect, which is composed of the languages ODELex, ODESyntax, and ODESem.

- It characterises existing ontology translation approaches from the perspectives of **semantic and pragmatic preservation**, that is, consequence and intended meaning preservation respectively. It also describes the lifecycle of ontologies in cyclic ontology translation processes, which are defined in this thesis as successive translations where the initial source and final target formats coincide, and with one or more intermediate formats.

This document is structured as follows:

**Chapter 2 (State of the Art on Ontological Engineering)** presents a survey of the current state of ontology tools and languages, and how they have evolved during the last decades. We will especially focus on their similarities and differences, on their knowledge models and on their expressiveness and reasoning capabilities. This chapter will also describe the current state of the art on ontology translation: we will present the classifications of ontology translation problems proposed so far, different ontology translation architectures and the existing technology that support them.

**Chapter 3 (Work Objectives)** describes the open research problems and goals, the contributions of this thesis to the state of the art, and the set of work assumptions, hypothesis, and restrictions considered in this work.

**Chapter 4 (A Layered Declarative Ontology Translation Model)** describes in detail the layered and declarative model proposed for building and maintaining ontology translation systems. In this model, ontology translation problems are divided into four layers: lexical, syntax, semantic, and pragmatic, where different types of problems are dealt with. Besides, the model proposes to specify translation decisions at each layer by expressing declaratively how the different ontology components must be transformed from the source format to the target format. For this purpose, we propose to use the ODEDialect language, which is composed of three languages:
ODELex, ODESyntax, and ODESem. Finally, the model proposes a method that can be followed to build ontology translation systems according to the previous features.

**Chapter 5** (*Semantic and Pragmatic Preservation in Ontology Translation*) presents the second contribution, namely how to preserve semantics and pragmatics in the ontology translation process. It analyses several proposals for ontology translation: indirect transformations that use common interchange formats, direct transformations that consist in establishing mappings between the knowledge models of the source and target formats, direct transformations that use knowledge representation ontologies of the source and target formats, and direct transformations that combine direct mappings and knowledge representation ontologies. This chapter presents a comparison between these approaches with respect to their semantic and pragmatic preservation properties in cyclic transformations.

**Chapter 6** (*Experimentation*) describes the experiments performed to evaluate the ontology translation model proposed: ontology translation systems from WebODE to RDF(S), OWL, and Protégé-2000, and vice versa. This chapter contains also the analysis of the results obtained from the use of the different ontology translation architectures proposed in chapter 5 in the ontology translation system between the tools WebODE and Protégé-2000: indirect transformations using KIF, RDF(S) and OWL as interchange languages, and the three types of direct ontology translations proposed.

**Chapter 7** (*Conclusions and Future Work*) presents the main conclusions of this work, emphasising its main contributions. The chapter presents also future work to be performed in the field of ontology translation.

The annexes I and II describe in detail the ontology languages KIF, RDF(S) (the combination of RDF and RDF Schema), and OWL, and the ontology tools Protégé-2000 and WebODE, respectively. These descriptions are included in this thesis because the examples of chapters 4 and 5, and the results presented in chapter 6, are based on them.

The annex III contains the BNF grammars of the three languages that compose ODEDialect: ODELex, ODESyntax, and ODESem.

Finally, the annex IV provides a long summary of the thesis in Spanish.
Chapter 2
State of the Art on Ontological Engineering

This chapter presents a summary of the state of the art on Ontological Engineering that is relevant for this thesis. Section 2.1 describes what an ontology is from the perspective of Computer Science. Section 2.2 deals with the modelling components normally used to define ontologies under different knowledge representation paradigms. Section 2.3 focuses on ontology languages: we show how they have evolved in the last decades, and which are the main similarities and differences between the most relevant ontology languages. Similarly, section 2.4 focuses on ontology tools: we show how ontology tools have evolved, and we describe and compare their main features, especially focusing on their knowledge models and on the ontology languages that they are able to export to and to import from. Section 2.5 is concerned with the ontology translation problem: we show how translation problems have been classified in the literature, the translations architectures proposed so far and the available technology for building ontology translation systems between languages and/or tools. We conclude in section 2.6 with a summary of the main lacks in ontology translation and ontology tools’ interoperability in the current state of the art.

2.1 WHAT IS AN ONTOLOGY?

As commented in the introduction, the word ontology is taken from Philosophy, where it means a systematic explanation of being. Whereas during the 1990s, this word became relevant for the Knowledge Engineering community. [Guarino and Giaretta, 1995] propose to use the words ‘Ontology’ (with capital ‘o’) and ‘ontology’ to refer to the philosophical and Knowledge Engineering senses respectively. Many definitions about what an ontology is have been proposed in the literature. In this section, we will review these definitions and explain the relationships between them.

One of the first definitions was given by Neches and colleagues [Neches et al., 1991], who defined an ontology as follows:
An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary.

This descriptive definition tells what to do to build an ontology, and gives some vague guidelines: this definition identifies basic terms and relations between terms, identifies rules to combine terms, and provides the definitions of such terms and relations. Note that, according to Neches' definition, an ontology includes not only the terms that are explicitly defined in it, but also the knowledge that can be inferred from it.

A few years later, Gruber [Gruber, 1993a] defined an ontology as follows:

An ontology is an explicit specification of a conceptualisation.

This definition became the most quoted in literature and by the ontology community. Based on Gruber's definition, many definitions of what an ontology is were proposed. Borst [Borst, 1997] modified slightly Gruber's definition as follows:

Ontologies are defined as a formal specification of a shared conceptualisation.

Gruber's and Borst's definitions have been merged and explained by Studer and colleagues [Studer et al., 1998] as follows:

An ontology is a formal, explicit specification of a shared conceptualisation. Conceptualisation refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group.

In 1995, Guarino and Giaretta [Guarino and Giaretta, 1995] collected and analysed the following seven definitions:

1. Ontology as a philosophical discipline.
2. Ontology as an informal conceptual system.
3. Ontology as a formal semantic account.
4. Ontology as a specification of a conceptualisation.
5. Ontology as a representation of a conceptual system via a logical theory
   5.1. characterised by specific formal properties.
5.2. characterised only by its specific purposes.

6. Ontology as the vocabulary used by a logical theory.

7. Ontology as a (meta-level) specification of a logical theory.

On that paper, Guarino and Giaretta proposed to consider an ontology as:

*A logical theory which gives an explicit, partial account of a conceptualisation.*

where a conceptualisation is basically the idea of the world that a person or a group of people can have. Though on the surface the notion of conceptualisation is quite similar to Studer and colleagues’ notion, Guarino and Giaretta went a step further because they formalised the notion of conceptualisation and established how to build the ontology by making a logical theory. Hence, strictly speaking, this definition would be only applicable to ontologies developed in logic. Guarino and Giaretta’s was further refined in [Guarino, 1998], with the following definition:

*A set of logical axioms designed to account for the intended meaning of a vocabulary.*

There is another group of definitions based on the process followed to build the ontology. These definitions also include some highlights about the relationship between ontologies and knowledge bases. For example, the definition given by Bernaras and colleagues [Bernaras et al., 1996] in the framework of the KACTUS project [Schreiber et al., 1995] is:

*It [an ontology] provides the means for describing explicitly the conceptualisation behind the knowledge represented in a knowledge base.*

This definition proposes “extracting” the ontology from a knowledge base, which reflects the approach the authors use to build ontologies. In this approach, the ontology is built, following a bottom-up strategy, on the basis of an application knowledge base by means of an abstraction process. As more applications are built, the ontology becomes more general, and, therefore, it moves further away from what would be a knowledge base.

Another strategy for building ontologies is to reuse large ontologies like SENSUS [Swartout et al., 1997] (with more than 70,000 nodes) to create domain specific ontologies and knowledge bases:

*An ontology is a hierarchically structured set of terms for describing a domain that can be used as a skeletal foundation for a knowledge base.*
According to this definition, the same ontology can be used for building several knowledge bases, which would share the same skeleton or taxonomy. Extensions of the skeleton should be possible at the low level by adding domain-specific subconcepts, or at the high level by adding intermediate or upper level concepts that cover new areas. If systems are built with the same ontology, they share a common underlying structure, therefore, merging and sharing their knowledge bases and inference mechanisms will become easier.

Sometimes the notion of ontology is diluted, in the sense that taxonomies are considered full ontologies [Studer et al., 1998]. For instance, UNSPSC1, e-cls@ss2, and RosettaNet3, proposals for standards on the e-commerce domain, and the Yahoo! Directory, a taxonomy for searching the Web, are also considered ontologies [Lassila and McGuinness, 2001] because they provide a consensual conceptualisation of a given domain. The ontology community distinguishes ontologies that are mainly taxonomies from ontologies that model the domain in a deeper way and provide more restrictions on domain semantics. The community calls them lightweight and heavyweight ontologies respectively. On the one hand, lightweight ontologies include concepts, concept taxonomies, relationships between concepts, and properties that describe concepts. On the other hand, heavyweight ontologies add axioms and constraints to lightweight ontologies. Axioms and constraints clarify the intended meaning of the terms gathered on the ontology.

Since ontologies are widely used for different purposes (natural language processing, knowledge management, e-commerce, intelligent integration of information, the Semantic Web, etc.) in different communities (i.e., knowledge engineering, databases and software engineering), Uschold and Jasper [Uschold and Jasper, 1999] provided a new definition of the word ontology to popularise it in other disciplines. Note that the database community as well as the object oriented design community also build domain models using concepts, relations, properties, etc., but most of the times both communities impose less semantic constraints than those imposed in heavyweight ontologies. Uschold and Jasper defined an ontology as:

An ontology may take a variety of forms, but it will necessarily include a vocabulary of terms and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms.

This section has collected the most relevant definitions of the word ontology, which provide different and complementary points of view of the same reality. Some definitions are

1 http://www.unspsc.org/
2 http://www.eclass.org/
3 http://www.rosettanet.org/
independent of the processes followed to build the ontology and of its use in applications, while other definitions are influenced by its development process. The main conclusion to this section would be that ontologies aim to capture consensual knowledge in a generic way, and that they may be reused and shared across software applications and by groups of people. They are usually built cooperatively by different groups of people in different locations.

2.2 WHICH ARE THE MAIN COMPONENTS OF AN ONTOLOGY?

Heavyweight and lightweight ontologies can be modelled with different knowledge modelling techniques and they can be implemented in various kinds of languages [Uschold and Grüninger, 1996]. Ontologies can be highly informal if they are expressed in natural language; semi-informal if expressed in a restricted and structured form of natural language; semi-formal if expressed in an artificial and formally defined language (i.e., Ontolingua [Farquhar et al., 1997], OWL [Dean and Schreiber, 2004]); and rigorously formal if they provide meticulously defined terms with formal semantics, theorems and proofs of properties such as soundness and completeness. According to the definition of [Studer et al., 1998], a highly informal ontology would not be an ontology since it is not machine-readable.

There are important connections and implications between the knowledge modelling components (concepts, roles, etc.) used to build an ontology, the knowledge representation (KR) paradigms (frames, description logic, first order logic) used to represent formally such components, and the languages used to implement the ontologies under a given KR paradigm.

This section presents different techniques that could be applied to model ontologies. Not all of them can represent the same knowledge with the same degree of formality and granularity. The section shows which components are used to model heavyweight ontologies with AI-based approaches that combine frames and first order logic or use description logic, and points out which other techniques could also be used to model lightweight ontologies.

2.2.1 Modelling heavyweight ontologies using frames and first order logic

[Gruber, 1993a] proposed to model ontologies using frames and first order logic. He identified five kinds of components: classes, relations, functions, formal axioms, and instances.

Classes represent concepts, which are taken in a broad sense, that is, they can represent abstract concepts (intentions, beliefs, feelings, etc.) or specific concepts (people, computers, tables, etc.).
Classes in the ontology are usually organized in taxonomies through which inheritance mechanisms can be applied.

In the frame-based KR paradigm, metaclasses can also be defined. Metaclasses are classes whose instances are classes. They usually allow for gradations of meaning, since they establish different layers of classes in the ontology where they are defined.

Relations represent a type of association between concepts of the domain. They are formally defined as any subset of a product of n sets, that is: \( R \subseteq C_1 \times C_2 \times \ldots \times C_n \). In the case of binary relations, the first argument is known as the domain of the relation, and the second argument is the range. For instance, the binary relation Subclass-Of is used for building the class taxonomy.

Relations can be also domain dependent, expressing relationships between the instances of two concepts of a specific domain. Binary relations are sometimes used to express concept attributes (aka slots). Attributes are usually distinguished from relations because their range is a datatype, such as string, number, etc., while the range of relations is a concept.

Functions are a special case of relations in which the \( n \)-th element of the relation is unique for the \( n \)-1 preceding elements. This is usually expressed as: \( F: C_1 \times C_2 \times \ldots \times C_{n-1} \rightarrow C_n \).

According to Gruber, formal axioms serve to model sentences that are always true. They are normally used to represent knowledge that cannot be formally defined by the other components. In addition, formal axioms are used to verify the consistency of the ontology itself or the consistency of the knowledge stored in a knowledge base.

Finally, instances are used to represent elements or individuals in an ontology.

2.2.2 Modelling heavyweight ontologies using description logic

Description Logic (DL) is a logical formalism whose first implementation languages and systems were, as will be described in section 2.3, the following: KL-ONE [Brachman and Schmolze, 1985], Krypton [Brachman et al., 1983], Classic [Borgida et al., 1989], LOOM [MacGregor, 1991] and Kris [Baader and Hollunder, 1991]. A Description Logic theory is divided into two parts: the TBox and the ABox. The TBox contains intensional (terminological) knowledge in the form of a terminology and describes general properties of concepts. The ABox contains extensional (assertional) knowledge, which is specific to the individuals of the

---

4 Inspired by classical first order logic terminology (Hermes, 1973) (Lloyd, 1993), an ontology is sound if and only if it does not allow deducing invalid conclusions. An ontology is complete if and only if it allows deducing all the possible valid conclusions starting from the ontology vocabulary and applying the deduction rules permitted.

5 Names previously used for Description Logic were: terminological knowledge representation languages, concept languages, term subsumption languages, and KL-ONE-based knowledge representation languages.
discourse domain [Baader et al., 2003]. In other words, the TBox contains the definitions of concepts and roles, while the ABox contains the definitions of individuals (instances).

Basically, in DL systems we represent ontologies with three kinds of components: concepts, roles and individuals. **Concepts** in DL have the same meaning as in the frame paradigm: they represent classes of objects. **Roles** describe binary relations between concepts, hence they also allow the description of properties of concepts. Higher arity relations among concepts are also allowed in some DL languages and systems. Finally, **individuals** represent instances of classes.

Concepts and roles are both described with terminological descriptions, which are built from pre-existing terms and with a set of constructors (conjunction, disjunction, negation, value restriction, existential quantification, existential restriction, qualified number restriction, etc.). The choice and combination of the different constructors permit designing different DL languages, as shown in table 2.1. For example, a SHIQ language is a language that combines \((\text{ALC}_R)\) intersection, value restriction, limited existential quantification, the concepts \(\text{top}\) and \(\text{bottom}\), atomic negation, negation, union, existential restriction, and transitive roles; with \((H)\) role hierarchies; \((I)\) inverse roles; and \((Q)\) qualified number restrictions. The different combinations of constructors give different expressiveness/reasoning tradeoffs to the corresponding language [Baader et al., 2003].

<table>
<thead>
<tr>
<th>Construct</th>
<th>Syntax</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept</td>
<td>(\Delta)</td>
<td>FL(_4)</td>
</tr>
<tr>
<td>Role name</td>
<td>(R)</td>
<td>FL(_\bullet)</td>
</tr>
<tr>
<td>Intersection</td>
<td>(C \land D)</td>
<td>AL</td>
</tr>
<tr>
<td>Value restriction</td>
<td>(\forall R.C)</td>
<td>S(^7)</td>
</tr>
<tr>
<td>Limited existential quantification</td>
<td>(\exists R)</td>
<td></td>
</tr>
<tr>
<td>Top or Universal</td>
<td>(T)</td>
<td></td>
</tr>
<tr>
<td>Bottom</td>
<td>(\bot)</td>
<td></td>
</tr>
<tr>
<td>Atomic negation</td>
<td>(\neg A)</td>
<td></td>
</tr>
<tr>
<td>Negation(^#)</td>
<td>(\neg C)</td>
<td>C</td>
</tr>
<tr>
<td>Union</td>
<td>(C \lor D)</td>
<td>U</td>
</tr>
<tr>
<td>Existential restriction</td>
<td>(\exists R.C)</td>
<td>E</td>
</tr>
<tr>
<td>Number restrictions</td>
<td>((\geq n R)) (\leq n R))</td>
<td>N</td>
</tr>
<tr>
<td>Nominals</td>
<td>{a(_1) ... a(_n)}</td>
<td>O</td>
</tr>
<tr>
<td>Role hierarchy</td>
<td>(R \subseteq S)</td>
<td>H</td>
</tr>
<tr>
<td>Inverse role</td>
<td>(R^#)</td>
<td>I</td>
</tr>
<tr>
<td>Qualified number restriction</td>
<td>((\geq n R.C)) (\leq n R.C))</td>
<td>Q</td>
</tr>
</tbody>
</table>

\(^6\) In this table, we use \(\Delta\) to refer to atomic concepts (concepts that are the basis for building other concepts), \(C\) and \(D\) to any concept definition, \(R\) to atomic roles and \(S\) to role definitions. FL is used for structural DL languages and AL for attributive languages [Baader et al., 2003].

\(^7\) S is the name used for the language \(\text{ALC}_R\), which is composed of \(\text{ALC}\) plus transitive roles.

\(^8\) \(\text{ALC}\) and \(\text{ALCUE}\) are equivalent languages, since union \((U)\) and existential restriction \((E)\) can be represented using negation \((C)\).
Concepts in DL can be primitive (if they are defined specifying necessary conditions for the individuals of that concept) or defined (if they are defined specifying both necessary and sufficient conditions that must be satisfied by individuals of the concept).

Roles in DL can be either primitive or defined (also called derived). Many DL systems do not permit defining derived roles, because of their reasoning disadvantages, though they do permit the creation of role hierarchies. Regarding n-ary roles, some DL systems do not permit defining them, but only binary ones.

Formal axioms in DL use a subset of the constructs of first order logic. They are usually embedded in concept or role definitions.

Individuals represent instances of concepts and the values of their roles (properties). DL systems usually separate individuals from concepts and roles descriptions. The former are included in the ABox (assertional knowledge) while the latter are included in the TBox (terminological knowledge).

In some DL systems, the ABox also contains rules that allow inferring information. There have been some discussions about the most convenient place for rules, and we think rules should be placed in the TBox, as they really define intensional rather than assertional knowledge.

The last comment about DL is on the reasoning capabilities of this paradigm. The decision about which constructs to use in a DL language has strong effects not only on its expressiveness, but also on its reasoning power. Reasoning in DL is mainly based on concept subsumption, for which DL systems provide efficient automatic classifiers. From these subsumption tests, DL systems derive concept satisfiability and consistency in the models represented. These classifiers are commonly built by means of tableaux calculus and constraint systems.

2.2.3 Modelling lightweight ontologies using software engineering and database modelling techniques

Other techniques widely used in software engineering and databases for modelling concepts, relationships between concepts, and concept attributes, such as Entity-Relationship diagrams or UML class diagrams, could also be appropriate for building lightweight ontologies. These techniques impose a structure to the domain knowledge and constrain the interpretations of terms.
However, it is important to remark that the model can only be considered an ontology if it is a shared and consensual knowledge model agreed by a community. The use of these techniques for building lightweight ontologies is described in [Gómez-Pérez et al., 2003; chapter 1].

2.3 ONTOLOGY LANGUAGES

Several ontology languages have been created during the last decades and other general Knowledge Representation (KR) languages and systems have been used for implementing ontologies though these were not specifically created for this purpose. In this section we first show how the most representative ontology languages have evolved in this period. Then we describe different formal and informal approaches used to compare their representation and/or reasoning capabilities. Finally, we compare those languages using an informal approach, which has been used successfully for determining the language or languages that best fit the representation and reasoning needs of several ontology-based applications.

2.3.1 Ontology language evolution

At the beginning of the 1990s, a set of AI-based ontology languages was created. Basically, the KR paradigms underlying such ontology languages were based on first order logic (e.g., KIF), on frames combined with first order logic (e.g., CycL, Ontolingua, OCML and FLogic), and on description logic (e.g., LOOM). OKBC was also created as a protocol to access ontologies implemented in different languages with a frame-based KR paradigm. The overall layout of these languages is shown in figure 2.1.

![Figure 2.1: Overview of traditional ontology languages.](image)

Of the previous set of languages, CycL [Lenat and Guha, 1990] was the first to be created. CycL is based on frames and first order logic and was used for building the Cyc Ontology.
KIF [Genesereth and Fikes, 1992][NCITS, 1998] was created later, in 1992, and was designed as a knowledge interchange format; KIF is based on first order logic, with some extensions to deal with sets, numbers, etc. Since ontologies were difficult to create directly in KIF, Ontolingua [Farquhar et al., 1997] was created on top of it. Ontolingua has a Lisp-like syntax and its underlying knowledge representation (KR) paradigm are frames and first order logic. Ontolingua was considered a standard de facto by the ontology community in the 1990s.

At the same time, LOOM [MacGregor, 1991] was built, though it was not intended to implement ontologies but for general knowledge bases. LOOM is based on description logic (DL) and production rules and provides automatic concept classification features. OCML [Motta, 1999] was developed later, in 1993, as a kind of “operational Ontolingua”. In fact, most of the definitions that can be expressed in OCML are similar to the corresponding definitions in Ontolingua. OCML was built for developing executable ontologies and models in problem solving methods (PSMs). Finally, in 1995 FLogic [Kifer et al., 1995] was released (although its development started several years before) as a language that combined frames and first order logic though it did not have Lisp-like syntax.

In the spring of 1997, the High Performance Knowledge Base program (HPKB)\(^9\) was started. This research program was sponsored by DARPA and its objective was to solve many of the problems that usually appear when dealing with large knowledge bases (concerning efficiency, content creation, integration of the content available in different systems, etc.). One of the results of this program was the development of the OKBC (Open Knowledge Base Connectivity) protocol [Chaudhri et al., 1998]. This protocol allows accessing knowledge bases stored in different Knowledge Representation Systems, which may be based on different KR paradigms. Of the languages aforementioned Ontolingua, LOOM and CycL are OKBC compliant.

---

The boom of the Internet led to the creation of ontology languages for exploiting the characteristics of the Web. Such languages are usually called web-based ontology languages or ontology markup languages. Their syntax is based on existing markup languages such as HTML [Raggett et al., 1999] and XML [Bray et al., 2000], whose purpose is not ontology development but data presentation and data exchange respectively. The relationships between these languages are shown in figure 2.2.

The first ontology markup language appeared in 1996. Such language was SHOE [Luke and Heflin, 2000]. SHOE is a language that combines frames and rules. It was built as an extension of HTML. It used tags different from those of the HTML specification, thus allowing the insertion of ontologies in HTML documents. Later its syntax was adapted to XML.

The rest of ontology markup languages are based on XML. XOL [Karp et al., 1999] was developed as a XMLization of a small subset of primitives from the OKBC protocol, called OKBC-Lite. RDF [Lassila and Swick, 1999] was developed by the W3C (the World Wide Web Consortium) as a semantic-network based language to describe Web resources. Its development started in 1997, and RDF was proposed as a W3C Recommendation in 1999. The RDF Schema [Brickley and Guha, 2004] language was also built by the W3C as an extension to RDF with frame-based primitives. This language was proposed as a W3C Candidate Recommendation in 2000 and then it suffered a major revision in November 2002, so that its reference document was published as a W3C Working Draft. Later, it was revised in January 2003 and proposed again as a W3C Recommendation in February 2004. The combination of both RDF and RDF Schema is normally known as RDF(S).

These languages have established the foundations of the Semantic Web [Berners-Lee, 1999]. In this context three more languages have been developed as extensions to RDF(S): OIL, DAML+OIL, and OWL. OIL [Horrocks et al., 2000] was developed at the beginning of the year 2000 in the framework of the European IST project On-To-Knowledge. It adds frame-based KR primitives to RDF(S) and its formal semantics is based on description logic. DAML+OIL [Horrocks and van Harmelen, 2001] was created later (between the years 2000 and 2001) by a joint committee from the US and the EU in the context of the DARPA project DAML. It was based on the previous DAML-ONT specification, which was built at the end of 2000, and on OIL. DAML+OIL adds DL-based KR primitives to RDF(S). In 2001 the W3C formed a working group called Web-Ontology (WebOnt) Working Group. The aim of this

---

10 http://www.ontoknowledge.org/
11 http://www.daml.org/
12 http://www.w3.org/2001/sw/WebOnt/
There are many ontology languages involved in this figure:
- KL-ONIE, KL-ONE, TURTLE, N3, OWL, and others.

Figure 2.3: Ontology language evolution
group was to make a new ontology markup language for the Semantic Web. The result of their work is the OWL language [Dean and Schreiber, 2004], which is a W3C Recommendation (February 2004). OWL covers most of the features of DAML+OIL and has renamed most of the primitives that appeared in that language.

Figure 2.3 shows graphically how these languages have evolved, and shows also their relationships with other existing KR languages and systems.

2.3.2 KR language comparison frameworks

Not all of the existing languages have the same expressiveness nor do they reason the same way. KR paradigms underlying ontology languages are diverse: frames, description logic, first (and second) order logic, semantic networks, etc.; and some languages combine more than one KR paradigm. This fact makes very important the correct selection of the language in which the ontology is to be implemented. In fact, a KR paradigm may prove to be appropriate for a specific task but not for others carried out in an application.

Consequently, before coding an ontology the ontology developer should detect first what is needed, in terms of expressiveness and reasoning, and then which languages satisfy such requirements. Several formal and informal approaches have been proposed to compare KR languages and systems. They usually distinguish two dimensions, knowledge representation and reasoning mechanisms, and deal with features from one or both of them.

2.3.2.1 Formal approaches for KR language comparison

According to the theory of signs (aka semiotics) [Morris, 1938], there are three aspects to be considered in the formal study of a language and of communication:

- Syntax: it deals with how symbols are structured.
- Semantic: it deals with what symbols mean.
- Pragmatic: it deals with how symbols are interpreted or used, considering features such as the environment and the mental state of the agents involved in a communication, which are external to the language.

The three aspects are tightly interrelated, although the meaning of a communication is, intuitively, a combination of the two latter aspects: semantic and pragmatic.

Existing formal approaches for KR language comparison are mainly focused on the semantic aspect of languages. They are normally used to prove that certain expressions in a language can
or cannot be expressed in another language. Therefore they allow measuring, to a certain extent, the amount of knowledge that would be lost and preserved in the translation process between languages\textsuperscript{13}. Basically, the following three approaches can be distinguished:

[Baader, 1996] proposes to define formally the expressive power of a language by means of the language semantics, which can be determined with different techniques, such as model theory or axiomatic semantics. Once the semantics of two languages are determined, it is possible to analyse whether they have the same expressive power or not. Two languages have the \textit{same expressive power} if they share the same models, taking into account two translation functions:

- $\psi: \text{Pred}(\Gamma_1) \rightarrow \text{Pred}(\Gamma_2)$, which transforms the predicates of the source language to the target language, plus any auxiliary predicates that may be needed in the translation process.

- $\chi: \Gamma_1 \rightarrow \Gamma_2$, which transforms any expression of the source language to the target language.

Similarly, [Borgida, 1996] defines \textit{the same expressiveness}\textsuperscript{14} relation as follows: the language $\Gamma_2$ is as expressive as the language $\Gamma_1$ if there is a total function $\text{trans} : \Gamma_1 \rightarrow \Gamma_2$ such that for every sentence $L$ in $\Gamma_1$, $\text{trans}(L)$ expresses the meaning of $L$ in $\Gamma_2$. The meaning of a formula is obtained by means of an interpretation function, as usually done in first order logic.

[Euzenat and Stuckenschmidt, 2001] and [Stuckenschmidt, 2003] define formally the \textit{coverage relation} between languages, as a relation that holds between $\Gamma_1$ and $\Gamma_2$ whenever there exists a mapping $\tau: \Gamma_1 \rightarrow \Gamma_2$ that transforms every expression of $\Gamma_1$ into an expression of $\Gamma_2$. We should notice that there is a difference with respect to the previous definitions, since the existence of a translation function is enough to determine that a language covers another, without taking into account whether the transformation preserves semantics or not.

These formal approaches are useful to determine whether an expression in a language can be expressed in another language or not, by giving a formal proof of the transformation to be performed. However, they are tedious to use to determine whether all the expressions of a language can be expressed in another, and they do not specify how the translation function has to be created. In fact, only the latter approach has been used in ontology translation, with the aim of determining the transformation preservation properties of ontology translation systems in families of ontology languages, as described in section 2.5.

Moreover, these approaches require that the languages to be compared have formal semantics. Some KR languages and most of the ontology tools' knowledge models lack from formal

\textsuperscript{13} Similarly, these approaches can be applied to the comparison of the knowledge models of ontology tools.

\textsuperscript{14} Different authors use different terms to refer to the fact that two languages can express the same knowledge: same expressive power, same expressiveness, and language coverage.
State of the Art on Ontological Engineering

semantics (e.g., OKBC, SHOE, XOL, OntoEdit, Protégé-2000, WebODE, etc.). This lack of formal semantics is a source of confusion and misunderstanding, making their formal comparison impossible [Hayes, 1974].

2.3.2.2 Semi-formal and informal approaches for KR language comparison

To overcome the difficulties of the previous formal approaches, several complementary semi-formal and informal approaches have been proposed to show the similarities and differences of the knowledge models of ontology languages and tools.

[Corcho and Gómez-Pérez, 2000] proposed the first semi-formal comparison approach, which consists in a comparison framework of the expressiveness and reasoning capabilities of ontology languages, based on the ontology components most commonly used in frame-based systems and on the reasoning mechanisms commonly present in ontology languages. The advantages of this approach is that it offers at a glance a coarse-grained comparison of languages, which helps in finding their main similarities and differences. The main features that are compared in this framework for each dimension are described below:

With respect to the knowledge representation (language expressiveness) dimension, the framework deals with concepts and their attributes. It distinguishes two types of attributes: instance attributes (which describe concept instances and can take their values in those instances) and class attributes (which describe the concept itself and take their values in it). It also shows how attributes can be constrained according to their type of value, cardinalities (how many values the attribute can take), and other constraints such as the addition of procedural knowledge. It also shows whether the language allows representing metaclasses or not.

The framework also covers concept taxonomies. It analyses whether a language allows expressing subclass-of relationships, disjoint and exhaustive decompositions and partitions.

Relations are very important components in ontology modelling as they describe the relationships that can be established between concepts and between the instances of those concepts. They are usually called relations or slots in frame-based languages and are known as roles or properties in DL-based languages. The framework analyses whether the language allows expressing binary and n-ary relations, relation hierarchies and general integrity constraints of relations.

The framework deals also with functions, in case they can be defined in the language. It analyses whether binary and n-ary functions can be defined in the language.
Finally, it deals with other ontology components, such as formal axioms, which represent sentences that are always true, and which are normally used to represent general restrictions of the domain or pieces of knowledge that cannot be implemented in the ontology with the rest of modelling components. Other components dealt with by the framework are instances, rules, procedures, etc.

With respect to the reasoning dimension, the framework deals with several aspects that can be found in their associated inference engines, such as the availability of automatic classifiers that compute the concept taxonomy automatically from the ontology concept definitions, the management of simple and/or multiple inheritance of concept attributes and relations through concept taxonomies, the management of exceptions in concept taxonomies (this is also called non-monotonic reasoning), and other constraint checking functionalities for detecting inconsistencies in the ontology.

Figure 2.4 shows an informal comparison approach used to compare the knowledge models of Protégé-2000 and OWL. This approach uses Venn diagrams to show which components can be defined in both formats, and which ones are specific of each format. From the figure, it can be inferred that classes, slots, instances, etc., can be defined in both formats (although they have in some cases different names, such as slots for Protégé-2000 and properties for OWL); that default values, PAL constraints, etc., can be defined in Protégé-2000 but not in OWL; and that transitive slots, disjointness between classes, etc., can be defined in OWL but not in Protégé-2000.

![Figure 2.4: Informal comparison of the knowledge models of Protégé-2000 and OWL, using Venn diagrams [Knublauch, 2003].](image-url)
2.3.3 Comparison of ontology languages

Table 2.2 shows the results of comparing the expressiveness of the most relevant ontology languages with the semi-formal comparison approach. This study does not aim to show an exhaustive list of features of all these languages, since each of them is based on different KR paradigms and has its own distinguishing features. Neither does it aim to establish a score of languages (in the sense of "language L₁ is better than language L₂"), as different, and sometimes incompatible, features may be needed for different ontology-based applications. It only aims to help understand better the similarities and differences between these languages.

Cells in the table are filled using '+' to indicate that it is a supported feature in the language, '-' for non supported features, and 'W' for non supported features that could be supported with some workaround. Examples of such workarounds are described in [Gómez-Pérez et al., 2003], and an extended version of this study can be found in [Corcho and Gómez-Pérez, 2000][Gómez-Pérez and Corcho, 2002], with the exception of DAML+OIL and OWL DL in the first reference, and of OWL DL in the second one.

Table 2.2: Semi-formal comparison of ontology language expressiveness.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Ontolingua</th>
<th>LOOM</th>
<th>OKBC</th>
<th>OCML</th>
<th>FLogic</th>
<th>SHOE</th>
<th>XOL</th>
<th>RDF(S)</th>
<th>OIL</th>
<th>DAML+OIL</th>
<th>OWL DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concepts</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Metaclasses</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Attributes</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Instance attributes</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Class attributes</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>W</td>
<td>-</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>Facets</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Type constraints</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Cardinality constraints</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>W</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Procedural knowledge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Concept Taxonomies</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Subclass-Of</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Disjoint-Decomposition</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>W</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Exhaustive-Decomposition</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>W</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>Partition</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>W</td>
</tr>
<tr>
<td>Relations</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Binary relations</td>
<td>+</td>
<td>+</td>
<td>W</td>
<td>+</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>n-ary relations</td>
<td>+</td>
<td>+</td>
<td>W</td>
<td>+</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>Relation hierarchies</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>W</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Integrity constraints</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Functions</td>
<td>+</td>
<td>+</td>
<td>W</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>W</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Binary functions</td>
<td>+</td>
<td>+</td>
<td>W</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>W</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>n-ary functions</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Other Components</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Instances</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Formal Axioms</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Rules</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Procedures</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
The analysis of the table reveals the different expressive power of each language. There are relevant differences in expressiveness between traditional languages and ontology markup languages. While some languages permit representing heavyweight ontologies (with formal axioms, functions, rules, procedures, and other rich modelling components), others only permit representing lightweight ontologies (basically, concepts, concept taxonomies, and relations between them). In fact, most of the modelling components can be represented in traditional languages (such as Ontolingua, LOOM and OCML), while ontology markup languages do not normally provide means to represent all of them.

The following comments can be made on each of the main features presented in the table.

The table shows that metaclasses can be created in half of the languages. OWL DL does not allow defining metaclasses, but OWL Full allows them. As described in chapter 5, metaclasses can be useful in ontology translation for some of the translation approaches proposed.

All the ontology languages allow representing concepts and their attributes (aka slots, roles, and properties). Instance attributes define concept properties whose value(s) are assigned by each concept instance. They can be defined in all the languages. Class attributes specify properties of the concept itself. There are no specific primitives or keywords to represent them in SHOE, RDF(S), and in none of the DL-based languages (LOOM, OIL, DAML+OIL, and OWL DL). This is due to the fact that reasoning with these attributes in description logic is difficult. However, they can be represented in RDF(S), DAML+OIL, and OWL (which becomes OWL Full) with a workaround, as specified in the table. This workaround consists in defining them as properties whose domain is the class rdfs:Class, daml:Class, and owl:Class respectively, and specifying their value(s) in the concept definition where the class attribute takes a value.

If we deal with facets (attribute constraints), we can see that all the languages allow constraining the type of attributes. Regarding cardinality constraints, both minimum and maximum cardinalities can usually be represented, except for SHOE and RDF(S), where they cannot, and FLogic, where minimum cardinalities cannot be represented and maximum cardinalities can only be 1 or N (unspecified). Table 2.2 also shows whether it is possible to attach procedures to attributes or not. Only LOOM and OCML allow representing that kind of knowledge.

Concept taxonomies allow structuring the domain concepts in a hierarchical way. The subclass relationship between concepts is considered in all the languages since it is the basic way to define these hierarchies. More complex knowledge such as partitions, and disjoint and exhaustive concept decompositions cannot be represented in languages like OKBC, SHOE, XOL, and RDF(S).
Sometimes, the same knowledge can be represented with different modelling components. For example, disjointness in DAML+OIL and FLogic is implemented differently: DAML+OIL provides special KR primitives to represent disjointness, while to represent disjointness in FLogic we must use formal axioms. That is the reason why the corresponding cell in DAML+OIL contains the symbol ‘+’ and the cell in FLogic contains the symbol ‘W’.

Another example is related to partitions. In DAML+OIL, partitions are defined with the primitive `daml:disjointUnionOf` (‘+’). In OWL, partitions must be defined in two steps (‘W’): first, expliciting that the concepts in the partition are disjoint; and second, defining the parent concept as the union of all the concepts in the partition.

All languages allow representing binary relations between concepts. In many languages, attributes and relations are represented with the same primitives, although they have different ranges: datatypes and concepts respectively. Relations of higher arity must be normally created by reification (representing them as concepts).

Relation hierarchies can be represented in the DL-based languages, plus Ontolingua and OCML. In FLogic, these hierarchies can be represented with formal axioms (‘W’). With regard to more complex constraints, such as general integrity constraints on relations, they can be only expressed by those languages that allow representing formal axioms.

**Functions** are very similar to relations. There are important differences between languages with respect to the arity of functions. Except for SHOE and RDF(S), binary functions can be defined (either directly or using workarounds) in all the languages. However, n-ary functions cannot be represented in ontology markup languages, but can be represented in all the classical languages except for OKBC.

Finally, we will comment on the other ontology components: instances, formal axioms, rules, procedures, etc.

With regard to **instances**, all the languages allow creating instances of the concepts and of the relations defined in an ontology. In the case of RDF(S), OIL, DAML+OIL and OWL, instances are represented in RDF.

As for **formal axioms**, most of the ontology markup languages (except SHOE and XOL) rely on a future "logic layer" that must be built on top of them, and that will provide extra logical features not provided by the languages themselves. This layered structure is shown in figure 2.5, which complements what was shown in figure 2.2: the lower part contains the XML language, which acts as the syntax of all the ontology markup languages; RDF and RDF Schema are defined on top of XML; OIL, DAML+OIL, and OWL are included in the ontology support.
layer, which is defined on top of RDF Schema; and the logic layer is on top of this ontology support layer.

With regard to rules, some languages include them as a useful extension to allow users to perform more inferences with the language. In the case of LOOM and OCML, the chaining mechanism of these rules (backward or forward) can be determined in the rule definition. In ontology markup languages, the logic layer will also provide support for rules, as shown in figure 2.5.

Finally, procedures can be created in Ontolingua, LOOM and OCML. These three languages are based on Lisp. Hence, procedures in these languages must be defined in Lisp and can be executed in the KR systems associated to them.

2.4 ONTOLOGY TOOLS

To build ontologies is complex and time consuming, and it is even more if ontology developers have to implement them directly in an ontology language, without any kind of tool support. To ease this task, in the mid-1990s the first ontology building environments were created. They provided interfaces that helped users carry out some of the main activities of the ontology development process, such as conceptualisation, implementation, consistency checking, and documentation. In the last few years, the number of ontology tools has greatly increased and they have been diversified. [Gómez-Pérez, 2002] distinguishes the following groups:
• **Ontology development tools.** This group includes tools and integrated suites that can be used to build a new ontology from scratch. In addition to the common edition and browsing functions, these tools usually give support to ontology documentation, ontology export and import to/from different formats and ontology languages, ontology graphical edition, ontology library management, etc.

• **Ontology evaluation tools.** They are used to evaluate the content of ontologies and their related technologies. Ontology content evaluation tries to reduce problems when we need to integrate and use ontologies and ontology-based technology in other information systems.

• **Ontology merge and alignment tools.** These tools are used to solve the problem of merging and aligning different ontologies in the same domain.

• **Ontology-based annotation tools.** With these tools users can insert instances of concepts and of relations in ontologies and maintain (semi)automatically ontology-based markups in Web pages. Most of these tools have appeared recently, in the context of the Semantic Web.

• **Ontology querying tools and inference engines.** These allow querying ontologies easily and performing inferences with them. Normally, they are strongly related to the language used to implement ontologies.

• **Ontology learning tools.** They can derive ontologies (semi)automatically from natural language texts, as well as semi-structured sources and databases, by means of machine learning and natural language analysis techniques.

Some tool suites integrate tools from different groups, and some other isolated tools provide a limited set of functions. But this section will be focused only on the first group of ontology tools: those strictly used for ontology development.

### 2.4.1 Ontology development tool evolution

Ontology development tools' technology has improved enormously since the creation of the first environments. Taking into consideration the evolution of ontology development tools since they appeared in the mid-1990s, two groups can be distinguished\(^\text{15}\):

• Tools whose knowledge model maps directly to an ontology language. These tools were developed as ontology editors for a specific language. This group includes: the Ontolingua

---

\(^{15}\) [Levesque, 1984] defines a knowledge representation system as a server supporting two kinds of operations: telling the system additional information and asking queries. He also states that, unlike other systems, KR systems use not only explicitly told but also inferred information, in answering questions.
Server [Farquhar et al., 1997], which supports ontology construction with Ontolingua and KIF; OntoSaurus [Swartout et al., 1997] with LOOM; WebOnto [Domingue, 1998] with OCML; and OilEd [Bechhofer et al., 2001a] with OIL first, later with DAML+OIL, and now with OWL.

- Integrated tool suites whose main characteristic is that they have an extensible architecture, and whose knowledge model is usually independent of an ontology language. These tools provide a core set of ontology related services and are easily extended with other modules to provide more functions. This group includes Protégé-2000 [Noy et al., 2000], WebODE [Arpírez et al., 2003], OntoEdit [Sure et al., 2002], and KAON [Maedche et al., 2003].

We are going to present how this evolution took place and to highlight the most relevant features of the tools of each group.

As commented above, the main characteristic of the first group of tools is that they have a strong relationship with a specific ontology language. These tools were created to allow editing and browsing ontologies in their corresponding languages and to import and export ontologies from/to some other ontology languages. These tools require that users have knowledge of their underlying ontology language.

The Ontolingua Server [Farquhar et al., 1997] was the first ontology tool created. It appeared in the mid-1990s, and was built to ease the development of Ontolingua ontologies with a form-based Web interface. Initially the main application inside the Ontolingua Server was the ontology editor. Then other systems were included in the environment, such as a Webster, an equation solver, an OKBC server, the ontology merge tool Chimaera, etc.

In parallel to the development of the Ontolingua Server, OntoSaurus [Swartout et al., 1997] was implemented as a Web editor and browser for LOOM ontologies. OntoSaurus consists of two main modules: an ontology server, which uses the knowledge representation system attached to the LOOM language, and a Web browser, which allows editing and browsing LOOM ontologies with HTML forms.

In 1997, WebOnto [Domingue, 1998] was released. WebOnto is an ontology editor for OCML ontologies. The main difference with respect to the previous tools lay in that the ontology editor was not based on HTML forms, but on Java applets, but its great advantage over the rest of ontology development tools was its strong support for collaborative ontology edition, which

---

16 In each group, a chronological order of appearance has been followed. Note that sometimes the most important bibliographic reference of a tool may have been written several years later than its first release.
allowed synchronous and asynchronous discussions about the ontologies being built by groups of users.

**OilEd** [Bechhofer et al., 2001a] was developed in 2001 as an ontology editor for OIL ontologies. With the creation of DAML+OIL, OilEd was adapted to manage DAML+OIL ontologies, and later it was adapted to manage OWL. Users of OilEd should know how to model ontologies using a DL approach. OilEd provides consistency checking functions and automatic concept classifications by means of the FaCT inference engine [Horrocks et al., 1999], though other DL inference engines such as RACER [Haarslev and Møller, 2001] can also be used.

In recent years a new generation of ontology-engineering environments has been developed. The design rationale for these environments is much more ambitious than that of previous tools. They are built as integrated tool suites that provide technological support to a wide variety of activities of the ontology development process. For this purpose, they have extensible, component-based architectures, where new modules can easily be added to provide more functions to the suite. Besides, the knowledge models underlying these environments are usually language independent and provide translations from and to several languages and formats.

**Protégé-2000** [Noy et al., 2000] is an open source, standalone application with an extensible architecture. The core of Protégé-2000 is its ontology editor, which can be extended with plug-ins that add more functions to the environment, such as ontology language import and export (FLogic, Jess, OIL, XML, Prolog), OKBC access, constraints creation and execution (PAL), ontology merge (PROMPT), etc. Recently, a new plug-in has been created to develop OWL ontologies. However, this plug-in is not compatible with many of other existing plug-ins, since it relies heavily on a specific extension of the tool's knowledge model to deal with ontologies implemented in that language.

**WebODE** [Arpírez et al., 2003] is also an extensible ontology-engineering suite based on an application server, whose development started in 1999. The core of WebODE is its ontology access service, which is used by all the services and applications plugged into the server. The WebODE's Ontology Editor, which allows editing and browsing WebODE ontologies, is based on HTML forms and Java applets. The workbench integrates services for ontology language import and export (XML, RDF(S), OIL, DAML+OIL, OWL, CARIN [Levy and Rousset, 1998], FLogic, Jess, Prolog), for axiom edition with WAB (WebODE Axiom Builder), for documentation, for evaluation, for merge, and an inference engine. WebODE can also interoperate with Protégé-2000. The examples shown throughout this thesis belong to the ontology translation systems developed for WebODE, especially those for the languages RDF(S) and OWL, and for the ontology tool Protégé-2000.
OntoEdit [Sure et al., 2002] is an extensible and flexible environment based on a plug-in architecture. Its ontology editor is a standalone application that allows editing and browsing ontologies, and includes functions for collaborative ontology building, inferencing, handling of domain lexicons, etc. This editor exports and imports ontologies in different formats (XML, FLogic, RDF(S), and DAML+OIL). There are two versions of OntoEdit: OntoEdit Free and OntoEdit Professional, each with a different set of functions.

The KAON tool suite [Maedche et al., 2003] is an open source extensible ontology engineering environment. The core of this tool suite is the ontology API, which defines its underlying knowledge model based on an extension of RDF(S). The OI-modeller is the ontology editor of the tool suite that provides capabilities for ontology evolution, ontology mapping, ontology generation from databases, etc.

2.4.2 Comparison of ontology tools

Section 2.3.2 described the formal, semi-formal and informal approaches used to compare the expressiveness of ontology languages, which can help in the task of developing ontology translation systems between languages. Clearly, these approaches can be also used to compare the knowledge models of ontology tools, so as to help in the provision of interoperability solutions between them, and also between tools and languages, and vice versa.

Table 2.3 shows the underlying KR formalism in which each tool is based, together with their underlying knowledge representation and formal axiom languages. These features determine most of the components to be used when building ontologies with them.

The table shows that the knowledge models of most of the tools are based on a combination of frames and first order logic (FOL). Only OntoSaurus and OilEd are based on description logic. The table also shows that OilEd, the Ontolingua Server, OntoSaurus and WebOnto are based on existing ontology languages, while the other tools are language-independent (except for formal axioms in OntoEdit, which are expressed in FLogic).

Table 2.3: Ontology tools’ KR paradigm and underlying KR languages (updated from [Gómez-Pérez et al., 2003]).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Underlying KR language</td>
<td>Frames</td>
<td>DL</td>
<td>Frames + FOL</td>
<td>Frames + FOL</td>
<td>DL</td>
<td>Frames + FOL</td>
<td>Frames + FOL</td>
<td>OCML</td>
</tr>
<tr>
<td>Formal axiom language</td>
<td>OWL</td>
<td>--</td>
<td>Ontolingua</td>
<td>LOOM</td>
<td>--</td>
<td>--</td>
<td>WAB</td>
<td>OCML</td>
</tr>
<tr>
<td></td>
<td>FLogic</td>
<td>KIF</td>
<td>LOOM</td>
<td>PAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2.4 shows the results of applying the semi-formal evaluation framework presented in section 2.3.2.2 to the ontology tools' knowledge models. As we can easily infer from their relationship with existing ontology languages, the results for OilEd, the Ontolingua Server, OntoSaurus, and WebOnto are almost equivalent to those obtained for OWL DL, Ontolingua, LOOM, and OCML, respectively. The only differences that appear are mainly related to those places where a workaround was proposed in table 2.2, since sometimes the ontology tool editors restrict the components that can be represented in their underlying language.

Finally, table 2.5 summarises the interoperability features available in the previous tools, that is, it summarises the tools and applications where the ontologies developed with each tool can be used and how the ontologies implemented in other languages or tools can be imported. The table shows first some of the tools with which each tool can interoperate. It contains knowledge portals such as KAON-Portal and ODESeW, for KAON and WebODE respectively; ontology-based annotators such as ODEAnnotate, OntoAnnotate, and MnM; Semantic Web Services building tools, such as ODESWS; UML modelling tools, such as ArgoUML; ontology merge...
tools, such as Chimaera and PROMPT; etc. In some of these tools OKBC is also included because any other applications could access their ontologies by means of this protocol.

The table also shows the lists of import and export formats of each tool, which provide a good overview of their interoperability choices. The lists contain ontology languages (Ontolingua, LOOM, RDF(S), DAML+OIL, OWL, etc.), general purpose languages (Prolog, Jess, etc.), and programming languages (C++, Java, etc.).

At first sight it may seem that two tools can interoperate if they share some import/export format. However, as we will describe in chapter 5, this is not necessarily true. Besides, much knowledge can be lost during the translation process. That is, there can be much knowledge not translated, and the resulting ontology in the target tool can be different from that of the original tool.

<table>
<thead>
<tr>
<th>With other tools</th>
<th>KAON-Portal</th>
<th>OilEd</th>
<th>OntoEdit</th>
<th>Ontolingua Server</th>
<th>OntoSaurus</th>
<th>Protégé-2000</th>
<th>WebODE</th>
<th>WebOnto</th>
</tr>
</thead>
<tbody>
<tr>
<td>KAON RDFS(S)</td>
<td>RDF(S)</td>
<td>OIL</td>
<td>DAML+OIL</td>
<td>OKBC</td>
<td>PROMPT</td>
<td>OKBC ArgoUML</td>
<td>ODE-KM</td>
<td>MnM</td>
</tr>
<tr>
<td>KAON RDF(S)</td>
<td>RDF(S)</td>
<td>OIL</td>
<td>DAML+OIL</td>
<td>KIF</td>
<td>XML RDF(S)</td>
<td>OWL OWL CARIN</td>
<td>OCML</td>
<td>RDFS(S)</td>
</tr>
<tr>
<td>KAON RDF(S)</td>
<td>RDF(S)</td>
<td>OIL</td>
<td>DAML+OIL</td>
<td>KIF</td>
<td>XML RDF(S)</td>
<td>OWL OWL CARIN</td>
<td>OCML</td>
<td>RDFS(S)</td>
</tr>
</tbody>
</table>

This section has focused on comparing the expressiveness of the ontology tools' knowledge models, and on their interoperability with other tools by means of common interchange languages. Clearly, there are more aspects of ontology tools that could be compared, such as their software architecture, the technology used to implement them, their ontology edition and browsing functions (including inference services attached, documentation options, logical expression editors, collaborative ontology construction, etc.), etc. However, these aspects are less related to the objective of building ontology translation systems, and consequently they have not been considered in this section. Such comparisons can be found in [Gómez-Pérez, 2002] and [Gómez-Pérez et al., 2003].
2.5 THE ONTOLOGY TRANSLATION PROBLEM

As shown in the previous two sections, there is a wide variety of languages and tools for building ontologies. The ontology translation problem [Gruber, 1993a] appears when an ontology developer decides to reuse an ontology (or part of an ontology) with a tool or language that is different from those ones in which the ontology is available. If ontology developers are forced, individually, to commit to the task of translating and incorporating the necessary ontologies to their systems, they will need both a lot of effort and a lot of time to achieve their objectives.

According to Gruber [Gruber, 1993b], the ontology translation problem would be ameliorated if ontology developers observed the minimal encoding bias design criterion when building their ontologies. This criterion states that “the conceptualisation should be specified at the knowledge level [Newell, 1982] without depending on a particular symbol-level encoding. The encoding bias of an axiomatisation, that is, representation choices that are made purely for the convenience of notation or implementation, should be minimised”.

However, this design criterion is not always observed by ontology developers, because they normally use the specific features of a language or tool when building ontologies. Therefore, ontology reuse will be highly boosted as long as automatic ontology translation services among those languages and/or tools are provided.

Till now, the work on ontology translation has mainly focused on developing translation systems for ontology tools, so as to import and export ontologies from and to various ontology languages. Only some work has been devoted to the direct translation between ontology languages, and there are not works on the direct exchange between ontology tools without common interchange languages.

2.5.1 Characterisation of ontology translation problems

Several classifications of ontology translation problems can be found in the literature ([Chalupsky, 2000], [Klein, 2001], [Euzenat, 2001]). It should be noticed that most of them are not specific to ontology translation per se, but related to more general objectives such as semantic interoperability, or ontology merge and integration, where ontology translation may be needed at some point.

[Chalupsky, 2000] defines several levels of ontology translation problems (also known as mismatches). There are problems strictly related to the differences between the source and target formats: syntax and expressivity. Other problems are related to how an ontology has been
modelled and to its ontological commitments: modelling conventions (which refers to the way in which an ontology has been modelled), model coverage and granularity (which refers to the domain coverage of the ontology), representation paradigm (which refers to the theories used for some specific parts of the domain, such as the representation of time, space, etc.), and inference system bias (which refers to how the ontology is constructed with respect to the inferences expected to be done with it).

[Klein, 2001] deals with translation problems from the ontology combination point of view. His classification is based on four levels: syntax, logical representation, semantics of primitives, and language expressivity. The first one corresponds to Chalupsky's syntax one. The other three correspond to Chalupsky's expressivity level, with slight differences between each other, which are explained in depth in [Klein, 2001]. All of these problems are tightly related to the ontology translation problem.

Finally, [Euzenat, 2001] also deals with translation problems in the context of semantic interoperability, distinguishing the following non-strict levels of language interoperability: encoding, lexical, syntactic, semantic, and semiotic. He also considers that the transformations can be set at these various levels, although in the approach that he presents (families of ontology languages, described in the next section) he only considers for ontology translation those that appear in the semantic level.

Figure 2.6 shows the relationships between all these layers, and with respect to the syntax, semantic and pragmatic layers defined in the area of semiotics [Morris, 1938], described in section 2.3.2.1. As we can see in the figure, the classification proposed by Euzenat is the most exhaustive one, since it considers the problems that may appear in the pragmatic layer.

![Figure 2.6: Relationships among different characterisations of ontology translation problems, and with the syntax, semantic and pragmatic layers of the theory of signs.](image-url)
2.5.2 Translation approaches and architectures

The following three translation approaches or architectures have been distinguished in the literature (based on [Euzenat and Stuckenschmidt, 2003] and [Stuckenschmidt, 2003]):

- **Peer-to-peer translations** (aka *mapping approach*). This is the most common approach to ontology translation. Translation systems rely on ad-hoc solutions for translating between two different formats, mapping certain types of expressions in the source format to corresponding expressions in the target format. Most of the import and export services of ontology development tools can be included in this group, such as those of the Ontolingua server, OntoSaurus, OntoEdit, OilEd, Protégé-2000, WebODE, the DAML+OIL to OWL converter\(^{18}\), etc.

- **Common interchange language** (aka *pivot approach*). This approach consists in using a single pivot language to and from which all other formats are translated. To preserve semantics in the translation process, the pivot language has to be expressive enough to allow representing the knowledge that can be expressed in the source and target formats. However, if the pivot language is too expressive, it normally allows expressing the same knowledge in different ways, which makes its use difficult for achieving interoperability between any pair of source and target formats.

An example of such a pivot language is KIF. Besides, due to the ontology language standardisation process promoted by the W3C (the World Wide Web Consortium), RDF(S) and OWL can be also considered as common ontology interchange languages. Chapter 5 describes the main advantages and drawbacks of this approach, analysing how KIF, RDF(S) and OWL have been used as common interchange languages between languages and tools. Besides, it should be noticed that the translations to and from the pivot language are created using the same kind of mappings between expressions described previously.

- **Families of ontology languages** [Euzenat and Stuckenschmidt, 2002]. This is a more formal and well-documented approach to the ontology translation problem. It considers having a predefined set of languages that are structured by a partial order relation of language coverage. The transformations to be performed between them are studied in advance, and it is possible to know which are the losses of knowledge between them. To perform a transformation, one of the languages in this set must be selected as the input language, and

---

17 Euzenat uses the term semiotic as a synonym of pragmatic. In linguistics, these two terms are not synonyms, since semiotics is composed of three levels: syntax, semantics and pragmatics [Morris, 1938].

another one must be selected as output language, and several transformations are composed in order to transform from the source to the target format. The main drawback of this approach is that it is heavily based on description logic and it is not clear how it can be used for transformations between formats that are not based on this KR formalism. Besides, it does not cover all the interoperability levels proposed in [Euzenat, 2001], but only translation problems related to the preservation of the ontology semantics in the translation.

[Euzenat and Stuckenschmidt, 2003] propose an additional architecture, based on languages composed of several layers with different expressiveness and reasoning capabilities, such as OIL and OWL. However, this approach can be considered as a specific part of the pivot one and hence it is not included in the previous list.

Figure 2.7 summarises how these three architectures would allow performing translations among \( n \) formats, together with the approximate number of ontology translation systems that would be needed in each case.

![Figure 2.7: Comparison of the three architectures for ontology translation between \( n \) formats](adapted from [Uschold and Grünlinger, 1996]).

If a unique translation system must be designed for every two party exchange, \( O(n^2) \) translators would be required to achieve interoperability among \( n \) different KR systems. If a common interchange format is used instead, the number of translators needed is reduced to \( O(2n) \). In the case of a family of ontology languages, the number of translation systems required depends on the number of levels of the structure formed by the language coverage relation. If this structure has \( k \) levels, with approximately \( n/k \) languages at each level, the number of translation systems required is \( O(2n^2/k) \).
2.5.3 Technology for building ontology translation systems

Ontology translation systems are usually created using general-purpose programming languages, such as LISP or Java, and the mappings between expressions in the source and target languages are neither well-documented or explained.

There are only two specific tools that allow creating ontology translation systems, Transmorpher and OntoMorph:

- **Transmorpher**[^19] [Euzenat and Tardif, 2001] is a tool that facilitates the definition and processing of complex transformations of XML documents. Among other domains, this tool has been used in the context of ontologies, using a set of XSLT documents that are able to transform from one DL language to another, expressed in DLML[^20]. This tool is aimed at supporting the "family of ontology languages" approach.

- **OntoMorph** [Chalupsky, 2000] is a tool that allows creating translators declaratively. Transformations between the source and the target formats are specified by means of pattern-based transformation rules, and are performed in two phases: syntactic rewriting and semantic rewriting. The last one needs the ontology or part of it translated into PowerLOOM, so that this KR system can be used for certain kinds of reasoning, such as discovering whether a class is subclass of another, whether a relation can be applied to a concept or not, etc. This tool can be used in the context of any of the transformation architectures presented in the previous section.

2.6 Conclusions

As shown in the sections 2.1 to 2.4, not all the ontology languages and tools are based on the same KR paradigm, nor do they represent knowledge with the same ontology components. Ontology translation is a difficult task, since it does not only consist in transforming between two syntactically different formats, but also does it deal with higher-level transformations to ensure semantic and pragmatic preservation in the target format, that is, preservation of the knowledge represented in the source format, and of its intended meaning. Therefore, it is important to give an adequate methodological and technological support for the creation and maintenance of ontology translation systems, so that they can be built and maintained more easily and so that they ensure higher quality translations.

[^19]: http://transmorpher.inrialpes.fr/
Section 2.5 has shown several complementary characterisations of ontology translation problems, structured at different levels. Currently there is no consensus on the exact levels to be used for classifying these problems, although all of them cover syntax and semantic problems as the most important ones to be overcome during ontology translation.

Section 2.5 has also described three different translation approaches that have been proposed in the literature. These approaches have been usually compared to each other taking into account the number of ontology translation systems needed for performing transformations between \(n\) formats. In this sense, the common interchange format approach is the best option, since it reduces significantly the number of translation systems needed. However, except for the “family of ontology languages” approach, there are no deep studies about how semantics and pragmatics are preserved in these transformations, and the approaches have not been compared to each other taking into account such knowledge preservation features.

Finally, only two systems have been built specifically for developing ontology translation systems: Transmorpher and OntoMorph. They allow specifying translation decisions declaratively, which helps in the task of developing and maintaining the translation system. However, they do not cover all the levels of translation problems that may appear in a transformation, nor are they related to any method for building ontology translation systems.

The analysis of the current state of the art on ontology translation shows that there is a need for a set of integrated methods and techniques, together with their supporting technology, to help in the complex task of developing and maintaining ontology translation systems. Most of the translation systems currently available have been developed ad hoc, the translation decisions that they implement are usually difficult to understand and hidden in the source code of the systems, and, besides, it is not clear nor documented how much knowledge is lost in the transformations that they perform.

This thesis will aim at providing an advance on the current state of the art on ontology translation by proposing an ontology translation model that takes into account that translation decisions must be specified declaratively and at all the different translation layers identified, and which includes a method for ontology translation system development and maintenance. The thesis will also explore in depth different translation approaches between two formats, with special focus on how they preserve the semantics and pragmatics of the original ontology, especially in cyclic transformation processes.
Chapter 3
Work Objectives

This chapter presents the goals of this thesis, together with the open research problems that we aim to solve. Besides, we detail the contributions to the current state of the art, and the work hypothesis, the assumptions considered as a starting point for this work and the restrictions of the results presented.

3.1 GOALS AND OPEN RESEARCH PROBLEMS

The goal of this thesis is to advance the current state of the art in ontology translation by providing a model to build ontology translation systems declaratively, taking into account the following layers of translation problems: lexical, syntax, semantic, and pragmatic; and by proposing solutions that ensure semantic and pragmatic preservation in the translation processes, especially for the case of cyclic transformations.

In order to achieve the first objective, the following (non exhaustive) list of open research problems must be solved:

- There is no consensus about the classification of ontology translation problems. Several characterisations have been proposed, under different perspectives (ontology translation, ontology merge and integration, semantic interoperability, etc.), where the same problem can be classified under different groups.

- From a methodological perspective, there is a lack of integrated methods and techniques that support the complex task of building ontology translation systems. Existing systems are usually built without taking into account that translation decisions can be taken at different layers. As a consequence, the criteria used on the translation are not clear and the translation systems are difficult to maintain, since the translation decisions are hidden inside their source code.
From a technological perspective, there are two open problems:

- There are formal approaches to compare the expressiveness of KR languages and tool knowledge models (such as [Baader, 1996] and [Borgida, 1996]). These approaches rely on the use of translation functions between the source and target formats. However, they do not specify how these translation functions should be built.

- Two systems give technological support to the task of building ontology translation systems, and allow specifying them declaratively: OntoMorph [Chalupsky, 2000] and Transmorpher [Euzenat and Tardif, 2001]. However, they impose important limits regarding the languages to which they can be applied: the former can only be applied to Lisp-based languages, and the latter can only be applied to XML-based languages.

With regard to the second objective (ensuring semantic and pragmatic preservation in translations), the following (non-exhaustive) list of open research problems must be solved:

- Some ontology building methodologies consider that it may be necessary to make transformations of ontologies between different formats (Methontology [Fernández-López et al., 1999] and On-To-Knowledge [Staab et al., 2001]). Other works describe the lifecycle of ontologies once they are developed ([Gómez-Pérez and Rojas, 1999]). However, neither of these works describe in detail how knowledge is preserved when it is being translated between different formats.

- There are not detailed studies about the different approaches used so far for ontology translation, and about their impact on the preservation of the semantics and pragmatics of the transformed ontologies, especially in the case of cyclic transformations.

3.2 CONTRIBUTIONS TO THE STATE OF THE ART

This thesis aims at giving solutions to the previous open research problems. Chapter 4 will describe the solutions proposed for the first objective (the creation of a layered declarative model for building and maintaining ontology translation systems) and chapter 5 will describe those related with the second one (the preservation of the ontology semantics and pragmatics in translation processes).

With regard to the first objective, this thesis presents advances to the state of the art in the following aspects:
- **Refinement of Euzenat's classification of ontology translation problems** ([Euzenat, 2001]), based on the theory of signs (aka semiotics). The classification proposed takes into account four different layers of translation problems: lexical, syntax, semantic, and pragmatic. It merges Euzenat's encoding and lexical layers into a single one: the lexical layer. It also considers pragmatics (the use and interpretation of the knowledge transformed) instead of semiotics (which covers, in the theory of signs, the syntax, semantic and pragmatic levels).

- **Integrated method for building ontology translation systems**, which takes into account the ontology translation layers previously identified and proposes to implement declaratively the ontology translation decisions for each layer. This method identifies the main activities and tasks to be performed, the participants of each task, and the techniques to be used in each task.

- **A language (ODEDialect) composed of three declarative languages** (ODELex, ODESyntax, and ODESem), which provide means to express and build declaratively the translation functions defined for the formal comparison approaches of KR languages and tool knowledge models. These languages improve the support provided by other tools used for building ontology translation systems as follows:

  - With regard to OntoMorph: (1) ODEDialect uses four layers to implement ontology translation decisions, while OntoMorph only uses two (syntax and semantic rewriting); (2) ODEDialect does not impose the restriction of having to know an additional KR language (PowerLoom) so as to translate ontologies intermediately to it in order to perform complex transformations; (3) ODEDialect deals with languages and tools' APIs instead of dealing directly with the ontology source code (which must be in Lisp-like syntax in OntoMorph); and (4) ODEDialect uses Java instead of Lisp, the former language being more widely used for ontology technology than the latter.

  - With regard to Transmorpher: (1) ODEDialect deals with different layers of ontology translation problems separately, rather than using a single XSLT file to implement all of them; and (2) ODEDialect does not impose the restriction that the ontologies must be transformed to and from XML syntax.

The second objective of this thesis deals with the preservation of the ontology semantics and pragmatics in translation processes. This thesis presents advances to the current state of the art in the following aspects:
- Description of the knowledge lifecycle during translation processes, paying special attention to cyclic transformations. As stated in the previous section, there are not detailed studies about the knowledge preservation when ontologies are translated back and forth between different formats.

- Empirical analysis of the ontology translation approaches used for exchanging ontologies between ontology tools. The thesis pays special attention to how each of these approaches preserves knowledge during the translation process, by using auxiliary ontology components, if necessary. It also focuses on the pragmatic preservation during the translation process, analysing how the ontology legibility is guaranteed in the target format and how the initial ontology components are recovered in cyclic transformations. This analysis takes into account ideas about the notational efficacy and conciseness of presentation of KR systems [Woods, 1983], and about the generation of auxiliary ontology components [Baader, 1996].

The layered declarative method for ontology translation proposed in this thesis gives support to cyclic transformations with semantic and pragmatic preservation.

All these contributions are backed up by a large number of experiments. These experiments show how the methodological and technological solutions proposed have been applied to real-world problems, tackled in the context of the following R&D projects: the EU funded MKBEEM\(^1\) and Esperonto\(^2\) projects, and the Spanish CICYT funded ContentWeb project (TIC-2001-2745).

### 3.3 Work Assumptions, Hypothesis, and Restrictions

The work performed in this thesis is based on the set of assumptions listed below. These assumptions help explain the decisions taken for the development of the methodological and technological solutions and the relevance of the contributions presented. Assumptions A1 to A3 are related to the first objective of the thesis, whilst the assumption A4 deals with the second one.

A1. The boundaries between different layers of ontology translation problems are not strict, that is, some translation problems may be related to several layers and the decisions taken at one layer may influence the decisions taken at the other ones.

---


A2. Ontology translation systems are complex software systems: they are difficult to create and maintain, and the translation decisions taken are normally difficult to understand. They are easier to create, maintain, and reuse if they are implemented taking into account different layers of ontology translation problem and using a declarative approach.

A3. An ontology translation system is a software system. Therefore, any software engineering methodology can be applied to its construction. However, specific activities and tasks can be inserted in the software engineering methodology to help building these systems more efficiently and with higher quality.

A4. An ontology translation system does not necessarily have to preserve the semantics and the pragmatics of the ontology that it transforms. These preservation properties, which are normally desirable features, depend on the foreseen application of the ontologies transformed with such a system.

Once that the assumptions have been identified and presented, the set of hypothesis of this thesis are described. This set of hypothesis covers the main features of the proposed solutions:

H1. The layered declarative ontology translation model proposed in this thesis combines existing translation proposals and techniques (used for describing languages and tools, for comparing them, for deciding the feasibility of the construction of an ontology translation system, etc.), and eases the task of expressing translation decisions between two formats.

H2. The method proposed as part of the ontology translation model gives an integrated support to the four ontology translation layers identified. The method proposes to use specific techniques to solve the problems of each layer, and to use declarative languages to implement the transformations at each layer.

H3. The method proposed distinguishes several user roles, in charge of different tasks. From these roles, the most important one is the knowledge engineer, who is in charge of analysing, designing and implementing the transformations between the source and target formats. This person must have a deep knowledge of the source and target formats' knowledge models, and of their ontology management APIs.

H4. The method and technology proposed in this thesis are independent of the ontology translation architecture where they are used. Hence, they can be applied to construct translation systems in pivot approaches, peer-to-peer translations, and in the "family of ontology languages" approach.
The method and technology proposed in this thesis can be applied to construct ontology translation systems independently of the requirements imposed with regard to semantic and pragmatic preservation.

Finally, the following set of restrictions defines the limits of the contributions of this thesis and allows determining future research objectives. Most of these restrictions are related to the technological aspects of the contribution (R1 to R6), although R7 deals with the study about semantic and pragmatic preservation and R8, R9, and R10 are related to the experimentation.

R1. The layers and the method proposed in this thesis are only proposed for the task of ontology translation. They do not aim at solving the translation or exchange problem of other types of knowledge, such as knowledge about planning, knowledge about machine learning, etc.

R2. The method and technology proposed in this thesis do not consider the optimisation of the translation process of the system generated, neither in terms of the space required during translation nor in terms of the time needed to complete the translation.

R3. The ontology translation systems generated from ODEDialect specifications take as an input an ontology in the source format and output the ontology in the target format, according to the translation decisions implemented.

R4. The ODEDialect language cannot be applied to create ontology translation systems to and from languages or tools that do not provide an ontology management API in Java, unless such an API is generated prior to the development of the ontology translation system.

R5. Ontology translation systems developed with the ODEDialect language do not make incremental translations, nor are they connected to ontology version control management methods or tools.

R6. The three declarative languages that compose ODEDialect (ODELex, ODESyntax, and ODESem) are not expressive enough to represent every possible type of transformation that may appear in any transformation. They only cover the most common types of transformations identified at each layer. If a more complex transformation must be performed, user-defined functions must be used.

R7. The description of semantic and pragmatic preservation during translation processes is restricted to the “representational aspects” of the origin and target formats [Baader, 1996]. It does not cover other aspects such as the reasoning tractability of the knowledge generated in the target format.
R8. The model proposed in this thesis has been derived from the experimentation on building WebODE import and export services from and to different types of ontology and other general-purpose languages, such as DAML+OIL, OIL, XCARIN, FLogic, Java, and XML.

R9. Once created, the evaluation of the ontology translation model proposed in this thesis is restricted to the ontology markup languages RDF(S) and OWL, and to the ontology tools Protégé-2000 and WebODE. They have been selected because RDF(S) and OWL are the current W3C recommendations for the Semantic Web, Protégé-2000 is the most widely-used ontology tool, and WebODE has been developed inside the Ontological Engineering Group to which the author belongs.

R10. The results shown in the experimentation chapter are based on the use of the following versions of the ontology tools and languages: Protégé-2000 version 2.0, WebODE version 2.0, and RDF(S) and OWL recommendations of February 2004.
Chapter 4
A Layered Declarative Ontology Translation Model

This chapter presents our model for ontology translation, based on our experience in the construction of ontology translation systems between languages and/or tools. Our model is characterised by two main features: on the one hand, translation decisions are defined at four different layers, which are tightly related to each other (lexical, syntax, semantic, and pragmatic); on the other hand, translation decisions are mainly implemented in a declarative fashion. Our assumption is that both aspects make ontology translation systems easier to build, maintain, and reuse. As part of this model, we propose a method that guides in the process of developing an ontology translation system between two different ontology languages and/or tools. In this method we identify four activities (feasibility study, analysis of source and target formats, design of the translation system, and implementation of the translation system), which are in its turn divided into several tasks. For each task, we specify its objectives, its inputs and outputs, a set of proposed techniques that can be used to perform it, and its main responsible.

4.1 INTRODUCTION

This section describes our model for developing ontology translation systems between two formats, which can be either ontology languages or ontology tools. As expressed by the chapter title, there are two parameters that fully characterize this model:

- The model is \textit{layered}, which means that translation decisions are structured in different levels \cite{Corcho2002}: lexical, syntactic, semantic, and pragmatic. These layers are related to different categories of ontology translation problems. We assume that it is easier to create and maintain ontology translation systems where translation decisions are clearly defined in different layers, because they are easier to understand, and that it is also easier to automate the ontology translation process. Besides, the problems to be solved in each layer are different (it is not the same to transform a concept identifier from a format to another,
e.g. by replacing its blank spaces with underscores, than to transform a set of logical restrictions into a disjoint decomposition of a concept). Therefore we will require using different translation techniques for each layer.

The model is mainly declarative, which basically means that most of the translation decisions are not “hidden” inside the source code of the ontology translation system. As we will describe later in this chapter, we will use different types of tables and diagrams to formalise the translation decisions to be made at each layer, and we will use declarative languages to implement them (ODELex, ODESyntax, and ODESem). For instance, in the semantic layer, for each type of ontology component of the source format (e.g., concepts, relations, etc.), we will express declaratively the transformations to be done to get the same knowledge in the target format. We will only use general-purpose programming languages (Java) for complex transformations that cannot be defined declaratively.

Sections 4.2 and 4.3 describe in detail both dimensions and show examples of how transformations are expressed at each layer. It is important to remark that this model does not aim to solve the ontology translation problem per se, since its objective is not to create a generic ontology translation system for every pair of source and target formats used for representing ontologies. Instead, its aim is to improve the state-of-the-art in ontology translation by providing a structured and declarative form of expressing transformations between different formats.

4.2 ONTOLOGY TRANSLATION LAYERS

As we commented above, our model proposes to structure translation decisions in four different layers. Our selection of layers is based on the classifications of ontology translation mismatches and interoperability levels for semantic integration shown in section 2.5, especially on Euzenat’s one [Euzenat, 2001]. We will now describe the types of translation problems that can be usually found in each of these layers and will show some examples of common transformations performed in each of them.

4.2.1 Lexical layer

The lexical layer deals with the “ability to segment the representation in characters and words (or symbols)” [Euzenat, 2001]. That is, different languages and tools normally use different character sets and different grammars for generating their terminal symbols (ontology component identifiers, natural language descriptions of ontology components, and attribute values). This translation layer deals with the problems that may arise in these symbol transformations.
Therefore, in this layer we deal with transformations of ontology component identifiers (e.g., a specific character cannot be used in one of the formats), of pieces of text used for natural language documentation purposes (e.g., a specific character in the natural language documentation of a component must be escaped since the target format does not allow it as part of the documentation), and of values (e.g., numbers must be represented as character strings in the target format).

From a lexical point of view, we can distinguish three groups of formats among the most representative ontology languages and tools:

- **ASCII-based formats.** Among these formats we can cite the following classical languages: KIF, Ontolingua, CycL, LOOM, OCML, and FLogic. We can also include in this group the ontology tools related to some of these languages (the Ontolingua Server, OntoSaurus, and WebOnto). These languages are based on ASCII encodings, and hence the range of characters allowed for creating ontology component identifiers, and for representing natural language texts and values is restricted to most of the characters allowed in this encoding.

- **UNICODE-based formats.** Among these formats we can cite the following ontology tools: OntoEdit, Protégé-2000, and WebODE. These formats are based on the UNICODE encoding, which is an extension of the ASCII encoding, and hence allows using more varied characters (including Asian and Arabic characters, more punctuation signs, etc.).

- **UNICODE&XML-based formats.** Among these formats we can cite the ontology markup languages: SHOE, XOL, RDF, RDFS, OIL, DAML+OIL, and OWL, and some of the tools that are related to them, such as KAON and OilEd. These formats are characterised not only for being UNICODE compliant, as the previous ones, but also for restricting the use of some characters and groups of characters in the component identifiers and in the natural language documentation and values, such as the use of tag-style pieces of text (e.g., `<example>`) inside documentation tags. An important restriction is the compulsory use of qualified names (QNames) as the identifiers of ontology concepts and properties, since they are used to construct tags when dealing with instances.

The easiest lexical transformations are usually those to be done from the first and third group of formats to the second ones, which is the most unrestricted one. In the other cases, the specific features of each format do not allow us generalising the types of transformations to be done, although they mainly consist in replacing non-allowed characters by other allowed ones, or replacing identifiers that are reserved keywords in a format to other identifiers that are not reserved keywords. Obviously, there are also differences among the languages and tools inside each group, although the transformations needed in those cases are minimal.
Other sources of problems in lexical transformations are related to the scope of the ontology component identifiers in the source and target formats, and the restrictions related to overlapping identifiers. These problems appear when, in the source format, a component is defined inside the scope of another component, and hence its identifier is local to the latter, and in the target format the correspondent component has a global scope. As a consequence, there could be clashes of identifiers in case that in the source format two components have the same identifier.

Table 4.1 shows some examples of how some ontology component identifiers can be transformed from WebODE to Ontolingua, RDF(S), OWL and Protégé-2000, taking into account the rules for identifier generation of each format and constraints about the scope of some ontology component identifiers in the source and target languages, and about the possibility of overlapping with other component identifiers.

<table>
<thead>
<tr>
<th>WebODE identifier</th>
<th>Target</th>
<th>Result</th>
<th>Reasons for transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Trip</td>
<td>Ontolingua</td>
<td>Business-Trip</td>
<td>Blank spaces in identifiers are not allowed in Ontolingua</td>
</tr>
<tr>
<td>1StarHotel</td>
<td>RDF(S)</td>
<td>OneStarHotel</td>
<td>Identifiers cannot start with a digit in RDF(S). They do not form valid QNames</td>
</tr>
<tr>
<td>Concepts Name and name</td>
<td>Ontolingua</td>
<td>classes Name and Name_1</td>
<td>Ontolingua is not case sensitive</td>
</tr>
<tr>
<td>Concept Room</td>
<td>OWL</td>
<td>classes Room, Flight datatypeProperty roomFare datatypeProperty flightFare</td>
<td>WebODE attributes are local to concepts. OWL datatype properties are not defined in the scope of OWL classes, but globally</td>
</tr>
<tr>
<td>Concept Flight</td>
<td>Protégé-2000</td>
<td>class Name; slot name</td>
<td>The identifiers of classes and slots cannot overlap in Protégé-2000</td>
</tr>
</tbody>
</table>

Inside this layer we also deal with the different naming conventions that exist in different formats. For instance, in Lisp-based languages and tools, such as Ontolingua, LOOM, OCML, and their corresponding ontology tools, compound names are usually joined together using hyphens: e.g. Travel-Agency. In tools like OntoEdit, Protégé and WebODE, words are separated with blank spaces: e.g. Travel Agency. In ontology markup languages, the convention used for class identifiers is to write all the words together, with no blank spaces nor hyphens, and with the first capital letter for each word: e.g. TravelAgency.

---

1 These types of problems may be also related to the pragmatic layer, as we will describe later in this section. We will also see that the limits of each translation layer are not strict; hence we can find transformation problems that are in the middle of several layers.
4.2.2 Syntactic layer

This layer deals with the "ability to structure the representation in structured sentences, formulas or assertions" [Euzenat, 2001]. Ontology components in each language or tool are defined with different grammars. Hence, this translation layer deals with the problems related to how the symbols are structured in the source and target formats, taking into account the derivation rules for ontology components in each of them.

The following types of transformations are included in this layer: transformations of ontology component definitions according to the grammars of the source and target formats (e.g., the grammar to define a concept in Ontolingua is different than that of OCML) and transformations of datatypes (e.g., the datatype date in WebODE must be transformed to the datatype &xsd;date in OWL). Figure 4.1 shows an example of how a WebODE concept definition (expressed in its XML format) is transformed to Ontolingua and OWL.

![WebODE's XML syntax](image)

**Ontolingua**

```
<Concept>
  <Name|rflID="flight">
    <Documentation> A trip made by or in an airplane or spacecraft. </Documentation>
  </Name>
  <Instance-Attribute>
    <Name>airCompany</Name>
    <Description>The air company(s) responsible for the flight</Description>
    <Type>String</Type>
    <Minimum-Cardinality>1</Minimum-Cardinality>
    <Maximum-Cardinality>-1</Maximum-Cardinality>
  </Instance-Attribute>
</Concept>
```

**OWL**

```
<owl:Class rdfID="flight">
  <rdfs:comment>A trip made by or in an airplane or spacecraft.</rdfs:comment>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="airCompany"/>
      <owl:allValuesFrom rdf:resource="~airCompany"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 4.1. Examples of transformations at the syntactic layer.

There are other aspects that can be taken into account in this layer, such as the fact that some ontology languages and tools allow defining the same component with different syntaxes. For example, Ontolingua provides at least four different ways to define concepts: using KIF, the Frame Ontology or the OKBC-Ontology exclusively, or embedding KIF expressions inside definitions that use the Frame Ontology [Corcho and Gómez-Pérez, 1999]. This adds
complexity both for the generation of such a format (we must decide which kind of expression to use\(^2\)) and for its processing (we have to take into account all the possible syntactic variants for the same piece of knowledge).

From a syntactic point of view, we can distinguish the following (overlapping) groups of formats among the most representative ontology languages and tools:

- **Lisp-based formats.** In section 2.3 we showed that the syntax of several classical ontology languages are based on the Lisp language. Among them we can cite: KIF and Ontolingua, LOOM, and OCML, together with their corresponding ontology tools (the Ontolingua Server, OntoSaurus, and WebOnto, respectively). The Lisp language was widely used for building knowledge representation systems, especially during the 1970s, the 1980s and the 1990s.

- **XML-based formats.** As shown also in section 2.3, ontology markup languages are characterised by being represented in XML syntax. Among them we can cite: SHOE, XOL, RDF, RDFS, OIL, DAML+OIL, and OWL. Besides, some ontology tools also provide ad hoc XML backends that can be used to implement their ontologies, such as OntoEdit, Protégé-2000, and WebODE.

- **Ad hoc text formats.** There are other ontology languages that do not provide any of the previous syntaxes but their own ad hoc formats. Among these languages we can cite F-Logic, the ASCII syntax of OIL, and the Notation-3 (N3) syntax used to represent ontologies in RDF, RDFS, and OWL. Except for F-Logic, these syntaxes are alternative, and are mainly intended for human-consumption.

- **Ontology management APIs.** Finally, several ontology languages and tools provide ontology management APIs. These APIs are included in the characterisation because they can be considered as another form of syntax: the expressions used to access, create, and modify ontology components in the programming language in which these APIs are available have to be created according to the specification provided by the API. Among the languages with an ontology management API we can cite: all the ontology markup languages, where ontologies can be created using available XML Java APIs such as DOM, SAX, etc.; and, more specifically, RDF, RDFS, DAML+OIL, and OWL, for which there are specific APIs that resemble the knowledge models of the ontology languages, such as Jena, the OWL API, etc. Among the tools we can cite: KAON, OntoEdit, Protégé-2000, and WebODE.

---

\(^2\) As with naming conventions, this decision will be also related to the pragmatic translation layer.
Inside this layer we must also take into account how the different formats represent datatypes. We can distinguish two groups taking into account this feature:

- **Formats with their own internal datatypes.** Among these formats we can cite all the ontology languages described in section 2.3 except for RDF, RDFS, and OWL, and all the ontology tools described in section 2.4.

- **Formats with XML Schema datatypes.** These datatypes have been defined with the aim of providing datatype standardisation in Web contexts (e.g., in Web services). They can be used in the ontology languages RDF, RDFS, and OWL, and in the ontology tool WebODE, which allows using both types of datatypes (internal and XML Schema ones).

Therefore, with regard to datatypes, the problems to be solved will mainly consist in finding the relationships between the internal datatypes of the source and target formats (not all the formats have the same group of datatypes) or finding relationships between the internal datatypes of a format and the XML Schema datatypes, or vice versa.

### 4.2.3 Semantic layer

This layer deals with the "ability to construct the propositional meaning of the representation" [Euzenat, 2001]. Different ontology languages and tools can be based on different KR paradigms (frames, semantic networks, first order logic, conceptual graphs, etc.) or on combinations of them, as we described in section 2.2. These KR paradigms do not always allow expressing the same type of knowledge. Besides, the languages and tools based on these KR paradigms sometimes allow expressing the same knowledge in very different ways.

Therefore, in this layer we deal not only with simple transformations (e.g., WebODE concepts are transformed into Ontolingua and OWL classes), but also with complex transformations of expressions that are usually related to the fact that the source and target formats are based on different KR paradigms (e.g., WebODE disjoint decompositions are transformed into subclass-of relationships and PAL³ constraints in Protégé-2000, WebODE instance attributes attached to a class are transformed into datatype properties in OWL and unnamed property restrictions for the class).

As an example, figure 4.2 shows graphically how to represent a concept partition in different ontology languages and tools. In WebODE and LOOM there are specific built-in primitives for representing partitions; in OWL, the partition must be represented by defining the `rdfs:subClassOf` relationship between each class in the partition and the parent class, by stating
that every possible pair of classes in the decomposition are disjoint to each other, and by defining the parent class as the union of all the classes in the partition; and in Protégé-2000, the partition is represented, like in OWL, with subclass-of relationships between all the classes in the partition and the parent class, with PAL constraints that represent disjointness between all the classes in the partition, and by stating that the parent class is abstract (that is, that it cannot have direct instances).

![Diagram showing examples of transformations at the semantic layer.](image)

Most of the work on ontology translation done so far has been devoted to solving the problems that arise in this layer. For example, the formal, semi-formal, and informal methods for comparing ontology languages and ontology tools' knowledge models, which were described in chapter 2, aim at helping to decide whether two formats have the same expressiveness or not, so that knowledge can be preserved in the transformation. And some of these approaches can be also used to decide whether the reasoning mechanisms present in both formats will allow inferring the same knowledge in the target format.

Basically, these studies allow analysing the expressiveness (and, in some cases, the reasoning mechanisms) of the source and target formats, so that we can know which types of components can be translated directly from a format to another, which types of components can be expressed

---

5 Protégé Axiom Language
using other types of components from the target format, which types of components cannot be expressed in the target format, and which types of components can be expressed, although losing part of the knowledge represented in the source format.

Therefore, the catalogue of problems that can be found in this layer are mainly related to the different KR formalisms in which the source and target formats are based. This does not mean that translating between two formats based on the same KR formalism is straightforward, since there might be differences in the types of ontology components that can be represented in each of them. This is specially important in the case of DL languages, since many different combinations of primitives can be used in each language, and hence many possibilities exist in the transformations between them, as shown in [Euzenat, 2002]. However, the most interesting results appear when the source and target KR formalisms are different.

We will present the most relevant problems that usually appear in the transformations between the following KR formalisms: frames combined with first order logic and description logics. With regard to the description logic formalism, there are many possibilities, depending on the primitives used for each specific language, as described in section 2.2. We will consider in the following sections that we are using a SHOIN(d+) languages (the language to which OWL DL is equivalent), since it is one of the most expressive DL languages for which there are decidable reasoners.

4.2.3.1 Semantic transformations from combinations of frames and first order logic to description logic

The following transformation problems usually appear at the semantic layer when the source format is a language or tool based on a combination of frames and first order logic, and the target format is a language or tool based on description logic. This list must not be considered exhaustive, since there are many other problems that appear at the semantic layer when transforming between formats based on both formalisms. We have instead tried to show the most relevant ones:

- Metaclasses, which can be usually expressed in frame-based formats, cannot be expressed in description logic, because of the reasoning problems that they provoke. Consequently, this knowledge is normally lost in the translation.

- In the frames KR paradigm, class definitions are normally partial (that is, classes are primitive, because their definition only expresses necessary conditions for instances that belong to the class). There are only two exceptions where the definition of a class is not partial but complete: (1) in some frame-based languages, such as Ontolingua and OCML, it
is possible to define necessary and sufficient conditions for instances to belong to the class (with the :iff-def keyword); (2) in other languages, and also in Ontolingua and OCML, formal axioms can be defined stating the sufficient conditions for instances to belong to the class (the necessary conditions are already provided in the class definition).

- Disjoint and exhaustive decompositions of classes, and class partitions, can be defined in many frame-based languages. However, most of the DL languages do not provide these primitives for building class taxonomies, but only the subclass-of relationship. Therefore, these decompositions and partitions must be transformed using the subclass-of relationships. Besides, the disjointness between classes can be expressed by stating that the class A is a subclass of the class not B. Finally, exhaustiveness in class taxonomies can be expressed with complete definitions using the union of all the classes in the decomposition or partition, that is, stating that $A = C_1 \cup C_2 \cup \ldots \cup C_n$.

- Own slots, which are used to specify properties of the classes themselves in the frame-based paradigm, cannot be usually expressed in DL. Consequently, this knowledge is normally lost in the transformation.

- Some frame-based systems define slots as second-class citizens of the ontology, that is, they are defined inside the scope of other ontology components (e.g., classes). On the contrary, in DL, roles are always global to ontologies. To transform frame-based slots into DL roles, the roles must be defined globally, they must be "attached" to classes by means of universal property restrictions; and their cardinality must be expressed by means of cardinality restrictions. There are several situations that may arise from the differences in the scope of slots/roles:
  
  - If the slot is defined in the scope of several classes, then its domain can be interpreted as the union of all those classes.
  - If the slot has several ranges, then the slot range is (usually) considered as the union of the classes in frame-based systems, while in DL it is considered as the intersection of the classes.

- Not all the axioms that can be created in FOL can be expressed in DL. For instance, variables that appear in FOL expressions cannot be represented in DL, although some workarounds exist for some specific types of expressions.
4.2.3.2 Semantic transformations from description logic to combinations of frames and first order logic

Unlike in the previous case, most of the problems that can appear in the transformations in this direction (from a description logic format to a format based on a combination of frames and first order logic) can be solved by the use of FOL axioms. The following problems are the most typical ones to be dealt with:

- DL formats usually allow representing class conjunctions (intersection of classes) and class disjunctions (union of classes) as part of other class expressions, as shown in section 2.2. However, the frame-based KR paradigm cannot express class disjunctions. To transform such expressions from DL, FOL axioms must be created to express the disjunction of a set of classes. For instance, the expression ManOrWoman, which is defined as the union of the classes Man and Woman (ManOrWoman = Man ∪ Woman) is transformed to the class ManOrWoman, and the following FOL axiom is created stating that it is the union of both classes: ∀x (ManOrWoman(x) ↔ Man(x) ∨ Woman(x)).

- DL allows using complex expressions inside class and role definitions that do not have identifiers, whereas in frame-based formats, all the ontology components must have an identifier. Therefore, these complex anonymous expressions must be transformed into ontology components with special identifiers (anonymous classes or relations), together with FOL axioms, if necessary, that define them. For instance, the class Agent can be defined as a subclass of the union of the classes Animal or Plant (Agent ⊆ Animal ∪ Plant). To transform this expression into a frame-based format, the classes Agent, Animal, and Plant must be created, plus the “anonymous” class AnimalOrPlant, and the FOL axiom ∀x (AnimalOrPlant(x) ↔ Animal(x) ∨ Plant(x)). Consequently, a new identifier has been created, together with the FOL axiom that defines it.

- Complete (necessary and sufficient) class definitions in DL must be transformed to normal definitions in frames (which are partial definitions, as described in the previous section), plus FOL axioms to state the equivalences expressed in the definition. For instance, the class definition Flight = Travel ∩ UsesTransportMean.Plane, which states that a flight is a kind of travel that uses a plane as a transport mean, must be transformed to a frame-based format as the class Flight, which is subclass of the class Travel, and with a slot usesTransportMean whose range is Plane. Besides, the following FOL axiom must be created to state the sufficient conditions of the class: ∀x
A declarative approach to ontology translation with knowledge preservation

\[(\text{Travel}(x) \land \forall y \ (\text{usesTransportMean}(x,y) \rightarrow \text{Plane}(y)) \rightarrow \text{Flight}(x))\]

- DL format usually allow representing cycles in class taxonomies, while frame-based formats do not usually allow them. To perform such transformations, FOL axioms must be created, if they are allowed in the target format.

- The DL expressions that state that a class is subclass of the complement of another class or expression \((C \subseteq \neg D)\) are transformed to disjointness relationships between the classes involved in the expression.

- Universal property restrictions of classes are transformed into range restrictions of the slots in the corresponding class. Existential property restrictions of classes are transformed to minimum cardinality restrictions of the slots in the corresponding class.

- If the range of a role is not specified in a property restriction, the range of the corresponding slot must be set to the most general class in the ontology in the frame-based format.

4.2.4 Pragmatic layer

This layer deals with the "ability to construct the pragmatic meaning of the representation (or its meaning in context)". Therefore, in this layer we deal with transformations to be made in the resulting ontology so that both human users and ontology-based applications will notice as less differences as possible with respect to the ontology in the original format, either in one-direction transformations or in cyclic transformations.

Therefore, transformations in this layer will require, among other, the following: adding special labels to ontology components so as to preserve their original identifier in the source format; transforming sets of expressions into more legible syntactic constructs in the target format; somehow "hiding" completely or partially some ontology components that were not defined in the source ontology, but which have been created as part of the transformations (such as the anonymous classes commented in section 4.2.3.2); etc.
Figure 4.3 shows two transformations of the OWL functional object property `usesTransportMean` to WebODE. The object property domain is the class `flight` and its range is the class `airTransportMean`. The figure shows two of the possible semantically equivalent sets of expressions that can be obtained when transforming that definition. In the first one, the object property is transformed to the ad-hoc relation `usesTransportMean` that holds between the concepts `flight` and `airTransportMean`, with its maximum cardinality set to one. In the second one, the object property is transformed to the ad-hoc
relation usesTransportMean whose domain and range is the concept Thing (the root of the ontology's concept taxonomy), with no restrictions on its cardinality, plus three formal axioms expressed in first order logic: the first one stating that the relation domain is flight, the second one that its range is airTransportMean, and the third one imposing the maximum cardinality constraint⁴.

From a human user point of view, the first WebODE definition is more legible: at one glance the user can see that the relation usesTransportMean is defined between the concepts flight and airTransportMean, and that its maximum cardinality is one. In the second case, the user must find and interpret the four components (the ad-hoc relation definition and the three formal axioms) to reach the same conclusion.

The same conclusion can be obtained from an application point of view. Let us suppose that we want to instantiate the ontology with the WebODE instance editor. The behaviour of the instance editor is different for both definitions. With the first definition, the instance editor will easily "understand" that its user interface cannot give users the possibility of adding more than one instance of the relation, and that the drop-down lists used for selecting the domain and range of a relation instance will only show direct or indirect instances of the concepts flight and airTransportMean, respectively. With the second definition, the instance editor will allow creating more than one relation instance from the same instance, and will display all the ontology instances in the drop-down lists, instead of just presenting instances of flight and airTransportMean respectively. After that, the instance editor will have to run the consistency checker to detect inconsistencies in the ontology.

We will present the most relevant pragmatic problems that usually appear in the transformations between frames combined with first order logic and description logics. As in the previous section, we will consider that we are using a SHOIN(d+) language.

4.2.4.1 Pragmatic transformations from combinations of frames and first order logic to description logic

This section describes some of the most typical typical transformations at the pragmatic level that must be performed when translating ontologies from a format based on a combination of frames and first order logic to another format based on description logic. As occurred with the

⁴ We must note that this second option may be obtained because expressions in OWL ontologies may appear in any order in an OWL file, and hence may be processed independently.
semantic transformations, this list is not exhaustive but only indicative of the main problems that may appear in such scenario.

As expressed in section 4.2.3.2, class definitions in the frames KR formalism are always primitive (that is, they only express necessary conditions, but not sufficient conditions for an instance to belong to that class). To express sufficient conditions, first order logic axioms must be represented. Consequently, class definitions should be transformed to primitive class definitions in DL-based formats. The only use of these types of definitions in DL-based formats may have a negative influence when reasoning with the transformed ontologies in the target format, especially in the case of instance classification. Consequently, in some cases it may be interesting to create complete definitions from the partial definitions expressed in frames, since in most of the frame-based systems this distinction is not made.

In DL, the specification of the domain and range of properties is not usually taken into account for reasoning, since property definitions are normally taken as atomic definitions. However, in frame-based systems, this type of information is usually provided and, although it is not necessary to transform it to DL, it could be interesting to do so, if the target format allows representing it, because of legibility reasons.

As in many other target formats, independently of the KR formalism upon which they are based, in DL-based formats there are usually many different ways to express the same piece of knowledge. In the pragmatic level, we must decide which of the different but semantically equivalent forms must be created from an expression in the source format. For instance, the disjointness relationship between two classes can be expressed in DL (if possible) in the following three forms: a) with a disjointWith primitive, b) stating that the intersection of the two classes is a subclass of the predefined class bottom, and c) stating that one of the classes is a subclass of the complement of the another.

4.2.4.2 Pragmatic transformations from description logic to combinations of frames and first order logic

As in the previous section, here we collect and describe some common transformations that have to be performed at the pragmatic level when transforming from a DL-based format to a format based on a combination of frames and first order logic. Most of these transformations are related to the avoidance of using first order logic axioms when they can be rewritten using other modelling components available in the frame-based format.

Anonymous class disjunction expressions in DL are normally transformed to anonymous classes in frame-based systems, with their corresponding first order logic axioms that defined them. In
order to improve the legibility of the resulting ontology after a transformation process, these anonymous classes should be clearly identified by users with special identifiers, so that they can see that they do not directly belong to the domain, or hidden to users (this last option is normally possible only with some ontology tools, such as Protégé-2000 form personalisation or WebODE views, and not directly with ontology languages).

DL complete class definitions should be transformed to frame-based partial class definitions, which express necessary conditions, plus first order logic axioms, which express the sufficient conditions. With this type of transformation, the semantics of the transformed expression is preserved, but the legibility is lower, since many formal axioms will appear in the ontology to express this sufficient conditions. As commented in the previous section, maybe it could not be so important to express the complete class definitions and it could just suffice with the partial class definitions commonly found in frame-based systems.

If a class is defined as the union of a set of classes \( A = C_1 \cup C_2 \cup \ldots \cup C_n \), and these classes are defined as subclasses of that class, then the expression that would be normally obtained when transforming to a frame-based system would be: the class \( A \), the classes \( C_1 \) to \( C_n \), a first order logic axiom expressing that the class \( A \) is the union of all those classes, and many subclass of relationships between the classes \( C_i \) and the class \( A \). In many frame-based systems, all this information can be expressed more compactly as an exhaustive decomposition of the class \( A \). Similarly, if all the classes in the set are disjoint to each other, then the information would be expressed as a partition.

In DL, all the subclass of relationships are not necessarily expressed in ontologies, since this kind of information can be extracted from their inference engines, with automatic classification. In the transformation to frame-based systems, it could be interesting, from a pragmatic point of view, to run the automatic classifier first in order to obtain all the subclass-of relationships that can be inferred from the ontology definitions, so that they can be translated later to frame-based systems.

4.2.5 Relationships among ontology translation layers

Figure 4.4 shows an example of a transformation from WebODE to OWL DL. In this example, we have to transform two ad hoc relations with the same name (usesTransportMean) and with different domains and ranges (a flight uses an airTransportMean and a cityBus uses a bus). In OWL DL the scope of an object property is global to the ontology, and so we cannot define two different object properties with the same name. In this example we show that translation decisions have to be taken at all layers, and we also show how the decision taken at
### Table: Translation Decisions

<table>
<thead>
<tr>
<th>RDF/XML Abbrev</th>
<th>RDF/XML Abbrev</th>
<th>RDF/XML Abbrev</th>
<th>Lexical layer</th>
<th>Syntactic layer</th>
<th>Semantic layer</th>
<th>Pragmatic layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different identifiers for each object property</td>
<td>The same identifiers for both object properties</td>
<td>The same identifiers for both object properties</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No losses of expressiveness</td>
<td>Some expressiveness lost; object property can be applied to any class</td>
<td>Some expressiveness lost; the exact correspondence between domain and range is lost</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Both properties are interpreted as different things</td>
<td>Both properties are interpreted as the same. By reading the object property definition, it is not easy to know where it is applied</td>
<td>Both properties are interpreted as the same. By reading the object property definition, it is easier to know where it is applied</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Option 1 is mainly driven by semantics: in order to preserve semantics in the transformation, two different object properties are defined, with different identifiers. Option 2 is driven by pragmatics: only one object property is defined from both ad hoc relations, since we assume that they refer to the same meaning, but some knowledge is lost in the transformation (the exact information about the domain and range of the object property). Finally, option 3 is also driven by pragmatics, with more care on the semantics: again, only one object property is defined, and its domain and range is more restricted than in option 2, although we still lose the exact correspondence between each domain and range.

To sum up, in this section we have identified a catalogue of transformation problems that can appear at different layers, and hence are usually solved using different approaches and with different purposes. However, the differences between the four translation layers described should not be considered strict. In fact, we already showed in section 2.5 that there is no consensus about the levels where interoperability problems can appear, and we have also shown, especially when dealing with pragmatics, that translation decisions taken at one layer can be strongly linked to the decisions made in the other layers, and will usually have an influence on them.

**4.3 DECLARATIVE SPECIFICATION OF TRANSLATION DECISIONS**

The previous section presented how we can structure our translation decisions in four layers, which are related to each other. Now we present how we can describe the translation decisions declaratively so as to allow creating and maintaining ontology translation systems more easily, and to avoid hiding translation decisions inside the source code of such systems.

As we also showed in the previous section, the problems to be solved in each layer are different: it is not the same to transform a concept identifier from a format to another (e.g., by replacing its blank spaces with underscores) than to transform a set of logical restrictions into a disjoint decomposition of a concept. Therefore, each layer requires the use of different translation techniques.

Translation decision will be described declaratively. We will only force to program in a generic programming language very specific input/output operations and complex transformations. This is an important advantage of this approach, since it allows concentrating on the transformations between knowledge models, rather than in other generic problems that arise when programming this type of information systems.
4.3.1 Declarative specification of transformations at the lexical layer

As commented in section 4.2.1, the transformations at the lexical layer are mainly related to transforming ontology component identifiers (either because they contain characters in the source format that are illegal in the target format, because of different naming conventions in the source and target formats, or because there are differences between the source and target formats with regard to the scope of identifiers) and pieces of text that are normally used to document ontology components. These problems are normally easy to handle, because it is usually enough to take into account the rules and conventions for creating identifiers and texts in both formats.

We propose to use the language ODELex to specify all the transformations to be made at this layer. This language is similar to lex [Lesk, 1975], a widely-used lexical analyser for building compilers. While lex is aimed at building compilers, ODELex is optimised for building ontology translation systems: it restricts some of the primitives available in lex and provides specific ones related to the construction of this type of systems. We will now describe the main parts of an ODELex specification, using as an example the one that corresponds to the WebODE export service to OWL DL.

An ODELex specification is composed of three parts, as shown in the following derivation rule\(^5\), which contain: user code, declarations, and lexical rules. These parts are separated by the %% symbol. Besides, Java-style comments can be added at the beginning of the document, or inside the other parts, as will be seen later, in derivation rules 24, 25, and 26

\[
\text{(1) } \textit{ODELexDocument} ::= \{\text{comment}\} \textit{userCode} [\text{declarations}] [\text{lexRules}] \\
\]

- **User code.** The first part of an ODELex document contains the user code, where users must include any Java functions used in the rest of the specification plus any Java import statements needed for these functions. These functions usually implement complex transformations to be made to ontology component identifiers or pieces of text, or they are used as a kind of macro definitions for a sequence of transformations that have to be performed using standard functions from the Java API.

\[
\text{(2) } \textit{userCode} ::= \{\text{comment}\} \textit{javaCode} \\
\]

\(^5\) The following notation will be used to describe the derivation rules of the ODELex, ODESyntax and ODESem grammars: words in *italics* will be used for non-terminal symbols, words in **bold** font will be used for terminal symbols, alternatives will be represented with the | symbol, optional elements will be enclosed in square brackets [ ], iterations of 0 or more items will be enclosed in braces { }, and ranges of values will be enclosed in parenthesis and separated by hyphens (-).
The derivation that corresponds to the symbol javaCode is not included here, since it corresponds to the grammar used for any Java file, and hence can be found in other documents that deal with that programming language. In our example, we have defined two Java functions:

- **convertToURI**, which transforms a string value into a corresponding string value that is a valid URI.

- **addNumber**, which adds the characters "_1" to the identifier that it receives as an input.

```java
import java.net.*;

private String convertToURI (String id)

    URI id_URI = new URI(URLEncoder.encode(id,"ISO-8859-1"));
    return id_URI.toString();
}

private String addNumber (String identifier)

    return (identifier + "_1");
}
```

The definitions included in the user code part will be copied verbatim into the source file of the **LexicalMapper.java** file generated from this specification. In this part there is also the possibility of referencing transformation functions from a lexer created with typical lexical analysis tools such as lex [Lesk, 1975; Levine et al., 1992], JLex^6 [Berk, 1997], etc.

- **declarations.** This part of an ODELex specification contains the declaration of which ontology components (both from the source format and from the target format) will be dealt with by the lexical transformation tool, and of which ontology components of the target format cannot overlap (which means that they cannot share the same identifiers because they share the same scope).

(3) declarations :: {comment | componentDecl | overlapDecl}

(4) componentDecl :: idComponent [%transient] [scopeDecl]

(5) scopeDecl :: %scope (idScope {, idScope})

(6) idScope :: idComponent | id

(7) overlapDecl :: no-overlap (idComponent, idComponent {, idComponent})

In the example below, the declaration part defines seven WebODE ontology components (concepts, instance attributes, class attributes, ad hoc relations, instances, references, and
axioms) and two types of values (values and documentation). With respect to OWL, it defines four ontology components (classes, object properties, datatype properties, instances) and two types of values (datatype values, and label and comments). It also states that the sets of identifiers of the four ontology components must be disjoint (they cannot overlap).

Three of the WebODE components are not first class citizens of the ontology, since they are defined inside the scope of others: instance and class attributes are defined inside the scope of concepts, and ad hoc relations are defined inside the scope of two concepts (their domain and range).

Finally, the %transient keyword states that once that we have performed a transformation of the corresponding component, it is not interesting to store the result of the transformation. In the example, this happens with attribute values and pieces of text used to document ontology components in both formats.

```java
/* WebODE components */
WebODE.Concept
WebODE.InstanceAttribute %scope (WebODE.Concept)
WebODE.ClassAttribute %scope (WebODE.Concept)
WebODE.AdHocRelation %scope (WebODE.Concept,WebODE.Concept)
WebODE.Instance
WebODE.Reference
WebODE.Axiom
WebODE.Value %transient
WebODE.Documentation %transient

/* OWL components */
OWL.Class
OWL.ObjectProperty
OWL.DatatypeProperty
OWL.Instance
OWL.DatatypeValue %transient
OWL.LabelComment %transient
no-overlap (OWL.Class, OWL.ObjectProperty,
OWL.DatatypeProperty, OWL.Instance)
```

If a component is not included in this declaration part, its identifier or the corresponding piece of text will not be transformed because it is legal in the target format, and hence do not have to be transformed. This part of the code will be used to generate the file LexicalTypes.java that contains an interface with the components to be used in the lexical transformations. This interface will be used by the rest of lexical tools, and also by the tools from other layers.

- **Lexical transformation rules.** This part of an ODELex specification contains the actual transformations that have to be performed to each ontology component of the source format in order to obtain its correspondence in the target format.

(8) `lexRules :: {lexRule}`

(9) `lexRule :: ruleHeader CR init CR table CR repeated CR overlap CR`

(10) `ruleHeader :: % idComponent IDENTIFIER {idComponent IDENTIFIER}`

As shown in rule 9, for each component specified in the lexical rule header the following information must be specified:

- **INIT:** it specifies the initial transformation to be performed. For instance, in the example below we propose to transform the identifier of a WebODE ad hoc relation to an OWL ObjectProperty by converting the identifier to a URI, with the function `convertToURI` specified above.

  (11) `init :: INIT:{javaCode}`

- **TABLE:** if the component is not transient, it specifies the information to be stored so as to obtain it later either in the source format or in the target format. In the example below we propose to store the ad hoc relation identifier, and its two associated concept identifiers in the table `WebODE.AdHocRelation`, and the corresponding identifier obtained from the transformation in the table `OWL.ObjectProperty`, maintaining the corresponding links to each other.

  (12) `table :: TABLE:{tableDecl}`

  (13) `tableDecl :: (tableColumn , tableColumn)`

  (14) `tableColumn :: [idComponent , numberPosition {, numberPosition}]`

  (15) `numberPosition :: % number | $ number`

- **REPEATED:** if the component is not transient and cannot be repeated under certain circumstances, it specifies the alternative transformation to be made. For instance, if after the transformation the OWL object property identifier already existed as an OWL object property identifier and the WebODE ad hoc relation with the same ad hoc relation identifier and domain concept already existed, then we propose to maintain the identifier to be provided for the transformed object property (and we obtain it with the predefined function `GET`). The same applies if the same ad hoc relation identifier and range concept already existed. In another situation, a new
identifier is created by adding the character “1” to the current transformed identifier.

(16) repeated :: REPEATED:{ transformation {, transformation} }

(17) transformation :: tablePatternColumn ==> {javaCode} |

  default ==> {javaCode}

(18) tablePatternColumn :: [idComponent , numberPatternPosition |

  (, numberPatternPosition)]

(19) numberPatternPosition :: % number |

- OVERLAP: if the component is not transient and there cannot be overlaps in the target format, it specifies the transformation to be performed to the already generated identifier. This is repeated until there is no overlap. In the example, we propose to add a number to the OWL object property identifier until there is no collision with other class, datatype property and instance identifiers (not object properties).

(20) overlap :: OVERLAP:{javaCode}

```
%WebODE.AdHocRelation IDENTIFIER WebODE.Concept IDENTIFIER WebODE.Concept IDENTIFIER
/* The 2nd and 3rd identifiers are of the domain and range concepts */
INIT: {$l=convertToURI(%1)}
TABLE:{([WebODE.AdHocRelation,%1,%2,%3],[OWL.ObjectProperty,$1])}
REPEATED:
  [[WebODE.AdHocRelation,%1,%2,_]==> 
   {$1=GET([WebODE.AdHocRelation,%1,%2,_])},
  [WebODE.AdHocRelation,%1,_,%3]==> 
   {$1=GET([WebODE.AdHocRelation,%1,_,%3])},
  default==> {$1=addNumber($1)}
OVERLAP: {$l=addNumber($1)}
```

This part of the ODELex specification will be used to generate most of the LexicalMapper.java file, which contains the lexical tools to be used by the tools generated by other layers in order to access the lexical information of each ontology component.

Finally, the following derivation rules are used for creating identifiers, Java-style comments, numbers and the end of line symbol. They have been used in the previous rules.

(21) idComponent :: idFormat . id
4.3.2 Declarative specification of transformations at the syntactic layer

Given that one of our assumptions is that both the source and the target formats of the transformations have their own Java APIs defined, the transformations to be expressed in this layer are also simple, as occurred with the lexical layer. With regard to the source format, specifications at this layer describe the correspondence between each component and its accessors (either for the component itself, for all the components of a specific type, or for pieces of information of each component). With regard to the target format, specifications at this layer describe the correspondence between each component and its constructors, adding and updating methods, as well as the accessors that might be needed. Besides, in this layer it is also specified the correspondence between the attribute datatypes of the source and target formats.

As occurred in the lexical layer, in the syntactic layer we propose the use of the language ODESyntax, which allows specifying all the correspondences outlined above. This language is also based on another one available for building compilers, such as yacc [Johnson, 1975]. A specification in this language is also divided in several parts, which will be described in detail below, using examples that correspond to the WebODE export service to OWL DL. The following rule shows that there are five parts: user code, declarations, accessors, constructor and updates, and datatype transformations.

(1) `ODESyntaxDocument :: {comment} %%% [userCode] %%% [declarations]
    %%% [accessDecls] %%% [updateDecls] %%% [datatype]`

- **User code.** Like with ODELex, the first part of an ODESyntax specification contains the user code, where all the auxiliary Java functions to be used in the rest of the specification are included. These functions usually implement complex syntactic transformations (e.g.,
of attribute datatypes) or they are used as a kind of macro definitions for transformations, accessors, updaters, etc., that have to be implemented as sequences of functions from the standard Java API or from the APIs of the source and target formats. The corresponding derivation rule is the same as in ODELex:

\[(2) \text{userCode :: \{comment\} } \%\text{(javaCode)}\%

The following examples, taken from the WebODE export service to OWL DL, show some auxiliary functions defined in this part:

- **getConcepts**, which receives as an input an ontology name and returns all the concepts in the WebODE ontology. This function is defined here because the current WebODE ontology access API does not provide such a method.

- **addComment**, which receives the class where the comment must be added and the text of the comment. This function is defined here because in the Jena API (used to generate the target OWL ontology) the language in which the comment text is available must be specified. This function ensures that this language is set to null. As we will see later, this could have been omitted, since this function is used mostly as a macro.

- **removeAllSuperclasses**, which receives an OWL class and removes all its references to its superclasses. This function is defined here because the Jena API does not provide a specific function for performing this action, and it will be used in one of the following parts of the ODESyntax specification.

```java
%(//Import statements for the source format
import es.upm.fi.dia.ontology.webode.service.*;
//Import statements for the target format
import com.hp.hpl.jena.ontology.*;

private Concept[] getConcepts(String ontology){
    return (Concept[] (ode.getTerms
    (ontology, new int [] {TermTypes.CONCEPT})));}

private void addComment(OntClass class, String commentText) {
    class.addComment(commentText, null);
    //the second argument is used for specifying the language
}

private void removeAllSuperclasses(OntClass class){
    ExtendedIterator iter = class.listSuperClasses();
    if (iter!=null){
        while (iter.hasNext()){ 
            class.removeSuperClass((OntClass)iter.next());
        }
    }
}
```
The definitions included in the user code part will be copied verbatim into the source file of the \texttt{SyntaxMapper.java} file generated from this specification. We also consider the possibility of referencing transformation functions from a syntax analyzer created with typical syntax analysis tools such as yacc [Johnson, 1975; Levine et al., 1992], JCup\textsuperscript{7} [Hudson, 1999], etc.

- \textbf{Declarations}. Like in ODELex, this part of an ODESyntax specification contains the declaration of which ontology components will be dealt with in the specification. This component list does not need to contain the same components than the ODELex one, since it has a different purpose (syntactic transformations instead of lexical ones). Besides, it does not have to consider whether a component is transient or not, nor whether there can be overlap or not with other components, since these problems are already solved by the lexical layer specification. The following rules are used to create this declaration part of the ODESyntax document:

\begin{verbatim}
(3) declarations :: {comment | namespaceDecl | componentDecl }

(4) namespaceDecl :: %NAMESPACE id javaPackage ;

(5) componentDecl :: idComponent [scopeDecl] : javaClassID CR

(6) scopeDecl :: %scope ( idComponent | idComponent )
\end{verbatim}

The ODESyntax component list is usually based on the knowledge models of the source and target formats, and usually corresponds as well to their APIs. If there is a strong correspondence between the format API and its knowledge model (either of the source or of the target format), the transformations to be specified in ODESyntax are quite straightforward. If not, the transformations are more complex and hence more effort is needed to construct the ODESyntax specification.

The declaration part also includes a list of namespaces, which is used to abbreviate long Java packages in the rest of the specification. To refer to these namespaces, the corresponding namespace identifier must be placed between square brackets (e.g., [ode] to refer to \texttt{es.upm.fi.dia.ontology.webode.service}).

The declaration part of the example below defines two namespaces (\texttt{ode} and \texttt{jenaOnt}), which correspond to the packages where the knowledge models of the source and target

\textsuperscript{7} \url{http://www.cs.princeton.edu/~appel/modenl/java/CUP/}
formats are defined. It also defines the ontology components from the source and target formats that will be used for the syntactic transformations, together with their scope, in case that they depend on another components, and with their corresponding Java class or interface. In the case that a component does not have an interface or class defined for it in the format API, we should create it in the user code part or in an external file, and make references to it from this specification.

```java
%NAMESPACE ode es.upm.fi.dia.ontology.webode.service;
%NAMESPACE jenaOnt com.hp.hpl.jena.ontology;

/* WebODE components */
WebODE.Ontology : [ode]OntologyDescriptor
WebODE.Concept : [ode]Concept
WebODE.Group : [ode]Group
WebODE.InstanceAttribute : [ode]InstanceAttributeDescriptor
WebODE.ClassAttribute : [ode]ClassAttributeDescriptor
WebODE.AdHocRelation : [ode]TermRelation
WebODE.Reference : [ode]ReferenceDescriptor
WebODE.Axiom : [ode]FormulaDescriptor
WebODE.Instance : [ode]Instance
WebODE.InstanceSet : [ode]InstanceSet

/* OWL components */
OWL.Ontology : [jenaOnt]Ontology
OWL.Class : [jenaOnt]OntClass
OWL.ObjectProperty : [jenaOnt]ObjectProperty
OWL.DatatypeProperty : [jenaOnt]DatatypeProperty
OWL.Instance : [jenaOnt]Individual
```

Unlike in ODELex, the declaration part of an ODESyntax specification must include all the components that will be managed by the syntax transformations.

- **Accessor methods.** For each ontology component of the source and target formats that has been declared in the previous piece of code, this part of an ODESyntax specification may declare the methods to be used for the following purposes:
  
  - To access all the ontology components of a specific type (e.g., a method that retrieves all the concepts of an ontology).  

A declarative approach to ontology translation with knowledge preservation

- To access a specific ontology component of a specific type (e.g., a method that retrieves a specific concept of an ontology, given its identifier).

- To access different pieces of information of the ontology component (e.g., methods or properties to access a concept identifier, a concept description, the superclasses of a concept, etc.).

The following rules show how we can declare these methods:

(7) accessDecls :: {accessDecl}

(8) accessDecl :: header CR [all CR] [individual CR] [information CR]

(9) header :: % idComponent IDENTIFIER {idComponent IDENTIFIER}

(10) all :: ALL:{functionDecl {;functionDecl}}

(11) individual :: INDIVIDUAL:{functionDecl {;functionDecl}}

(12) functionDecl :: number : id (parameters) : javaClassID array

(13) parameters :: % number {, % number} | λ

(14) array :: [ ] | λ

(15) information :: INFORMATION:{informDecl {;informDecl}}

(16) informDecl :: id : id [(parameters)] : javaClassID array

As shown in the previous rules, the first group of accessors will be included in the ALL keyword, the second group will be included in the INDIVIDUAL keyword, and the third group will be included in the INFORMATION keyword. For each method in the first and second groups, we will indicate a number that identifies the method (because there can be different methods with the same purpose), the method name and parameters, and the object that it returns. For the third group, we specify the piece of information's identifier (which determines how we will access this information from the semantic and pragmatic layers), the property or method to be used to access to that information given a specific object of that type, and the datatype.

The following example shows the declaration corresponding to an ontology component of the source format: a WebODE instance attribute. It shows that instance attributes are defined inside the scope of an ontology and of a concept. There are two methods that allow accessing all the instance attributes of a concept in an ontology: the first one is used to access the instance attributes that are defined explicitly in that concept, and the second one
returns also those inherited through the concept taxonomy. Both of them return an array of objects of the class `InstanceAttributeDescriptor`.

There is one method to get a specific instance attribute, given the ontology name, the concept name and the attribute name.

Finally, the following information can be accessed from an instance attribute: the concept to which it belongs, the attribute name, the attribute description, its type, its maximum and minimum cardinality, its maximum and minimum value, and the set of explicit values. They are accessed using properties of the Java class `InstanceAttributeDescriptor`, except for the attribute type, which is accessed with an ad hoc function defined in the user code part.

```java
%WebODE.InstanceAttribute IDENTIFIER
WebODE.Concept IDENTIFIER
WebODE.Ontology IDENTIFIER
ALL:
{1:getInstanceAttributes(%3,%2):[ode]InstanceAttributeDescriptor[];
 //gets instance attributes defined locally to the concept
2:getInstanceAttributes(%3,%2,true)
 :[ode]InstanceAttributeDescriptor[]
 //gets also inherited instance attributes
}
INDIVIDUAL:
{1:getInstanceAttribute(%3,%2,%1):[ode]InstanceAttributeDescriptor}
INFORMATION:
{concept :termName :String;
Name :name :String;
Description :description :String;
Type :getValueTypeName(valueType) :String;
MaxCard :maxCardinality :int;
MinCard :minCardinality :int;
MaxValue :maxValue :float;
MinValue :minValue :float;
Values :values :String[]}
```

The following example shows the declaration that corresponds to an OWL class (an ontology component of the target format). It shows that OWL classes are not in the scope of other components, that there is a method that lists all the classes of an ontology, that there is also a method to access a specific class, given its identifier, and that we can access to the class name, description and to its superclasses. In some of these cases a `java.util.Iterator` is returned.

```java
%OWL.Class IDENTIFIER
ALL: {1:listClasses() :java.util.Iterator}
INDIVIDUAL: {1:getOntClass(%1) :[jenaOnt]OntClass}
INFORMATION:
{name :getURI() :String;
description :listComments(null) :java.util.Iterator;
subclassOf :listSuperClasses() :java.util.Iterator}
```
As commented for the declaration part, if a specific method or property that we want to specify in this part is not defined in the ontology access API of a format, we can define them either in the user code part of the ODESyntax specification or in a separate file that will extend the current API.

- **Constructor and update methods.** For each ontology component of the target format that has been declared in the declaration part, this part of an ODESyntax specification may declare the methods that will be used for the following purposes:
  
  - To create the ontology component in the target ontology (e.g., a constructor or a method that creates an OWL class). We use the keyword `CREATE`.
  
  - To remove the ontology component from the target ontology (e.g., a method that removes a class from an OWL ontology). We use the keyword `REMOVE`.
  
  - To add information about the ontology component (e.g., the method or property to be used to add information about the class documentation, the class superclasses, etc.). We use the keyword `ADD`.
  
  - To remove all the information about a specific piece of information of the ontology component (e.g., the method or property to be used to remove all the class documentations, all the class superclasses, etc.). We use the keyword `REMOVEALL`.
  
  - To remove a value from a specific piece of information of the ontology component (e.g., the method or property to be used to remove a specific class documentation, a specific class superclass, etc.). We use the keyword `REMOVEINDIVIDUAL`.

The following derivation rules show the syntax to be used to declare these constructors and update methods:

```plaintext
(17) updateDecls :: {updateDecl}

(18) updateDecl :: header CR [create CR] [remove CR] [add CR]
    [removeall CR] [removeindividual CR]

(19) create :: CREATE:{ creatoremDecl {; creatoremDecl } }

(20) remove :: REMOVE:{ creatoremDecl {; creatoremDecl } }

(21) creatoremDecl :: number : id (parameters )

(22) add :: ADD:{ addremDecl {; addremDecl } }

(23) removeall :: REMOVEALL:{ addremDecl {; addremDecl } }
```
(24) \texttt{removeindividual :: \texttt{REMOVEINDIVIDUAL}:{\langle addremDecl ; addremDecl \rangle}}

(25) \texttt{addremDecl :: id : id (parameters)}

For each method in the first and second groups, we will indicate a number that identifies the method (because there can be different methods with the same purpose), the method name and parameters. For the rest of groups, we specify the piece of information's identifier (which determines how we will access this information from the semantic and pragmatic layers), and the property or method to be used to add, remove all, and remove one of the value(s) of that piece of information, respectively. Not all the pieces of information must be specified in all the cases, but only those ones that will be used by the transformations in the semantic and pragmatic layers.

The following example shows the declaration corresponding to an OWL class. It shows that there is one method that allows creating classes in the target ontology, \textit{addClass}, which receives as an input the identifier of the class. It also defines the methods that can be used to add a natural language description to the class and to remove all the descriptions, and the ones to be used to add a superclass, to remove all the class superclasses, and to remove a specific superclass. The \%1 parameter refers to the OWL class identifier, and the $1$ in the parameter means that the value for this parameter (the actual description and the class identifier) will be provided by the semantic or pragmatic layers in the first order. We will see how to define this information in the semantic and pragmatic transformation layers.

\begin{verbatim}
%OWL.Class IDENTIFIER
CREATE: {1:createClass(%1)}
ADD:
  {description :addComment(%1,$1);
   subclassOf :addSuperClass(createClass($1))}
REMOVEALL:
  {description :removeAllComments(%1);
   subclassOf :removeAllSuperclasses(%1)}
REMOVEINDIVIDUAL:{subclassOf :removeSuperClass(createClass($1))}
\end{verbatim}

As commented for other parts of the specification, if a specific method or property used in this part is not defined in the ontology access API of a format, we can define them either in the user code part of the ODESyntax specification or in a separate file that will extend the current API.

- **Datatype transformations.** This part of an ODESyntax specification contains the transformations that have to be made to the attribute datatypes that appear in the source format so as to transform them to the target format. This specific feature, which may have been included in the previous part of the specification (constructors and updater methods), aims to make it easier to specify these common transformations.
The following derivation rules are defined in the ODESyntax grammar to deal with this type of transformations:

(26) \textit{datatype} :: \{\textit{datatypeTransf}\}

(27) \textit{datatypeTransf} :: " \textit{datatypeID} " : " \textit{datatypeID} " ; |

\hspace{1cm} \texttt{default %1} : (\textit{datatypeID} | %1 | \{\textit{javaCode}\})

The following example shows how to specify these transformations. In the first column, we specify the String term used to identify the type in the source format. The second column specifies the correspondence for each datatype of the source format in the target format (in this example, all of them are XML Schema datatypes, since OWL DL only allows them to specify datatypes). The transformations will be checked sequentially, according to the order specified in this table. Besides, the "default" keyword can be used at the last line to specify any other kind of datatype that could be found.

/* Datatype transformations */
"boolean": "http://www.w3.org/2001/XMLSchema#boolean";
"cardinal": "http://www.w3.org/2001/XMLSchema#nonNegativeInteger";
"integer": "http://www.w3.org/2001/XMLSchema#integer";
"float": "http://www.w3.org/2001/XMLSchema#float";
"string": "http://www.w3.org/2001/XMLSchema#string";
"date": "http://www.w3.org/2001/XMLSchema#date";
"range": "http://www.w3.org/2001/XMLSchema#float";
"URL": "http://www.w3.org/2001/XMLSchema#anyURI";
\hspace{1cm} \texttt{default %1: %1;}

Finally, there are some non-terminal symbols that have been used by the previous rules of the ODESyntax grammar and whose derivation rules have not appeared yet. Besides the rules of this type that were defined for ODELex, we have the following additional ones:

(36) \textit{javaClassID} :: \{id\} id

(37) \textit{javaPackage} :: id . \{id .\}

(38) \textit{datatypeID} :: \{a-z,A-Z,0-9,.;/#\}

\subsection*{4.3.3 Declarative specification of transformations at the semantic and pragmatic layers}

The previous sections have described how to specify declaratively the transformations to be made in the lexical and syntactic layers. The main objective of both transformation layers is to abstract the low-level details of the source and target formats (their syntax specific features, the restrictions and naming conventions of ontology component identifiers, etc.), so as to allow specifying the semantic and pragmatic transformations at a higher abstraction level.
In our proposal, we have considered the assumption that the problems to be solved in the semantic and pragmatic layers are the most important (and complex) ones, and they decide which transformations have to be performed by the ontology translation system. Hence the importance of abstracting low-level details of the source and target formats so as to allow knowledge engineers to focus on the transformations between their knowledge models.

As described in section 4.2.3, the problems to be solved in the semantic layer are mainly related to complex transformations of expressions that go beyond the rather simple limits of syntax and whose general aim is usually the meaning preservation of the knowledge transformed. Some examples of such transformations are:

- Transform a component of the source format into several components in the target format. For instance, an OWL value restriction is transformed to a WebODE instance attribute and an explicit value assignment to that instance attribute.

- Transform a set of components of the source format into one component in the target format. For instance, a set of OWL *disjointWith* expressions and *subclass of* relationships are transformed into a WebODE disjoint decomposition.

- Transform a component of the source format to different components of the target format depending on some conditions. For instance, an RDF property must be transformed to a WebODE ad hoc relation if its range is an RDF class, and to a WebODE instance attribute if its range is a literal or an XML Schema datatype.

As described in section 4.2.4, the problems to be solved in the pragmatic layer are those that permit both human users and ontology-based applications to notice as less differences as possible in the ontologies in the original and target formats. Examples of such transformations are:

- **Pre-processing transformations.** They usually consist in the creation of predefined ontology components in the target format so as to facilitate the transformation of other ontology components during the semantic processing. For example, creating the meta-class :WebODEConcept in Protégé-2000 so as to store additional information about the ontology concepts being transformed from WebODE to Protégé-2000

- **Post-processing transformations.** They usually consist in transforming sets of expressions in the target format, which have been created as a result of the semantic transformations, into more legible (and usually equivalent) syntactic constructs in the target format. Some examples of such transformations are: transforming a set of WebODE formal axioms that define that several concepts are disjoint to each other into a WebODE concept group;
transforming a domain of an OWL object property of the form $C_1 \cup C_2 \cup \ldots \cup C_n$ into a more general concept that subsumes the union of that concept; etc.

- **In-processing transformations.** Pragmatic decisions can be also taken during the processing of individual ontology components of the source format. These transformations differ from the semantic ones in the fact that they do not change the semantics of the knowledge transformed, but only how it can be interpreted by humans. Examples of such decisions are: whether to transform an RDF property without range to a WebODE instance attribute with type `String`, to a WebODE ad hoc relation with the ontology root concept as the range, or to both; which name should be assigned to the WebODE formal axiom derived from an OWL complete class definition; which name should be assigned to the WebODE anonymous class obtained from an OWL union of classes; etc.

- **Other transformations.** In this group we include any other transformations and decisions to be taken during the ontology translation process for pragmatic reasons, and that cannot be easily classified in the previous groups. For instance, hiding in the Protégé-2000 user interface the additional ontology components used to transform a WebODE ontology, so that users cannot find differences with the standard Protégé-2000 knowledge model.

The translation decisions in both layers will be specified with the same declarative language: ODESem. As with the other two languages presented in this chapter (ODELex and ODESyntax), we will now describe the parts in which this language is divided. The first derivation rule of the grammar shows that an ODESem specification is divided into three parts: user defined code, declarations and semantic and pragmatic rules:

(1) $ODESemDocument :: \{\text{comment}\} \%\% \text{[userCode]} \%\% \text{[declarations]} \%\% \text{[semRules]}$

- **User code.** Like with the other two languages, the first part of an ODESem specification contains the user code, where all the auxiliary Java functions to be used in the rest of the specification are included. As we have commented above, there might be a need for performing complex transformations that might not be easily represented with the primitives used in the other parts of the specification. In this case, this user defined functions can be used from those parts. The grammar rule used for the user code part of an ODESem specification is like the ones used in the other two languages:

(2) $userCode :: \{\text{comment}\} %(javaCode)%$

The following examples, taken from the WebODE export service to OWL DL, show some auxiliary functions defined in this part:
- `obtainDatatypes` and `obtainDomains`, which receive as an input the identifier of an OWL datatype property and return all the datatypes and domains associated to it, as stored in the lexical transformation mapping tools created from the ODELex specification. These functions are provided as shortcuts for the use of the predefined lexical function `obtainDistinctSourceParametersFromTargetId`.

- `obtainDatatype` and `obtainDomain`, which are similar to the previous ones, but return only one datatype and domain, respectively. These functions are used when the datatype property has only one datatype or range.

- `allDatatypesEqual` and `allDomainsEqual`, which receive as an input the identifier of an OWL datatype property and return whether it has only one datatype or not, and one domain or not, respectively. These functions are also shortcuts for the use of the predefined lexical function `obtainDistinctSourceParametersFromTargetId` and the check of the return value length.

```java
private String[] obtainDatatypes (String propID){
    return obtainDistinctSourceParameterFromTargetID
        (LexicalMapper.OWL_DatatypeProperty, propID, 2);
}

private String obtainDatatype (String propID){
    return obtainDistinctSourceParameterFromTargetID
        (LexicalMapper.OWL_DatatypeProperty, propID, 2)[0];
}

private String[] obtainDomains (String propID){
    return obtainDistinctSourceParameterFromTargetID
        (LexicalMapper.OWL_DatatypeProperty, propID, 1);
}

private String[] obtainDomain (String propID){
    return obtainDistinctSourceParameterFromTargetID
        (LexicalMapper.OWL_DatatypeProperty, propID, 1)[0];
}

private boolean allDatatypesEqual (String propID){
    return (obtainDistinctSourceParameterFromTargetID
        (LexicalMapper.OWL_DatatypeProperty, propID, 2).length<=1);
}

private boolean allDomainsEqual (String propID){
    return (obtainDistinctSourceParameterFromTargetID
        (LexicalMapper.OWL_DatatypeProperty, propID, 1).length<=1);
}
```

As with the other two languages, the user code part will be copied verbatim into the source file of the `SemMapper.java` file generated from this specification.

**Semantic and pragmatic transformation rule declarations and processing order.** This part of an ODESem specification contains the declaration of the transformation rules that will be defined later, together with the order in which they have to be processed. Unlike in the rest of declarative specifications, where the order of the definitions is not relevant, in
this case the processing order of the semantic and pragmatic transformations might be relevant and hence it is considered in this specification.

The following rules of the ODESem grammar define how these rules and their processing order are defined:

(3) \textit{declarations} :: \{comment \mid \textit{ruleDecl} \}

(4) \textit{ruleDecl} :: \textit{number} : \% \textit{id} ;

The following piece of code shows these declarations for the WebODE export service to OWL DL. This export service does not need to execute pre-processing rules. The rules 1 to 7 are in charge of the semantic and pragmatic transformations of the components found in the source format (the WebODE ontology). As we can infer from the rule names, the ontology translation system will first add general information about the ontology (ontology container), the classes, the object properties, the disjoint and exhaustive decompositions, and the ontology instances. Finally, three pragmatic post-processing transformations will be performed, in order to remove redundant domains in OWL datatype property definitions, and redundant domains and ranges in OWL object property definitions.

/* Semantic and pragmatic transformation rule declaration, and processing sequence */
1: \%AddOntologyContainer;
2: \%AddClasses;
3: \%AddObjectProperties;
4: \%AddDisjointDecompositions;
5: \%AddExhaustiveDecompositions;
6: \%AddInstances;
7: \%PostProcessing\_RemoveDPRedundantDomains;
8: \%PostProcessing\_RemoveOPRedundantDomains;
9: \%PostProcessing\_RemoveOPRedundantRanges;

- \textbf{Semantic and pragmatic transformation rules}. The last part of an ODESem specification contains the rules to be applied in order to transform the ontology components in the source format to the corresponding ontology components in the target format. This part of the specification must at least contain the rules declared previously, plus any other auxiliary rules that can be called from these ones.

A semantic and/or pragmatic transformation rule is referenced to with an identifier, which is preceded by the symbol ‘\%’. This is what is called header in the ODESem grammar, as shown below:

(7) \textit{semRules} :: \{\textit{semRule} \mid \textit{comment}\}

(8) \textit{semRule} :: \textit{header} \textbf{CR} \textit{lhs} \textbf{-->} \{ \textit{rhs} \}

(9) \textit{header} :: \% \textit{id}
Besides, as shown in the previous derivation grammar rule, a transformation rule is defined with two parts: the left hand side (LHS) and the right hand side (RHS):

- The LHS (Left Hand Side) of the rule, aka antecedent, contains the information needed to trigger the rule, which can be either the source ontology component to be transformed or a target component to be modified. In both cases, the antecedent is defined with the ontology component type, as defined in the ODESyntax specification, and with an identifier that will be used to refer to the specific component in the RHS (Right Hand Side) of the rule. If no information is needed to trigger the rule, the keyword NULL must be used.

(10) \( \text{lhs} :: \text{idComponent id | NULL} \)

- The RHS of the rule, aka consequent, contains the sequence of actions that have to be performed in order to obtain the corresponding ontology component(s) in the target format.

The following rule of the ODESem grammar defines the set of actions that can be performed in the RHS of the transformation rule:

(11) \( \text{rhs} :: \{ \text{create} | \text{add} | \text{remove} | \text{removeall} \\
\text{exec} | \text{ifThen} | \text{forEach} \\
\text{error} | \text{assign} | \text{functionCall} \} ; \)

These actions can be grouped in three types: actions that create new ontology components in the target format, and add or remove components or information; actions that specify the control flow of the translation system; and general actions used to throw error messages, assign values to variables or call other functions, either predefined or defined by the user.

Let us start with the first group of actions, whose corresponding piece of grammar is shown below:

(18) \( \text{create} :: \text{CREATE ( idComponent , number , var \{ , var\})} \)
(19) \( \text{add} :: \text{ADD ( id , id , (create | var | getComponents) )} \)
(20) \( \text{remove} :: \text{REMOVE ( id , id , var) } \)
(21) \( \text{removeall} :: \text{REMOVEALL ( id , id )} \)

- CREATE. It creates (and returns) an ontology component in the target ontology.

This action receives as input parameters the ontology component type to be created,
the number of the specific constructor to be used, and the rest of parameters needed to create the ontology component. All the information needed (the ontology component type, the number of the specific constructor and the number and order of the additional input parameters) must follow the restrictions coded in the ODESyntax specification.

- ADD. It adds one or several values to a specific property of an ontology component of the target ontology. The ontology component, the property and the value(s) are specified as input parameters.

- REMOVE. It removes a value from a specific property of an ontology component of the target ontology. The ontology component, the property and the value to be removed are specified as input parameters.

- REMOVEALL. It removes all the values from a specific property of an ontology component of the target ontology. The ontology component and the property are specified as input parameters.

The actions specified in the consequent of a transformation rule are executed sequentially. Besides, the following control flow structures can be used:

\[
\begin{align*}
(22) & \quad \text{exec} :: \text{EXEC} (\% \text{id}, \text{var}) \\
(15) & \quad \text{ifThen} :: \text{if} (\text{condition}) \; \{\text{rhs}\} \; \text{else} \; \{\text{rhs}\} \\
(16) & \quad \text{condition} :: \text{javaComparison} \; | \text{functionCall} \\
(17) & \quad \text{forEach} :: \text{forEach id} \; \text{IN} \; \text{var} \; \{\text{rhs}\} \\
\end{align*}
\]

- EXEC. It starts the execution of a rule, with the set of parameters that match its antecedent.

- If condition \{actions\} else \{actions\}. It specifies the set of actions to be performed if the condition specified is evaluated as true (then body), and, optionally, the set of actions to be performed if the condition specified is evaluated as false (else body).

- ForEach variable IN set_variable \{actions\}. It specifies the set of actions to be performed for each ontology component inside the multiple-valued variable.

Finally, any variable assignments can be used, using the form \text{var} = \text{value}, and the predefined functions GETCOMPONENT and GETALLCOMPONENTS can be used to obtain a specific component or all the components of either the source or the target format (with the parameters specified in the ODESyntax specification). The ERROR
function can be used in cases where the options that allow executing it are not allowed. These actions are considered in the following ODESem grammar rules:

(12) \[ \text{assign} :: \text{id} = \{ \text{create} | \text{functionCall} | \text{getComponents} \] 

(13) \[ \text{functionCall} :: \text{id} (\text{parameters}) \] 

(14) \[ \text{parameters} :: \text{id} \{, \text{id} \} | \lambda \] 

(26) \[ \text{getComponents} :: \text{GETCOMPONENT} (\text{idComponent}, \text{id}) | \] 

\[ \text{GETALLCOMPONENTS} (\text{idComponent}, \text{id}) \]

Below we provide the code of some transformation rules of the WebODE export service to OWL DL, specifically those that transform WebODE concepts and their instance attributes in OWL classes and datatype properties.

The first rule \((\text{AddClasses})\) appeared in the declaration and processing order part of the specification. For each concept defined in WebODE, it creates an OWL class with the concept name, it adds a natural language description, in case that it exists and that the concept is not an imported term from another ontology, and it states that it is a subclass of the parent concepts, as specified in the source format. Finally, for each instance attribute of the concept, the rule \(\text{ADDInstanceAttributes}\) is triggered.

\[
%\text{AddClasses}
\]

\[
\text{WebODE.Concept concept} --> \\
\{ \text{C = CREATE(OWL.Class,1,concept.name);} \\
\text{if (concept.description != null && !concept.isImported)} \\
\text{ADD(C,description,concept.description);} \\
\text{ADD(C,subClassOf,concept.parentConcepts);} \\
\text{forEach ia IN concept.instanceAttributes} \\
\text{EXEC(%AddInstanceAttributes,WebODE.InstanceAttribute,ia);} \\
\}
\]

The second rule \((\text{ADDInstanceAttributes})\) did not appear in the declaration and processing order part of the specification, hence it is considered as an auxiliary rule. It creates an OWL datatype property out from a WebODE instance attribute, and implements a decision tree that depends on whether there are several instance attributes in WebODE that produce the same OWL identifier or not, and in the first case, whether all those instance attributes have the same datatype associated or not. Depending on these conditions, the rules \(\text{ADDInstanceAttributes}_1\) or \(\text{ADDInstanceAttributes}_2\) will be triggered, or an error will be obtained.

\[
%\text{AddInstanceAttributes}
\]

\[
\text{WebODE.InstanceAttribute instAttr -->} \\
\{ \text{P = CREATE(OWL.DatatypeProperty,1,propertyID);} \\
\text{if (targetIDhasSeveralSources (WebODE.InstanceAttribute,} \\
\text{instAttr.name,instAttr.concept,instAttr.type))} \\
\text{if (allDatatypesEqual (WebODE.InstanceAttribute,} \\
\text{instAttr.name,instAttr.concept,instAttr.type))} \\
\}
\]
A declarative approach to ontology translation with knowledge preservation

```plaintext
instAttr.name, instAttr.concept, instAttr.type))
EXEC(%AddInstanceAttributes_1,P);
else
  ERROR("Datatype property with multiple datatypes");
} else
  EXEC(%AddInstanceAttributes_2,P);
}

Finally, the last two auxiliary rules are in charge of updating the information that corresponds to the datatype property created by the previous one. The first one (AddInstanceAttributes_1) is used when there are several instance attributes in WebODE that produce the same identifier, since they share the same datatype. This rule creates the union of all the concepts where those instance attributes are defined and set it as the domain of the datatype property. It also establishes the range of the property as the datatype of all those instance attributes. Finally, for each OWL class in the domain of the property it creates an OWL AllValuesFrom restriction, and the corresponding minimum and maximum cardinality restrictions, and attach them as subclass of restrictions of the class. The second rule (AddInstanceAttributes_2) is used when there is only one instance attribute that produces that datatype property identifier. In that case, the rule establishes the domain and range of the property, and adds OWL AllValuesFrom and maximum and minimum cardinality restrictions to the corresponding domain class.

%AddInstanceAttributes_1
OWL.DatatypeProperty P -->
{domains = obtainDomains(P);
  ADD(P,domain,CREATE(OWL.UnionOf,1,domains));
  ADD(P,range,obtainDatatype(P.name));
  foreach d in domains {
    D = GETCOMPONENT(OWL.Class,d);
    ADD(D,subClassOf,
        CREATE(OWL.AllValuesFrom,1,P,obtainDatatype(P.name)));
    if (instAttr.minCard! = 0)
      ADD(D,subClassOf,
        CREATE(OWL.MinCardRestriction,1,P,instAttr.minCard));
    if (instAttr.maxCard!=1)
      ADD(D,subClassOf,
        CREATE(OWL.MaxCardRestriction,1,P,instAttr.maxCard));
  }
}

%AddInstanceAttributes_2
OWL.DatatypeProperty P -->
{ADD(P,domain,obtainDomain(P.name));
  ADD(P,range,obtainDatatype(P.name));
  D = GETCOMPONENT(OWL.Class,d);
  ADD(D,subClassOf,
    CREATE(OWL.AllValuesFrom,1,P,obtainDatatype(P.name)));
  if (instAttr.minCard! = 0)
    ADD(D,subClassOf,
      CREATE(OWL.MinCardRestriction,1,P,instAttr.minCard));
  if (instAttr.maxCard!=1)
    ADD(D,subClassOf,
      CREATE(OWL.MaxCardRestriction,1,P,instAttr.maxCard));
}
Finally, we will show a transformation rule that corresponds to the pragmatic post-
processing of datatype properties so as to remove the redundant domains. This
transformation rule consists in checking whether the domain of a datatype property is an
OWL union of classes. In that case, if any of the classes in that union is already a subclass
of any of the other concepts in the union, then the class can be removed, since it is already
considered in the domain.

```%
PostProcessing_RemoveDPRedundantDomains
OWL.DatatypeProperty P -->
| {forEach U in P.domain {
  if (U.isUnionOf)
    forEach D in U.classes
      forEach E in U.classes
        if (E.isSubclassOf(D)) REMOVE(U,class,E)
}|}
```

4.4 Method for Building an Ontology Translation System

This section presents a method for developing ontology translation systems taking into account
the two restrictions described in sections 4.2 and 4.3: (1) transformations between two formats
can be designed and implemented in four different layers: lexical, syntax, semantic and
pragmatic; and (2) the approach for building ontology translation systems must be declarative,
so that they are easier to build and maintain.

The method consists of four activities (feasibility study, analysis of source and target formats,
design of the translation system, and implementation of the translation system) that are tightly
related to each other. These activities are divided into tasks, which can be performed by
different sets of people and with different techniques.

The method recommends to develop ontology translation systems following an iterative life
cycle. It proposes to identify a first set of expressions that can be easily translated from one
format to another, so that the first version of the ontology translation system can be quickly
developed and tested; then it proposes to refine the transformations performed, to analyse more
complex expressions and to design and implement their transformations, and so on and so forth.
The reason for such a recommendation is that developing an ontology translation system is
usually a complex task that requires taking into account too many aspects of the source and
target formats, and many different types of decisions on how to perform specific translations. In
this sense, an iterative life cycle ensures that complex translation problems are tackled once that
the developers have a better knowledge of the source and target formats and once that they have
tested simpler translations performed with earlier versions of the software produced.
The feasibility activity is performed at the beginning of the development project. If this study recommends to start with the ontology translation system development, then for each cycle, the other three activities will be performed sequentially, although developers can always go back to a previous activity using the feedback provided by the subsequent ones, as shown in figure 4.5, which summarises the proposed development process.

As a summary, table 4.2 lists the activities that the method proposes and the tasks to be performed inside each activity. The design and implementation activities take into account the four translation layers described in the previous sections.

Table 4.2. List of activities and tasks of the method for developing ontology translation systems.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Feasibility study</td>
<td>1.1. Identify ontology translation system scope</td>
</tr>
<tr>
<td></td>
<td>1.2. Analysis of current ontology translation systems</td>
</tr>
<tr>
<td></td>
<td>1.3. Ontology translation system requirement definition</td>
</tr>
<tr>
<td></td>
<td>1.4. Feasibility decision-making and recommendation</td>
</tr>
<tr>
<td>2. Analysis of source and target</td>
<td>2.1. Describe source and target formats</td>
</tr>
<tr>
<td>formats</td>
<td>2.2. Determine expressiveness of source and target formats</td>
</tr>
<tr>
<td></td>
<td>2.3. Compare knowledge models of source and target formats</td>
</tr>
<tr>
<td></td>
<td>2.4. Describe and compare additional features of source and target</td>
</tr>
<tr>
<td></td>
<td>formats</td>
</tr>
<tr>
<td></td>
<td>2.5. Determine the scope of translation decisions</td>
</tr>
<tr>
<td></td>
<td>2.6. Specify test plan</td>
</tr>
</tbody>
</table>
3. Design of the translation system

3.1. Find and reuse similar translation systems
3.2. Propose transformations at the pragmatic level
3.3. Propose transformations at the semantic level
3.4. Propose transformations at the syntax level
3.5. Propose transformations at the lexical level
3.6. Propose additional transformations

4. Implementation of the translation system

4.1. Find translation functions to be reused
4.2. Implement transformations in the pragmatic level
4.3. Implement transformations in the semantic level
4.4. Implement transformations in the syntax level
4.5. Implement transformations in the lexical level
4.6. Implement additional transformations
4.7. Declarative specification processing and integration
4.8. Test suite execution

The method does not put special emphasis in other activities that are usually related to software systems’ development, either specific to the software development process, such as deployment and maintenance, or related to support activities, such as quality assurance, project management and configuration management. Nor does it emphasise other tasks that are usually performed during the feasibility study, analysis, design, and implementation activities of general software systems’ development. It only describes those tasks that are specifically related to the development of ontology translation systems, and recommends to perform such additional activities and tasks, which will be beneficial to their development.

4.4.1 Activity 1. Feasibility study

In software engineering, the objective of the feasibility study is to analyse the idea or need for a software system, potential approaches, and all its life cycle constraints and benefits [IEEE Standard 1074-1995, 1996]. The feasibility study may consider modelling and prototyping techniques, and may suggest whether to make or buy the system, in part or in total. Besides, each recommendation must be fully justified, documented and formally approved by all concerned organizations (including the user and the developer).

In our specific case the objective of this activity is to analyse the current ontology translation needs in the organization, so that the proposed solution takes into account not only the technical restrictions (technical feasibility), but also other restrictions related to the business objectives of the organization (business feasibility) and to the project actions that can be successfully undertaken (project feasibility). As a result of this activity, the main requisites to be satisfied by the ontology translation system are obtained, and the main costs, benefits, and risks are identified. The most important aspect of this feasibility study regards the technical restrictions, which can determine whether it is recommended or not to go on with the ontology translation system development.
Our proposal for the organization of this activity into tasks is inspired by the tasks proposed by [METRICA v.3] for the feasibility study process of software systems. The techniques (and documents) used in the execution of these tasks are instead inspired by knowledge engineering approaches (such as [Gómez-Pérez et al., 1997] and [Schreiber et al., 1999]).

As shown in figure 4.6, we first propose to determine the scope of the ontology translation system that will be implemented, which is the expected outcome of such a system, in which context will it be used, etc. We then propose to analyse the current translation systems available, if any, and determine which are the requisites that the new system will accomplish. Finally, we propose to fill in a checklist where the three dimensions identified above are considered (technical, business, and project feasibility), allowing us to make a decision on the feasibility of the system and to propose a set of actions and recommendations to be followed.

Consequently, the input to this activity consists in some preliminary high-level information about the current system, the KRR needs and the source and target formats. The results consist in a deeper description of the current ontology translation systems available for the origin and target formats, a preliminary catalogue of requisites for the system to be developed and the recommendation about its feasibility, including the main costs, benefits, and risks involved.

A summary of the results obtained, techniques used and participants of each task is presented in table 4.3. In the remainder of this section we will describe in detail each of the tasks inside this activity. For each task, we will describe first the goals to be achieved in the task, then the participants of the task, and finally the techniques to be used.

4.4.1.1 Task 1.1. Identify ontology translation system scope

In this task we must determine the scope of the ontology translation system, with the aim of providing a high-level description of the needed translation system. This high-level description will mainly address the expected outcome of the translation system, the context where it will be used, the ontology components to which it will be applied, etc.

To perform this task, all the people involved in the project will participate, including the knowledge and software engineers, the project manager and the executive manager. The techniques to be used are mainly interviews with the executive manager and joint brainstorming sessions, where the main requisites of such a system will be determined.

Input products and output results

Input: knowledge representation and reasoning needs in the target format.

Output: high-level system description.
Participants

Project manager, knowledge engineers, and software engineers.

Techniques

Interviews and brainstorming sessions.

Practical example

Let us assume that we have to develop an ontology translation system from the WebODE platform to OWL. The reason for developing such a system is that we want to make WebODE ontologies available in that language, since we consider that it will be widely used in the future as a standard for developing ontologies for the Semantic Web.
Figure 4.6. Task decomposition of activity 1 (feasibility study).

Table 4.3. Task products, techniques and participants of activity 1 (feasibility study).
After several interviews with people who have developed ontologies in OWL, and after several brainstorming sessions held up with the members of the Ontological Engineering group at UPM, we come up with a high-level description of the translation system to be developed:

"The WebODE ontology export service to OWL will perform ontology translations to the OWL DL sublanguage. It will not export ontologies to OWL Lite, since this sublanguage is too restrictive and much knowledge would be lost in the translation process, nor to OWL Full, since we want to have decidable inference systems for the ontologies generated.

The export service has to be integrated in the WebODE platform. It will not be developed as a backend, that is, we do not mind about maintaining knowledge in cyclic transformations to and from OWL. That is a requisite that we may consider in the future. We only want to be able to export WebODE ontologies to that language so as to make them available for their use in the Semantic Web. Therefore, the system to be developed will be available as an export service that any WebODE user can use at any time during the ontology development process, although it will be usually used once that the ontology is finished.

We hope that the WebODE export service to OWL DL will be widely used in the future, and we consider its development as a strategic part of the WebODE workbench, and as a contribution to the Semantic Web vision.

Finally, we should take into account that the OWL DL specification has been recently proposed as a W3C Recommendation (February 2004). Consequently, it may suffer from modifications in the short, medium and long term."

4.4.1.2 Task 1.2. Analysis of current ontology translation systems

The goal of this task is to analyse existing ontology translation systems that could be reused, either partially or totally, for the new development. The ontology translation systems to be analysed include existing translation systems between the source and target formats, between earlier versions of the source and/or target formats, between the source format and other similar target formats, etc. It may also include the analysis of indirect transformations, that is, transformations from the source to the target format by means of another intermediate format.

Among the participants of this task we may include expert users of the source and target formats, if they are available. Expert users may provide indicators to analyse whether the target format is similar to another one for which there is currently support (hence its translation system could be reused). They can also provide information about which expressions are more widely used than others in the source and target formats.

Regarding the techniques to be used to achieve this task, we mainly recommend performing interviews with expert users and applying cataloguing practices to collect the details of the related translation technology.

Input products and output results
Input: high-level descriptions of the system to be developed, of the current translation systems, and of the source and target formats.

Output: catalogue of related translated technology and recommendations for reuse.

Participants

Project manager, knowledge and software engineers, and expert users.

Techniques

Interviews and cataloguing.

Practical example

Following with the previous example, we analyse the ontology translation systems that could be reused for developing our WebODE export service to OWL DL. In a first step, and after an interview with experts in ontology markup languages, we select the WebODE export services to RDF(S), OIL and DAML+OIL, the DAML+OIL to OWL converter, the OWL plug-in for Protégé-2000, and the Jena2 library.

After the analysis of these systems, we obtain the following conclusions:

1. We can reuse the WebODE export service to RDF(S), because the OWL DL instances will be expressed in RDF.
2. We can also reuse some parts of the WebODE export service to DAML+OIL, since DAML+OIL and OWL are similar to each other.
3. OIL is also similar to OWL. However, the current WebODE export service to OIL generates the output in a plain text format, which is not reusable for our objective, and many of the most complex translation decisions are already considered in the DAML+OIL export service.
4. The DAML+OIL to OWL converter does not work properly in some cases, generating non valid OWL files. Besides, this would require making two transformations (from WebODE to DAML+OIL first, and then from DAML+OIL to OWL), which would make the system not much efficient for large ontologies.
5. The OWL plug-in is too specific for Protégé-2000 ontologies, and has been implemented as a backend, with a strong interaction with the Protégé-2000 ontology management API. Hence it cannot be easily reused.
6. The Jena2 library provides an easy to use Java API to generate OWL files automatically.

And from the previous conclusions we obtain the following catalogue of translation systems that should be considered for partial or total reuse:

- WebODE export service to RDF(S). RDF export can be reused in OWL instance generation. The service is already integrated in WebODE and has been developed using this method.
- WebODE export service to DAML+OIL. Several translation decisions can be reused, since DAML+OIL is based on description logics, as OWL DL.
- Jena2 library. It provides an easy-to-use Java API to generate OWL files automatically.
4.4.1.3 Task 1.3. Ontology translation system requirement definition

In this task we define the catalogue of high-level requirements of the ontology translation system. This catalogue considers both functional requirements (ontology components to be transformed, expected outcomes regarding knowledge preservation and pragmatic preservation, etc.) and non-functional requirements (considering aspects related to the integration with other systems, the expected performance of the translation system, context where it will be used, etc.).

This task is performed in parallel to the task 1.2, and by the same participants, using interviews to elicit the requirements, and cataloguing practices to document them.

Input products and output results

**Input:** high-level descriptions of the system to be developed and of the source and target formats.

**Output:** catalogue of requirements of the ontology translation system.

Participants

Project manager, knowledge and software engineers, and expert users.

Techniques

Interviews and cataloguing.

Practical example

In our example, the following requirements are elicited:

**Functional requirements:**

Req1. The ontology translation system does not aim at preserving all the knowledge represented in the ontology, but only those pieces of knowledge that can be directly transformed to OWL DL, without workarounds. Regarding pragmatics, the transformations to be made must ensure that the resulting OWL DL ontologies are easy to understand for OWL-aware human users and systems.

Req2. The following WebODE knowledge model components will be translated to OWL DL: concepts and their instance attributes, ad hoc relations, concept taxonomies (including subclass-of relationships, disjoint and exhaustive concept decompositions, and concept partitions), and instances.

Req3. The following WebODE knowledge model component will be translated partially to OWL DL: formal axioms that can be transformed to OWL DL.

Req4. The following WebODE knowledge model components will not be translated to OWL DL: class attributes, synonyms and acronyms, bibliographic references, and rules.

Req5. WebODE attribute datatypes will be transformed to XML Schema datatypes.

Req6. WebODE instance sets will be transformed to separate RDF files.
Non-functional requirements:

Req7. The translation system must be integrated in the WebODE platform as an export service, providing the user with a ZIP file that contains the OWL DL ontology file and the instances' files.

Req8. There are no specific requirements related to the performance of the transformations to be made by the WebODE export service to OWL DL.

4.4.1.4 Task 1.4. Feasibility decision-making and recommendation

The objective of this task is to make a decision on the feasibility of the ontology translation system to be developed, and to recommend which specific actions should be taken into account to overcome the problems that may arise during its development.

In this task, all the key implications obtained in the previous tasks are wrapped up to support the decision-making process. They are summarised in a checklist, inspired by the worksheet OM-5 proposed by the CommonKADS methodology [Schreiber et al., 1999], which is adapted to the specific development of ontology translation systems and where we do put more stress on the technical feasibility rather than on the other two dimensions (business and project feasibility).

If the project is considered feasible, the proposed actions (which are also included in the questionnaire presented below) help to prepare the start of the ontology translation system development, and are one of the main inputs for the following activities. They also determine the number of iterations in the development process that will be needed and the ontology components that will be transformed in each iteration. Besides, the main costs, benefits, and risks of the development are identified.

Input products and output results

Input: catalogue of requirements of the ontology translation system, and catalogue of related translated technology and recommendations for reuse; high-level descriptions of the system to be developed, of the current translation systems, and of the source and target formats, and knowledge representation and reasoning needs in the target format.

Output: recommendations and proposed actions

Participants

Executive manager, project manager, knowledge engineers, and software engineers.

Techniques

Usage of checklists. As specified above, we recommend to use a checklist that is based on one of the worksheets proposed by CommonKADS, specifically worksheet OM-5 (checklist for feasibility decision making). This checklist is presented in table 4.4. The questions that
appear in it should not be considered exhaustive: users may add as many questions and answers as they want in order to help in the decision-making process.

Table 4.4. Checklist for the feasibility decision making (based on the CommonKADS worksheet OM-5 [Schreiber et al., 1999]).

<table>
<thead>
<tr>
<th>Business feasibility</th>
<th>Proposed</th>
<th>Focus: What is the recommended focus in the identified problem-opportunity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. What are the expected benefits for the organization from the development of an ontology translation system?</td>
<td>Focus:</td>
<td></td>
</tr>
<tr>
<td>2. How large is the expected value?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Will the ontology translation system be used for a long-term period?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. What are the expected costs for the ontology translation system development?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. How does this compare to possible alternative solutions, such as performing translations by hand or by means of intermediate translation systems?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project feasibility</td>
<td>1. Is there adequate commitment from the actors and stakeholders involved?</td>
<td>Focus:</td>
</tr>
<tr>
<td>2. Can the needed resources in terms of time, budget, equipment, and staffing be made available?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Are the required knowledge and other competencies available?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Are the expectations regarding the project and its results realistic?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Are the project organization and its internal as well as external communication adequate?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Are there further project risks and uncertainties?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business feasibility</td>
<td>1. Do the project participants have a deep knowledge of the source format (including the knowledge model components and the reasoning services available)?</td>
<td>Focus:</td>
</tr>
<tr>
<td>2. Does the source format have a clear and well-defined Java API?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Have the project participants built and/or used ontologies in the source format?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project feasibility</td>
<td>1. Do the project participants have a deep knowledge of the target format (including the knowledge model components and the reasoning services available)?</td>
<td>Focus:</td>
</tr>
<tr>
<td>2. Does the target format have a clear and well-defined Java API?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Have the project participants built and/or used ontologies in the target format?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technical feasibility</td>
<td>1. Are the source and target formats based on the same KR paradigms?</td>
<td>Focus:</td>
</tr>
<tr>
<td>2. Does the ontology translation system need to translate completely the ontology or only parts of it?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Does the ontology translation system need to preserve all the knowledge in the transformation?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Does the ontology translation system need to preserve completely the pragmatics of the ontology in the transformation?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Is it clear what the success measures are and how to test for validity, quality, and satisfactory performance?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. How complex, in terms of knowledge stored and reasoning processes to be carried out, is the translation process?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-functional requirements</td>
<td>13. How complex is the interaction with other information systems and possible other resources? Are state-of-the-art methods and techniques available and adequate?</td>
<td>Focus:</td>
</tr>
<tr>
<td>14. Are there any specific requirements about the ontology translation system performance?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15. Are there critical aspects involved, relating to time, quality, needed resources, or otherwise? If so, how to go about them?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16. Are there further technological risks and uncertainties?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A Layered Declarative Ontology Translation Model 97
actions

2. Target solution: What is the recommended solution direction for this focus area?
3. What are the expected results, costs, and benefits?
4. What project actions are required to get there?
5. How many development iterations will be performed, and which components will be transformed in each iteration?
6. Risks: If circumstances inside or outside the organization change, under which conditions is it wise to reconsider the proposed decisions?

Practical example

In our example, we filled the checklist and obtained the result presented in table 4.5:

Table 4.5. Checklist for the feasibility decision making of the WebODE export service to OWL DL

<table>
<thead>
<tr>
<th>Business feasibility</th>
<th>The organization expects large benefits from the development of the WebODE export service to OWL DL, because WebODE will be one of the first ontology platforms with OWL support. We expect that the WebODE user community will grow. The export service will be used for a long-term period. It may require slight modifications during its life, due to the recent proposal of the OWL language to W3C Recommendation. The expected cost of developing such a service is around 2 man month. The other alternatives explored (transforming OWL ontologies through DAML+OIL or through other ontology tools) have several drawbacks: we are dependent on their support for new updates of the OWL specification, and they would have a negative impact on the performance of the translation system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project feasibility</td>
<td>People in the organization are committed to the development of the WebODE export service to OWL DL, and all the needed resources will be made available. The developers of the translation service have experience in the development of other translation systems with the same method, and have also a deep knowledge of the main features of the knowledge models of WebODE and OWL DL. The expectations are realistic, and the communication channels between all the people involved are adequate. The development team has also good communication with the OWL developers. The only risks foreseen are those related to the changes that the OWL language specifications may suffer from. However, the language has been already published as a W3C Recommendation, and we only foresee minor changes.</td>
</tr>
<tr>
<td>Technical feasibility</td>
<td><strong>With regard to the source format:</strong> The project participants have a deep and sound knowledge of the source format (WebODE) and of its Java API. They have also built and used a large number of WebODE ontologies. <strong>With regard to the target format:</strong> The project participants have a good knowledge of the target format (OWL DL) and of the Jena2 library, which will be used as its API. They have also built and used some OWL ontologies. <strong>With regard to the transformation process:</strong> WebODE and OWL DL are based on different KR paradigms: a combination of frames and first order logic, and description logic respectively. The ontology must be translated almost completely to OWL DL, since we want to lose as less knowledge as possible in the transformation process. We do not aim at preserving all the knowledge of the ontology, but we do aim at preserving the pragmatics of what we transform. Several test ontologies will be defined to test for the satisfactory performance of the export service developed. Transforming from the frames KR paradigm to description logic is a complex task that requires implementing difficult translation decisions: such as transforming the scope of ad hoc relations and instance attributes from being local to concepts to being global to the ontology, transforming concept and instance attribute definitions into property restrictions, etc. However, there are already similar translation systems implemented that may help in making these translation decisions. <strong>With regard to the non-functional requirements:</strong> The interaction with WebODE is complex, although well known by the project participants, and the interaction with the Jena2 library is considered easy. Besides, there are no specific requirements about the system performance. We do not foresee</td>
</tr>
</tbody>
</table>
either critical aspects or risks about time, quality, needed resources, etc.

| Proposed actions | Our recommendation is to create the WebODE export service to OWL DL reusing the Jena2 library and part of the WebODE export service to RDF(S). We will also reuse some of the translation decisions performed for the WebODE export service to DAML+OIL. We foresee great benefits from the development of this system, which will increase the WebODE user community and will allow developing ontologies for the Semantic Web with its ontology editor. The development of the export service will be performed in three iterations: first, the WebODE concepts, attributes, ad hoc relations, simple concept taxonomies, and instances will be transformed; then, we will deal with complex primitives for creating concept taxonomies, such as disjoint and exhaustive decompositions, and partitions; finally, we will deal with WebODE formal axioms and will perform all the pragmatic transformations. The only risks foreseen are related to the fact that the OWL language has been published as a W3C Recommendation very recently, and consequently may suffer from minor changes in the near future. |

### 4.4.2 Activity 2. Analysis of source and target formats

The objective of this activity is to obtain a thorough description and comparison of the source and target formats of the ontology translation system. We assume that they will allow us to gain a better understanding of their similarities and differences in expressiveness, which will be useful to design and implement the translation decisions in the subsequent activities. Besides, in this activity we refine the catalogue of requirements already obtained as a result of the feasibility study, and we identify the test suite that will be used to test the translation system validity after each iteration in its development process.

To perform the description and comparison tasks we can use either formal or informal approaches, which were identified in chapter 2. As we will see, we recommend informal ones, which are more informative and useful for the specific task of developing ontology translation systems. However, this does not prevent us from using any of the formal approaches that have been proposed so far in the literature, and which could help us determine formally whether an expression in the source format can be transformed to the target format without losses of knowledge.

Once that the two formats have been described, evaluated and compared, we recommend to focus on other additional features that could be needed in the translation process. They may include reasoning mechanisms or any other specific details that could be interesting for the task of translation. For these description and comparison tasks we can also follow formal or informal approaches, from which we recommend again the latter.

The information gathered in the previous tasks is used to determine the scope of the translation decisions to be made: which components map to each other, which components of the source format must be represented by means of others in the target format, which components cannot be represented in the target format, etc. As a result, we obtain a refinement of the catalogue of
requirements obtained during the feasibility study, which serves as the basis for the next activities (design and implementation of the translation system).

Finally, we propose to define the test plan, which will consist of a set of unitary tests that the translation system must pass in order to be considered valid. We will define a test suite consisting of several ontologies covering all the possible translation situations that the translation system must cover. These ontologies will be available in the source format and in the target format (which should be the output of the translation process). The test execution will consist on comparing the output obtained and the output expected. For each iteration of the software development process we will define different sets of ontologies.

Consequently, this activity receives as an input all the results of the feasibility study, together with the description of the source and target formats, which was also used as an input for that activity, and outputs a comparison of both formats, the scope of the translation decisions to be performed, with a refined requirements catalogue, and a test plan with its corresponding test suite.

The activity is mainly performed by the knowledge engineer that will be in charge of designing and implementing the translation decisions, with the help of expert users for determining the expressiveness of the source and target formats, and of software engineers for describing additional features of the formats, for determining the scope of translation decisions, and for specifying the test plan. The project manager will help in the refinement of the requirements catalogue.

A summary of the tasks to be performed in this activity and their timeline is presented in figure 4.7. The results obtained, techniques used and participants of each task are summarised in table 4.6.
Activity 2. Analysis of source and target formats

- Task 2.1: Describe source and target formats
- Task 2.2: Determine the expressiveness of source and target formats
- Task 2.3: Compare knowledge models of source and target formats
- Task 2.4: Describe and compare additional features of source and target formats
- Task 2.5: Determine scope of translation decisions
- Task 2.6: Specify test plan

Table 4.6. Task products, techniques, and participants of activity 2 (analysis of source and target formats).
4.4.2.1 Task 2.1. Describe source and target formats

In this task we provide a detailed description of the source and target formats, by describing their knowledge representation ontologies (aka KR ontologies). The advantage of describing the formats by means of their KR ontologies is that they present the primitives to be used when building ontologies with them. Besides, the KR class taxonomy provides a hierarchy of some of those primitives. This description will be used as an input for other tasks inside this activity.

A knowledge engineer will be in charge of these detailed descriptions. This person will analyse the documentation available for each format, especially focusing on their knowledge models and the ontology components that they can represent.

Input products and output results

Input: documentation of source and target formats.

Output: KR ontologies of the source and target formats.

Participants

Knowledge engineer.

Techniques

KR ontology description, with its class taxonomies and its list of properties. These types of descriptions can be found in the documentation of ontology markup languages such as OIL [Horrocks et al., 2000]. They can be also found at [Gómez-Pérez et al., 2003] for the ontology markup languages RDF(S), DAML+OIL, and OWL.

Practical example

We show an extract of the description of the OWL DL language. A more complete description can be found in [Gómez-Pérez et al., 2003, pages 65-71]:

"The OWL language [Dean and Schreiber, 2004] was published as a W3C Recommendation in February 2004. It is derived from the DAML+OIL language, and it builds upon RDF(S). OWL is divided in layers: OWL Lite, OWL DL, and OWL Full. OWL Lite extends RDF(S) and gathers the most common features of OWL, so it is intended for users that only need to create class taxonomies and simple constraints. OWL DL includes the complete OWL vocabulary. Finally, OWL Full provides more flexibility to represent ontologies than OWL DL does, but does not ensure that there will be decidable reasoners available for it.

OWL DL is based on the description logic language SHOIN(d†). The OWL DL KR ontology, which is implemented in OWL, is available at: http://www.w3.org/2002/07/owl. There are 40 primitives (16 classes and 24 properties). Figure 4.8 shows the class taxonomy of the primitives that are classes in the OWL KR ontology. All belong to OWL Lite. Hence, they also belong to OWL DL and OWL Full. These primitives can be grouped as follows:
A Layered Declarative Ontology Translation Model

Figure 4.8. Class taxonomy of the OWL DL KR ontology.

- Classes for defining classes and restrictions (owl:Class and owl:Restriction). The primitive owl:Class specializes rdfs:Class and is used to define classes. The primitive owl:Restriction specializes owl:Class and is used to define property restrictions for classes (number restrictions, existential restrictions, universal restrictions, etc.).

[...]

OWL class expressions are built with KR primitives that are properties. Some of them belong to OWL Lite and others belong to OWL DL. Table 4.7 summarizes the main features of the properties of the OWL KR ontology, specifying their domain and range. If the value for the range is “not specified”, we mean that the property can take any value which is not restricted to a specific class of the OWL KR ontology. Besides, in OWL we can use the properties rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, rdfs:range, rdfs:comment, rdfs:label, rdfs:seeAlso, and rdfs:isDefinedBy from the RDF(S) KR ontology.

Table 4.7. Property descriptions of the OWL DL KR ontology.

<table>
<thead>
<tr>
<th>Property name</th>
<th>domain</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>owl:intersectionOf</td>
<td>owl:Class</td>
<td>rdf:List</td>
</tr>
<tr>
<td>owl:unionOf</td>
<td>owl:Class</td>
<td>rdf:List</td>
</tr>
<tr>
<td>owl:complementOf</td>
<td>owl:Class</td>
<td>owl:Class</td>
</tr>
<tr>
<td>owl:oneOf</td>
<td>owl:Class</td>
<td>rdf:List</td>
</tr>
<tr>
<td>owl:onProperty</td>
<td>owl:Restriction</td>
<td>rdfs:Property</td>
</tr>
<tr>
<td>owl:allValuesFrom</td>
<td>owl:Restriction</td>
<td>rdfs:Class</td>
</tr>
<tr>
<td>owl:hasValue</td>
<td>owl:Restriction</td>
<td>not specified</td>
</tr>
<tr>
<td>owl:someValuesFrom</td>
<td>owl:Restriction</td>
<td>rdfs:Class</td>
</tr>
<tr>
<td>owl:minCardinality</td>
<td>owl:Restriction</td>
<td>xsd:nonNegativeInteger</td>
</tr>
<tr>
<td>owl:maxCardinality</td>
<td>owl:Restriction</td>
<td>xsd:nonNegativeInteger</td>
</tr>
<tr>
<td>owl:cardinality</td>
<td>owl:Restriction</td>
<td>xsd:nonNegativeInteger</td>
</tr>
<tr>
<td>owl:inverseOf</td>
<td>owl:ObjectProperty</td>
<td>owl:ObjectProperty</td>
</tr>
<tr>
<td>owl:sameAs</td>
<td>owl:Thing</td>
<td>owl:Thing</td>
</tr>
<tr>
<td>owl:equivalentClass</td>
<td>owl:Class</td>
<td>owl:Class</td>
</tr>
<tr>
<td>owl:equivalentProperty</td>
<td>rdf:Property</td>
<td>rdf:Property</td>
</tr>
<tr>
<td>owl:sameIndividualAs</td>
<td>owl:Thing</td>
<td>owl:Thing</td>
</tr>
<tr>
<td>owl:differentFrom</td>
<td>owl:Thing</td>
<td>owl:Thing</td>
</tr>
<tr>
<td>owl:disjointWith</td>
<td>owl:Class</td>
<td>owl:Class</td>
</tr>
<tr>
<td>owl:distinctMembers</td>
<td>owl:AllDifferent</td>
<td>rdf:List</td>
</tr>
<tr>
<td>owl:versionInfo</td>
<td>not specified</td>
<td>not specified</td>
</tr>
<tr>
<td>owl:priorVersion</td>
<td>owl:Ontology</td>
<td>owl:Ontology</td>
</tr>
<tr>
<td>owl:incompatibleWith</td>
<td>owl:Ontology</td>
<td>owl:Ontology</td>
</tr>
<tr>
<td>owl:backwardCompatibleWith</td>
<td>owl:Ontology</td>
<td>owl:Ontology</td>
</tr>
<tr>
<td>owl:imports</td>
<td>owl:Ontology</td>
<td>owl:Ontology</td>
</tr>
</tbody>
</table>

The OWL KR primitives can be grouped as follows:
- Properties for defining class expressions:
  - Class conjunction (owl:intersectionOf), disjunction (owl:unionOf), and negation (owl:complementOf).
  - Property restrictions. They are defined with the class owl:Restriction. Restrictions are defined with two elements: owl:onProperty (which refers to the property name) and another element that expresses value restriction (owl:allValuesFrom), existential restriction (owl:someValuesFrom), and number restriction (owl:cardinality, owl:maxCardinality, and owl:minCardinality).

4.4.2.2 Task 2.2. Determine expressiveness of source and target formats

This task aims at determining clearly the expressiveness of each of the formats involved in the transformation. To achieve its objective, we can use different kinds of techniques, based on formal, semi-formal, and informal approaches, described in section 2.3:

- Formal techniques have been widely used to describe the semantics of KR systems. They are also known as model-theoretic semantics, since they use model theories[^Manzano1999] to determine exactly that semantics. Other formal techniques consist in defining mappings between the format whose semantics is described to another format with a well-defined model-theoretic semantics.

- Semi-formal techniques determine which knowledge model components can be defined in a format. They usually consist in evaluation frameworks that provide checklists with details of all the knowledge model components that can be represented in KR systems.

- Informal techniques also determine which knowledge model components can be defined in a format, but without using specific evaluation frameworks nor checklists. They usually consist in enumerating the knowledge model components that can be defined in a KR format.

Given that our aim is to develop ontology translation systems between different formats, we recommend to use the semi-formal and informal approaches, which provide coarse-grained overviews of the components that can be defined in each format.

This task is performed by the same knowledge engineer, with the help of experts in the source and target formats. Once the expressiveness of both formats has been defined, we will be able to compare them and propose the translations to be implemented by the ontology translation system.

[^Manzano1999]: Among the many bibliographic references that could be used to refer to model theory, one of the basic textbooks has been selected.
**Input products and output results**

*Input:* detailed description of source and target formats.

*Output:* formal, semi-formal or informal semantics of the source and target formats.

**Participants**

Knowledge engineer and expert users.

**Techniques**

Formal approaches to define semantics: model theory and mappings to other formats.

Semi-formal approaches to define semantics: evaluation frameworks [Corcho and Gómez-Pérez, 2000].

Informal approaches to define semantics: unstructured enumerations and informal descriptions of knowledge components [Garshol, 2001; Garshol, 2003; Gil and Ratnakar, 2002; Noy et al., 2001].

**Practical example**

Table 4.8 shows the results of applying the evaluation framework described in [Corcho and Gómez-Pérez, 2000] to the source and target formats of the ontology translation system to be developed: WebODE and OWL DL respectively. The cells in the table are filled using '+' to indicate that it is a supported feature in the format, '-' for non supported features, and 'W' for non supported features that could be supported with some workaround.

<table>
<thead>
<tr>
<th>CONCEPTS</th>
<th>WebODE</th>
<th>OWL DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instance attributes</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Class attributes</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Facets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type constraints</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Cardinality constraints</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Procedural knowledge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CONCEPT TAXONOMIES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subclass-Of</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Disjoint-Decomposition</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Exhaustive-Decomposition</td>
<td>+</td>
<td>W</td>
</tr>
<tr>
<td>Partition</td>
<td>+</td>
<td>W</td>
</tr>
</tbody>
</table>

Table 4.8. Results of applying the evaluation framework described in [Corcho and Gómez-Pérez, 2000] to the knowledge models of WebODE and OWL DL.

An informal description of the WebODE knowledge model semantics can be found at [Corcho et al., 2002], and several formal semantics for OWL DL can be found at [Patel-Schneider et al., 2004].

**4.4.2.3 Task 2.3. Compare knowledge models of source and target formats**

The objective of this task is to provide a comprehensive comparison of the knowledge models of the source and target formats, which will be used, together with the detailed description
obtained in task 2.2, to propose the translations to be performed between them. This task is accomplished by the knowledge engineer, who has to decide which components are equivalent to each other, although they may be called differently, which details differ in both formats, etc.

The format of the results obtained from this task are determined by the type of technique used to carry out the comparison. As in the task 2.2, we can use both formal, semi-formal, and informal techniques, from which we also recommend the semi-formal and informal ones, which are more informative and useful for the specific task of developing ontology translation systems.

**Input products and output results**

*Input:* detailed description of the source and target formats.

*Output:* comparison of the source and target formats.

**Participants**

Knowledge engineer.

**Techniques**

Formal comparisons ([Baader, 1996], [Borgida, 1996], [Euzenat and Stuckenschmidt, 2002], etc.). They are based on the definition of formal semantics, as described in section 2.3.

Informal and semi-formal comparisons, based on evaluation frameworks [Corcho and Gómez-Pérez, 2000], Venn diagrams [Knublauch, 2003] or tables describing how different ontology components are represented in each language [Garshol, 2001; Garshol, 2003; Gil and Ratnakar, 2002; Noy et al., 2001].

**Practical example**

The results presented in table 4.8 allow determining the main aspects to be taken into account in the translation from WebODE to OWL DL, which are derived from the differences shown in the table. Some of these differences are due to the fact that the WebODE knowledge model is based on a combination of frames and first order logic, while the OWL DL knowledge model is based on description logics:

- Class attributes cannot be represented in OWL DL. This type of attributes is useful to describe characteristics of the classes that are not inherited by their subclasses nor by their instances.

- In OWL DL there are not specific primitives for representing exhaustive decompositions of classes (and consequently, nor are there primitives for partitions). This kind of knowledge must be expressed by stating that the superclass is equivalent to the union of all the classes in the decomposition or partition, and that all these classes are subclasses of the superclass.

- N-ary relations between classes must be represented similarly in both formats: by creating a class that simulates the relation.
- WebODE represents relation hierarchies by means of formal axioms, while in OWL DL there is a specific primitive to represent them.

- WebODE allows representing any first order logic axiom, and rules. The formal axioms that can be represented in OWL DL are restricted, and rules cannot be represented. This will be done in the future in another language that will be built on top of OWL (e.g., OWL Rule, SWRL).

This preliminary list does not pretend to be exhaustive but only to provide some guidance for the analysis of the main sources of differences that will be relevant for the translation.

4.4.2.4 Task 2.4. Describe and compare additional features of source and target formats

In this task we describe and compare those features of the source and target formats that are not considered by the previous knowledge model descriptions and comparisons, and that the knowledge and software engineers may consider interesting from the point of view of ontology translation. For instance, the source and target formats will usually have different reasoning mechanisms attached, may require different parameters for representing the ontologies (namespaces, documentation options, etc.), may provide different visualization options for the ontologies (in the case of ontology tools), etc.

Input products and output results

Input: detailed description of the source and target format knowledge models, and evaluation of expressiveness and comparison of source and target formats.

Output: catalogue of similarities and differences between the source and target formats, which are not specifically related to their knowledge models, or which are not covered by the previous informal, semi-formal, and formal comparison approaches.

Participants

Knowledge and software engineers.

Techniques

Cataloguing.

Practical example

Besides the differences between the WebODE and OWL DL knowledge models, which have been obtained as a result of the previous tasks, the following additional ones exist.

- WebODE ontologies can contain bibliographic references, which can be associated to any ontology component.

- WebODE instance attributes contain more information than their correspondent components in OWL DL (datatype properties). In case of numeric attributes, they contain their minimum and maximum value, their
measurement unit, and their precision. Besides, their datatypes are not restricted to a fixed set of built-in XML Schema datatypes, but any derived XML Schema datatype can be also used.

- The scope of the identifiers of WebODE instance attributes and ad hoc relations is local to the concept where they are defined (that is, their domain), while the scope of these identifiers in OWL DL is global to the ontology.

- Class definitions in OWL DL can be either complete (expressing necessary and sufficient conditions to belong to the class) or partial (expressing only necessary conditions). WebODE concept definitions are partial. To express sufficient conditions in WebODE, formal axioms must be used.

- OWL DL allows expressing unions of classes, which must be expressed by means of formal axioms in WebODE.

- The WebODE reasoning engine implements attribute and relation inheritance, which are typical problems in frame-based systems. Available OWL DL reasoners mainly aim at detecting inconsistencies in class taxonomies, obtaining class subsumption relationships from class definitions, and some of them aim at detecting inconsistencies in instances.

4.4.2.5 Task 2.5. Determine the scope of translation decisions

The objective of this task is to determine which components and expressions of the source format will be transformed to the target one, and which approaches will be taken with regard to the preservation of the knowledge and of the intended meaning of the ontology. As we will see in chapter 5, the translation decisions to be taken can be driven by knowledge preservation constraints, by pragmatics preservation constraints or by both types of constraints at the same time, and there can be different degrees of knowledge and pragmatics preservation.

This task is performed jointly by the project manager, and the knowledge and software engineers, since many factors influence on the decisions to be proposed. As a result of this task, a refined catalogue of requirements for the ontology translation system will be output, where there is more detail about the transformations to be performed than in the catalogue obtained during the feasibility study.

**Input products and output results**

**Input:** detailed description of the source and target format knowledge models, evaluation of expressiveness and comparison of source and target formats, and catalogue of additional similarities and differences between the source and target formats.

**Output:** scope of the ontology translation decisions and refined catalogue of requirements.

**Participants**

Project manager, and knowledge and software engineers.
Techniques

Cataloguing

Practical example

The requisite catalogue obtained as a result of task 1.3 is refined, as shown below:

"Functional requirements:

Req1. The ontology translation system does not aim at preserving all the knowledge represented in the ontology, but only those pieces of knowledge that can be directly transformed to OWL DL, without workarounds, except for exhaustive concept decompositions and concept partitions. Regarding pragmatics, the transformations to be made must ensure that the resulting OWL DL ontologies are easy to understand for OWL-aware human users and systems.

Req2. The following WebODE knowledge model components will be translated to OWL DL: concepts and their instance attributes, ad hoc relations, concept taxonomies (including subclass-of relationships, disjoint and exhaustive concept decompositions, and concept partitions), and instances. The correspondences between these WebODE components and OWL DL components are as shown in table 4.9:

Table 4.9. Correspondences between WebODE and OWL DL components.

<table>
<thead>
<tr>
<th>WebODE component</th>
<th>OWL DL component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept</td>
<td>Class</td>
</tr>
<tr>
<td>Instance attribute (local scope)</td>
<td>Datatype property (global scope)</td>
</tr>
<tr>
<td>Ad hoc relation (local scope)</td>
<td>Object property (global scope)</td>
</tr>
<tr>
<td>Subclass of</td>
<td>Subclass of</td>
</tr>
<tr>
<td>Disjoint decompositions</td>
<td>Disjointness between pairs of classes</td>
</tr>
<tr>
<td>Exhaustive decompositions</td>
<td>Class equivalence to union of subclasses</td>
</tr>
<tr>
<td></td>
<td>Subclass-of relationships</td>
</tr>
<tr>
<td>Partitions</td>
<td>Disjointness between pairs of classes</td>
</tr>
<tr>
<td></td>
<td>Class equivalence to union of subclasses</td>
</tr>
<tr>
<td></td>
<td>Subclass-of relationships</td>
</tr>
<tr>
<td>Instances</td>
<td>Individuals</td>
</tr>
</tbody>
</table>

Req3. The following WebODE knowledge model component will be translated partially to OWL DL: formal axioms that can be transformed to OWL DL.

Req4. The following WebODE knowledge model components will not be translated to OWL DL: class attributes, synonyms and acronyms, bibliographic references, and rules.

Req5. WebODE attribute datatypes (including XML Schema datatypes) will be transformed to XML Schema datatypes. Derived XML Schema datatypes will be transformed to the XML Schema datatype string.

Req6. WebODE instance sets will be transformed to separate RDF files.

Non-functional requirements:

Req7. The translation system must be integrated in the WebODE platform as an export service, providing the user with a ZIP file that contains the OWL DL ontology file and the instances' files.

Req8. There are no specific requirements related to the performance of the transformations to be made by the WebODE export service to OWL DL."
4.4.2.6 Task 2.6. Specify test plan

This task, which is performed in parallel to the definition of the scope of ontology translation decisions, consists in defining the test plan for the ontology translation system. Like in general software system development, in this phase we define the system tests, which will be used to test the validity of the generated ontology translation system.

The test suite consists of several ontologies that will usually cover all the possible translation situations that the translation system must deal with. The ontologies are defined in the source and target formats, the latter being the expected output of the translation system when applied to the former. Consequently, each test execution will simply consist in comparing the output obtained when the translation system is applied to any of the source ontologies in the test suite with its corresponding target ontology.

Since the development of the translation system will be made iteratively, in each iteration we will define a new set of ontologies for the test suite, which must cover the one used in the previous iteration plus new ontologies with the features that are being transformed in the current iteration.

This task is performed by the knowledge engineer, who decides which ontologies should be included in the test suite and generates manually their corresponding ontologies in the target format, and by the software engineer, who decides whether other aspects should be taken into account in the definition of these tests, such as input parameters for the ontology translation system, etc.

Input products and output results

Input: detailed description of the source and target format knowledge models, evaluation of expressiveness and comparison of source and target formats, and catalogue of additional similarities and differences between the source and target formats.

Output: ontology translation system test suite and test plan.

Participants

Knowledge and software engineers.

Techniques

System test cases

Practical example
A subset of the synthetic ontologies described in [Corcho et al., 2003b] will be used as a test suite for the WebODE export service to OWL DL. This set of synthetic ontologies provide several combinations of those features that have to be tested to check the correct translation of ontologies by the ontology translation system developed (e.g., instance attributes with the same name defined for different concepts, with either different types or with the same type, concepts and ad hoc relations with the same name, etc.).

For each of these WebODE ontologies, we have also defined the corresponding OWL ontology that is expected after the translation.

We are using synthetic ontologies because they allow combining all the possible translation situations considered by the ontology translation system. Besides, the test suite is divided in three groups, depending on the translations performed in each iteration of the ontology translation system development.

4.4.3 Activity 3. Design of the translation system

The objective of the design activity is to provide a detailed specification of the transformations to be performed by the ontology translation system. From this specification we will be able to generate, in the implementation activity, the declarative implementation of the system, which will be used in its turn to generate the final ontology translation system.

The tasks inside this activity are divided in three groups, as shown in figure 4.9. We describe next the goals and responsible of each task, which are also summarised in table 4.10:

The objective of the first task is to analyse similar ontology translation systems and to detect which of their translation decisions can be actually reused. We assume that by reusing existing translation decisions we will be able to minimise the sources of errors in our translation proposals. Furthermore, we will benefit from work already known, for which we already know its properties (namely, how they preserve semantics and pragmatics). We must remember that the potential reusable systems were already identified and catalogued during the feasibility study. Therefore, the knowledge and software engineers will have to focus on those systems, with two different purposes: the knowledge engineer will aim at detecting reusable translation decisions, and the software engineer will aim at detecting reusable pieces of software code.

The second group of tasks deals with the four layers of translation problems described in section 4.1: we propose to design transformations at the different inter-related levels, using different techniques for each layer, as we will describe later. All these tasks should be mainly performed in parallel, and the decisions taken at one task provide feedback for the others, as shown in figure 4.8. We propose to start with the translation decisions at the pragmatic and semantic levels, leaving the syntax and lexical transformations for the last steps. The pragmatic and semantic translation decisions are mainly proposed by knowledge engineers (it is recommended that these tasks are performed by the same persons that have carried out the analysis and
comparisons of the source and target formats). The syntax and lexical transformations can be proposed jointly by knowledge and software engineers, since they have more to do with general programming aspects, rather than with the complexity of transforming knowledge. Besides, the software engineers can help in the pragmatic and semantic tasks in taking the decisions of whether the transformations that cannot be specified declaratively, due to its complexity, can be easily implemented or not in a general programming language.

Finally, the objective of the last task is to propose any additional transformations or design issues that have not been covered by the previous tasks, because they could not be catalogued as lexical, syntactic, semantic or pragmatic transformations, and that are necessary for the correct functioning of the ontology translation system. These transformations include design issues such as the initialisation and setting up of parameters in the source and target formats, any foreseen integration needs of the generated system in the case of transformations where ontology tools or specific libraries are used, etc. This task will be mainly performed by software engineers, with the help of the knowledge engineers that have proposed the other types of transformations.

As shown in the figure, we may need to come back to the second group of activities after proposing some additional transformations. This is a cyclic process until we have determined all the transformation to be performed. All the output results obtained from the tasks in this activity are integrated in a single document called “translation system design document”, as shown in the figure.
Activity 3. Design of translation system

Input:
- Source and target format comparison
- Ontology translation decisions scope
- Related requirement catalogue
- Related technology description

Output:
- Translation system design document

Figure 4.9. Task decomposition of activity 3 (design of the translation system).

<table>
<thead>
<tr>
<th>Task</th>
<th>Products</th>
<th>Techniques</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 3.1. Find and reuse similar translation systems</td>
<td>Catalogue of translation decisions to be reused</td>
<td>Cataloguing</td>
<td>Knowledge engineer, Software engineer</td>
</tr>
<tr>
<td>Task 3.2. Propose transformations at the pragmatic level</td>
<td>Catalogue of translation decisions at the pragmatic level</td>
<td>Cataloguing, Translation decision tables and diagrams</td>
<td>Knowledge engineer, Software engineer</td>
</tr>
<tr>
<td>Task 3.3. Propose transformations at the semantic level</td>
<td>Catalogue of translation decisions at the semantic level</td>
<td>Cataloguing, Translation decision tables and diagrams</td>
<td>Knowledge engineer, Software engineer</td>
</tr>
<tr>
<td>Task 3.4. Propose transformations at the syntactic level</td>
<td>Catalogue of translation decisions at the syntactic level</td>
<td>Cataloguing, Translation decision tables and diagrams, BNF grammar</td>
<td>Software engineer, Knowledge engineer</td>
</tr>
<tr>
<td>Task 3.5. Propose transformations at the lexical level</td>
<td>Catalogue of translation decisions at the lexical level</td>
<td>Cataloguing, Translation decision tables and diagrams, BNF grammar</td>
<td>Software engineer, Knowledge engineer</td>
</tr>
<tr>
<td>Task 3.6. Propose additional transformations</td>
<td>Catalogue of additional transformations</td>
<td>Cataloguing, Translation decision tables</td>
<td>Software engineer, Knowledge engineer</td>
</tr>
</tbody>
</table>

Table 4.10. Task products, techniques and participants of activity 3 (design of the translation system).
4.4.3.1 Task 3.1. Find and reuse similar translation systems

In this task we analyse the ontology translation systems that had been selected as potentially reusable systems during the feasibility study. Our objective is to detect parts of these systems that can be reused for the one that we are developing, taking into account that now we have a better understanding of the differences between the source and target formats, and of the software constraints.

Our assumption for proposing this task is that some translation decisions at all levels are normally reusable across different systems. We present two examples below:

- Let us suppose that the source format is a classical ontology language or an ontology tool with no support for the use of URIs for its identifiers, and that the target format is an ontology markup language. Hence the lexical transformations to be performed include transforming identifiers to valid URIs, by using the appropriate XML encoding, removing invalid characters in URIs, such as blank spaces, etc. The transformations to be performed will be similar no matter which are the specific source and target formats, as long as they belong to the previous groups.

- An interesting situation appears when the source and target formats belong to the same KR formalism than the source and target formats of another available system, respectively. In this case, we will be able to reuse many translation decisions at the semantic and pragmatic levels. For example, let us suppose that we are transforming ontologies from a description-logic format to a frame-based format. Description logic formats allow defining concepts with complex expressions that may include the union and the negation of concepts. These complex expressions are not usually allowed in frame-based formats, and normally we have to create anonymous classes to represent these expressions. The translation decisions taken are easily reusable, since they will be similar in all the possible translation systems with source and target formats belonging to those KR formalisms.

Similarly, we assume that many design decisions (e.g., those taken for the sake of integrating the ontology translation system in an ontology tool or platform) can be reused, and hence the ontology translation system developers can also benefit from reusing those parts of the design of existing translation systems.

Consequently, in this task we will aim at reusing two types of elements: translation decisions, and software design decisions. The first subtask will be performed by knowledge engineers, while the second subtask will be performed by software engineers.

Input products and output results
Input: Related technology description, ontology translation decision scope, and refined requirement catalogue.

Output: Catalogue of translation decisions to be reused.

Participants

Knowledge and software engineers.

Techniques

Cataloguing

Practical example

As a result of the task 1.2 we obtained a catalogue of ontology translation systems that could be reused, and of the OWL library to be used. The following specific aspects will be reused from each of them:

- WebODE export service to RDF(S).
  
  o Translation decisions: each WebODE instance is transformed to an RDF resource, whose type is the class that corresponds to the WebODE concept of which the instance is instance of. The values for its instance attributes and for its ad hoc relations are also transformed to RDF.
  
  o Software design decisions: for each WebODE instance set, this export service generates a file that contains the instances of the instance set. This is also the expected result for the WebODE export service to OWL DL, and hence it can be reused.

- WebODE export service to DAML+OIL.
  
  o Translation decisions: OWL DL is based on a description logic similar to DAML+OIL. Hence many translation decisions can be reused, such as those related to the transformation of the scope of instance attributes and ad hoc relations from being local to concepts to being global to the ontology. We can also reuse the transformations of formal axioms to DAML+OIL expressions. However, there are other aspects that cannot be reused, such as the transformation of concept disjoint and exhaustive decompositions, and partitions, since in DAML+OIL they could be represented easily with specific primitives whilst in OWL DL a workaround must be done, or transformations of instance attribute and ad hoc relation attachments into property restrictions, due to the fact that OWL DL does not define qualified property restrictions.
  
  o Software design decisions: the file that contains the ontology is output as a ZIP file in the WebODE export service to DAML+OIL. This is also the expected result for the WebODE export service to OWL DL, and hence it can be reused.

4.4.3.2 Task 3.2. Propose transformations at the pragmatic level

The objective of this task is to propose the set of transformations needed to comply with the restrictions related to the pragmatic preservation properties of the transformation. These
restrictions were imposed during the analysis activity, and described in the refined catalogue of requirements and in the ontology translation decision scope document.

This task will be mainly performed by knowledge engineers, who will describe in detail the transformations needed by means of diagrams and tables. This is the first task that we propose to start with, from the four-task group dealing with transformations in the different translation layers, as shown in figure 4.9. That is, we propose to use a top-down approach to design transformations: from the more abstract layer (pragmatic) to the most code-specific layer (lexical). As we presented elsewhere, translations will be driven by the pragmatic and semantic layers, and the decisions taken at the lower layers will be mainly influenced by the decisions taken in the more abstract ones.

As described in section 4.2.4, the translation decisions taken at the pragmatic layer mainly consist in deciding which transformations to use when several ones can be used indistinctly. This is usually the case when it is possible to use different syntaxes to express the same ontology component, or different workarounds to express the same knowledge, or where we have different choices for the transformations of identifiers, etc. In this layer we also decide which components must be created before and after the ontology translation, such as creating specific metaclasses, transforming sets of components in a simpler one, adding specific pieces of documentation to ontology components transformed, etc. Finally, in this layer we may also decide, in the case that the target format is an ontology tool, which transformations have to be made to the user interface so as to comply with the pragmatic properties determined during the analysis activity (we will show examples of these transformations in chapter 5).

**Input products and output results**

*Input:* Catalogue of translation decisions to be reused, ontology translation decision scope, and refined requirement catalogue.

*Output:* Catalogue of translation decisions at the pragmatic level.

**Participants**

Knowledge and software engineers.

**Techniques**

Cataloguing, and translation decision tables and diagrams.

**Practical example**
As stated in the functional requisite Req1, the ontology translation system to be developed will be mainly driven by pragmatics; that is, the ontologies must be readable enough both for human users and OWL-aware systems once they have been transformed.

Taking into account this requisite, our first decision has consisted in not transforming some WebODE ontology components that have not a direct correspondence in OWL DL, although they could be represented by means of workarounds. The only exception is related to the transformation of WebODE exhaustive decompositions and partitions of concepts, which will be transformed even when there are not specific OWL DL primitives to represent them.

The second decision is related to the differences in the scope of instance attributes/datatype properties and ad hoc relations/object properties between WebODE and OWL DL. In section 2.4.5 we already presented some translation possibilities related to this aspect. The following translation decisions are taken in the WebODE export service to OWL DL, as summarised in the decision tree of figure 4.10:

- If several WebODE ad hoc relations or instance attributes have the same name and the same range R, but their domains are different (D1, D2, ..., Dn), then we consider that they refer to the same OWL object property or datatype property, whose range is R and whose domain is D1 ∪ D2 ∪ ... ∪ Dn.

- If several WebODE ad hoc relations have the same name, the same domain D and different ranges (R1, R2, ..., Rn), then we consider that they refer to the same OWL object property, whose domain is D and whose range is R1 ∪ R2 ∪ ... ∪ Rn. This decision contrasts with what be interpreted in OWL if these domain and range definitions were expressed independently (R1 ∩ R2 ∩ ... ∩ Rn).

- If two WebODE ad hoc relations or instance attributes have the same name, but not the same domain nor range, then we consider that they refer to different OWL object properties or datatype properties respectively.

Relation S appears several times in the ontology

![Decision Tree for Translating WebODE Relations](image-url)
The third decision is related to the treatment of the class unions generated as a consequence of the previous transformations. These complex domains and ranges can be post-processed in order to obtain simpler expressions, so that if for any $C_i \in C$, it holds that $C_i \subseteq C \setminus C_i$, then $C_i$ can be removed from the expression.

### 4.4.3.3 Task 3.3. Propose transformations at the semantic level

Similarly to the previous task, in this one we must propose the transformations needed to comply with the restrictions related to the semantic preservation properties of the transformation. As with the pragmatic restrictions, the semantic ones were imposed during the analysis activity, and described in the refined catalogue of requirements and in the ontology translation decision scope document.

This task will be mainly performed by knowledge engineers, who will describe in detail the transformations needed by means of diagrams and tables. According to the top-down approach described earlier, this task will be the second one to be started, although it will be mainly performed in parallel. Besides, the figure 4.9 suggests that this task is the one that will require more effort (both in terms of time and of resources), since this is the most complex part of the ontology translation system design.

To specify the transformations at this layer, we recommend to use the same techniques than in the previous task, that is, we recommend using mainly tables and diagrams to specify the relationships between the ontology components in the source and target formats. Based on the experience, in this layer we assume that we will find more translation decisions to be reused from other translation systems. Besides, as an input of this task we will use the results of the tasks 2.1, 2.2 and 2.3 (used to describe the source and target formats, and to determine and compare their expressiveness), which will help to specify these translation decisions.

**Input products and output results**

**Input:** Catalogue of translation decisions to be reused, ontology translation decision scope, refined requirement catalogue, KR ontologies of source and target formats, and source and target format comparison.

**Output:** Catalogue of translation decisions at the semantic level.

**Participants**

Knowledge and software engineers.

**Techniques**

Cataloguing, and translation decision tables and diagrams.

**Practical example**
There are many translation decisions that can be taken in the semantic layer. As an example, we will show the proposal for transforming a WebODE concept partition into a class equivalence between the parent class and the union of all the subclasses, several subclass of relationships between the subclasses and the parent class, and disjointness axioms between each pair of classes in the partition. This transformation was already identified in the refined requisite catalogue (and more specifically, in table 4.9).

The design of this transformation is shown in the table 4.11 which shows the knowledge found in the source format (WebODE), and its exact correspondence in the target format (OWL DL).

Table 4.11. Semantic transformation of WebODE partitions to OWL DL.

<table>
<thead>
<tr>
<th>WebODE</th>
<th>OWL DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition ((C, {C_1, C_2, ..., C_n}))</td>
<td>(C = C_1 \cup C_2 \cup ... \cup C_n)</td>
</tr>
<tr>
<td></td>
<td>(C_i \subseteq C)</td>
</tr>
<tr>
<td></td>
<td>(\forall C_i \in {C_1, C_2, ..., C_n})</td>
</tr>
<tr>
<td></td>
<td>(C_i \cap C_j \models \bot)</td>
</tr>
<tr>
<td></td>
<td>(\forall C_i \neq C_j), (C_i, C_j \in {C_1, C_2, ..., C_n})</td>
</tr>
</tbody>
</table>

4.4.3.4 Task 3.4. Propose transformations at the syntax level

The objective of this task is to propose the transformations to be done at the syntax layer, so as to overcome the inherently different syntaxes of the source and target formats. Consequently, this task mainly consists in determining which are the syntactic restrictions of the source and target formats, which is the syntax to be used to express each ontology component specified in the previous tasks, and which are other specific correspondences between the source and target formats, such as correspondences in the datatypes used, in the information needed for the ontology headers, etc.

Unlike the previous two tasks, this task can be mainly performed by a software engineer, since it does not have much to do with the knowledge that is represented and its intended meaning, but with how it is represented physically. To perform this task, several techniques can be used: translation decision tables and diagrams, as in the previous cases, which are useful, for instance, for specifying the correspondences between datatypes; BNF grammars of the source and target formats (either for all the language expressions or for part of them), so that we can gain more understanding of how each component is represented, of how we should read expressions in the source format and of how we should generate expressions in the target format; and API descriptions of the source and target formats, in case that they provide it. As we will see later, during the implementation activity, we will force the source and target formats to provide an ontology management API, so that the transformations can be implemented more easily in the ODESyntax language described there. However, at this step this constraint should not be considered yet.

Input products and output results
Input: Catalogue of translation decisions to be reused, ontology translation decision scope, and refined requirement catalogue.

Output: Catalogue of translation decisions at the syntax level.

Participants

Software and knowledge engineers.

Techniques

Cataloguing, translation decision tables and diagrams, and BNF diagrams.

Practical example

The following examples show how to design transformations at the syntax level in the WebODE export service to OWL DL.

The table 4.12 shows the correspondence between the datatypes used for instance attributes in WebODE and the ones used in OWL DL.

<table>
<thead>
<tr>
<th>WebODE datatypes</th>
<th>OWL DL datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>&amp;xsd:boolean</td>
</tr>
<tr>
<td>Cardinal</td>
<td>&amp;xsd;nonNegativeInteger</td>
</tr>
<tr>
<td>Integer</td>
<td>&amp;xsd;integer</td>
</tr>
<tr>
<td>Float</td>
<td>&amp;xsd;float</td>
</tr>
<tr>
<td>String</td>
<td>&amp;xsd;string</td>
</tr>
<tr>
<td>Date</td>
<td>&amp;xsd;date</td>
</tr>
<tr>
<td>Range</td>
<td>&amp;xsd;float</td>
</tr>
<tr>
<td>URL</td>
<td>&amp;xsd:anyURI</td>
</tr>
<tr>
<td>XML Schema datatype x</td>
<td>if x e [built-in XML Schema datatypes] &amp;xsd;string otherwise</td>
</tr>
</tbody>
</table>

As another example of the documentation used in this task, we now show the description of the WebODE API for concept management, and an excerpt of the BNF grammar of OWL DL (in abstract syntax) for class generation.

WebODE API. Concept management

getConcept(ontology,concept):Concept
getChildConcepts(ontology,concept):Concept[]
g getParentConcepts(ontology,concept):Concept[]
g getGroups(ontology):Group[]
g getRootConcepts(ontology):Concept[]
g isSubClassOf(ontology,concept1,concept2):boolean
g getConceptsInWhichAttributeIsDefined(ontology,attribute):Concept[]

OWL DL Abstract syntax for concept generation

axiom := 'Class(' classID ['[Deprecated]' modality { annotation } { description } ']')
modality := 'complete' | 'partial'
description := classID
4.4.3.5 Task 3.5. Propose transformations at the lexical level

This is the last of the tasks in this group. Its objective is to propose the transformations to be done at the lexical level so as to overcome the differences between how the source and target formats express their ontology component identifiers and natural language descriptions, as well as their attribute values.

Like in the previous task, this task can be mainly performed by software engineers, which may use translation decision tables and diagrams, and BNF grammars of the ontology component identifiers, natural language documentation, attribute values, etc. Besides, in this layer, the reuse of translation decisions is also very common, since there are not many differences at this level between different formats.

As described in section 4.2.1, the most typical problems to be solved in this layer refer to the differences about the legal and illegal characters of the source and target formats, and problems related to the scope of ontology components (some ontology components are defined inside the scope of others in one format, while their correspondent ontology components are defined outside their scope), which may provoke collisions between identifiers.

**Input products and output results**

*Input:* Catalogue of translation decisions to be reused, ontology translation decision scope, and refined requirement catalogue.

*Output:* Catalogue of translation decisions at the lexical level.

**Participants**

Software and knowledge engineers.

**Techniques**

Cataloguing, translation decision tables and diagrams, and BNF grammars.

**Practical example**

The design of transformations at the lexical level in the WebODE export service to OWL DL are reused from the WebODE export service to DAML+OIL. There are specially three types of transformations to be highlighted at this level:
- OWL ontology component identifiers must be valid namespace-qualified names (QNames), which usually consists of two parts: a namespace and a local name, which in its turn must be an NCName [Bray et al., 2000]. Therefore, the ontology translation system must transform WebODE ontology components to QNames by prefixing all the components with the ontology namespace, and by ensuring that the ontology component identifier is transformed to a valid QName. The BNF grammar shown below [Bray et al., 2000] describes how to generate valid QNames:

\[
\begin{align*}
\text{QName} &::= (\text{Prefix}:')? \text{LocalPart} \\
\text{Prefix} &::= \text{NCName} \\
\text{LocalPart} &::= \text{NCName} \\
\text{NCName} &::= (\text{Letter}|\_)(\text{NCNameChar})* /\text{An XML Name, minus the ":" */} \\
\text{NCNameChar} &::= \text{Letter} | \text{Digit} | \_ | \text{CombiningChar} | \text{Extender} \\
\text{Letter} &::= \text{BaseChar} | \text{Ideographic} \\
\text{BaseChar} &::= \text{[\#x0041-\#x005A]} | \text{[\#x0061-\#x007A]} | \ldots \text{[\#xAC00-\#xD7A3]} \\
\text{Ideographic} &::= \text{[\#x4E00-\#x9FA5]} | \text{[#x3007]} | \text{[\#x3021-\#x3029]} \\
\text{CombiningChar} &::= \text{[\#x0300-\#x0345]} | \ldots | \text{[\#x3099]} | \text{[\#x309A]} \\
\text{Digit} &::= \text{[\#x0030-\#x0039]} | \ldots | \text{[\#x0F20-\#x0F29]} \\
\end{align*}
\]

- The scope of some ontology components is different in the source and target formats, as explained in the pragmatic level (section 4.4.3.2). The transformations proposed in that section are implemented at the lexical level.

- In WebODE, concepts and instance attributes can share the same identifier, while in OWL DL the sets of identifiers of classes and datatype properties are disjoint. Consequently, if a WebODE concept Name and a WebODE instance attribute Name have to be transformed to OWL DL, two different identifiers will be generated for them: for example, Name and Name_1 respectively.

### 4.4.3.6 Task 3.6. Propose additional transformations

Finally, in this task we must propose any additional transformation that has not been taken into account by any of the previous task, either because it was not possible to classify it easily in any of the previous layers, or because it is more related to the integration of the ontology translation system generated in other information systems (normally, in other ontology tools).

This task will be mainly performed by software engineers, who must take into account the software and integration restrictions of the resulting ontology translation system. We recommend the use of different techniques to specify these transformations: catalogues of design specifications to be taken into account in the implementation, translation decision tables and diagrams, as in the previous cases, or general design techniques such as class and interaction diagrams (we recommend using design patterns for this [Larman, 2001]), database design specifications, etc.

**Input products and output results**

*Input: Catalogue of translation decisions to be reused, ontology translation decision scope, refined requirement catalogue, and catalogues of translation decisions at the pragmatic, semantic, syntax and lexical levels.*
Output: Catalogue of additional transformations.

Participants

Software and knowledge engineers.

Techniques

Cataloguing, translation decision tables and diagrams, database design specifications, design class and interaction diagrams, design patterns, etc.

Practical example

The ontology translation system between WebODE and OWL DL must be integrated as a service in the WebODE platform. To achieve this goal, once that the code of the system has been generated, it has to be wrapped as a stateless service in the application server underlying the WebODE platform. Figure 4.11 shows the class diagram that corresponds to the classes and interfaces needed to integrate it as a WebODE export service. There are four classes, in charge of the service description and management (OWLExportServiceManager), the service configuration (OWLExportServiceConfiguration), the service interface (OWLExportService), and the service implementation (OWLExportServiceImpl). The exportOntology method of this class will call the necessary methods of the ontology translation service in order to perform the actual translation of ontologies, and will use the ODES ervice interface to access WebODE ontologies.

```
OWLExportService
  exportOntology(ontology : String, bConceptualisation : boolean, instanceSets : String[])  
  implements
  OWLExportServiceImpl
  exportOntology(ontology : String, bConceptualisation : boolean, instanceSets : String[])  
  ode
  ODEService
```

Figure 4.11. Classes and interfaces necessary for integrating the ontology translation system as a WebODE export service.

4.4.4 Activity 4. Implementation of the translation system

The objective of the implementation activity is to create the declarative specifications of the transformations to be performed by the ontology translation system, which will be used to
generate its final code. These declarative specifications are based on the results of the design activity (the ontology translation system design), and are implemented in three different formal languages: ODELex, ODESyntax, and ODESem, which correspond to the lexical, syntax, and semantic/pragmatic ontology translation layers, respectively. The same language (ODESem) is used for implementing semantic and pragmatic transformations because the translation decisions at both layers are similar.

Like in the design activity, the tasks inside this implementation activity are divided in groups, which are four in this case, as shown in figure 4.12. We describe next the goals and responsible of each task, which are also summarised in table 4.13:

The goal of the first task is to select reusable pieces of code from the declarative specifications of other ontology translation systems. These pieces of code are selected on the basis of the results obtained from the first task of the design activity, and can be related to any of the four translation layers. As occurred in the corresponding task of the design activity, this task is performed by knowledge and software engineers, who focus on different aspects of the existing translation systems. They will catalogue the pieces of code to be reused so that it will be easy to find them in the following implementation tasks.

The next five tasks are grouped together and should be performed almost in parallel, as shown in the figure. In these tasks the software and knowledge engineers must actually implement the transformations at the four layers: lexical, syntax, semantic, and pragmatics, and the additional transformations described in task 3.6. Unlike in the design activity, we propose to start with the low-level transformations (those at the lexical and syntax layers), and continue with the more abstract (and difficult) ones. The reason for the task ordering suggested is that the semantic and pragmatic transformation implementations usually need to take into account the specific implementations at the lexical and syntax layers.

In the next task (4.7: declarative specification processing and integration) the software engineer is in charge of transforming the previous declarative implementations at all levels, plus the additional transformations, into actual running code, which will perform the translations as specified in the previous code. Besides, the software engineer has to integrate the resulting ontology translation system into another information system (such as an ontology tool), if required. Given that most of the transformations have been implemented in formal languages, and that the additional ones have been specified in Java, most of the processes involved in this task can be automated. The ODEDialect system is in charge of this automation. If problems are detected during this task, the method recommends to go back to the implementation activities so as to solve them.
Finally, the method proposes to execute the test suite that was defined during the analysis activity, which are considered as the system tests for our system. This does not prevent us from defining and executing other kinds of tests (from unitary tests to integration tests) at any point during the development. This task consists on inputting the ontologies in the test suite to the resulting ontology translation system and checking whether the output corresponds to the one expected. Note that in most of the cases, this check will consist in comparing whether the output file(s) and the expected file(s) are identical, but there are cases where this kind of comparison will not be possible, since the results can come in any order (for instance, in RDF and OWL ontologies). If any of the test fails, we must go back to the previous implementation activities to detect the problems. Furthermore, we must consider that the method allows moving to previous activities if problems are detected at any point of our development.
Figure 4.12. Task decomposition of activity 4 (implementation of the translation system).

<table>
<thead>
<tr>
<th>Task</th>
<th>Products</th>
<th>Techniques</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 4.1. Find and reuse similar translation system</td>
<td>Catalogue of translation functions to be reused</td>
<td>Cataloguing</td>
<td>Software engineer, Knowledge engineer</td>
</tr>
<tr>
<td>Task 4.2. Propose transformations at the pragmatic level</td>
<td>ODESem implementation</td>
<td>ODESem</td>
<td>Knowledge engineer, Software engineer</td>
</tr>
<tr>
<td>Task 4.3. Propose transformations at the semantic level</td>
<td>ODESem implementation</td>
<td>ODESem</td>
<td>Knowledge engineer, Software engineer</td>
</tr>
<tr>
<td>Task 4.4. Propose transformations at the syntactic level</td>
<td>ODESyntax implementation</td>
<td>ODESyntax</td>
<td>Software engineer, Knowledge engineer</td>
</tr>
<tr>
<td>Task 4.5. Propose transformations at the lexical level</td>
<td>ODELex implementation</td>
<td>ODELex</td>
<td>Software engineer, Knowledge engineer</td>
</tr>
<tr>
<td>Task 4.6. Propose additional transformations</td>
<td>Additional transformations implementation</td>
<td></td>
<td>Software engineer, Knowledge engineer</td>
</tr>
<tr>
<td>Task 4.7. Declarative specification processing and integration</td>
<td>Java source code, Integrated translation system</td>
<td>ODEDialect</td>
<td>Software engineer</td>
</tr>
<tr>
<td>Task 4.8. Test suite execution</td>
<td>Test results, Corrective actions, System tests</td>
<td></td>
<td>Software engineer, Knowledge engineer</td>
</tr>
</tbody>
</table>

Table 4.13. Task products, techniques and participants of activity 4 (implementation of the translation system).
4.4.4.1 Task 4.1. Find translation functions to be reused

In this task we select the pieces of code that will be reused from other ontology translation systems, which may include parts of their declarative specifications or user-defined functions. The input for this task is the catalogue of reusable translation decisions obtained as a result of its counterpart task during the design activity (task 3.1).

The pieces of code obtained may belong to any of the four ontology translation layers, and will be inserted in the corresponding layer in the next implementation tasks. The selection is performed by software and knowledge engineers, who focus on different aspects of the reuse, as occurred during the design.

**Input products and output results**

*Input:* translation system design document (catalogue of translation decisions to be reused).

*Output:* catalogue of translation functions to be reused.

**Participants**

Software and knowledge engineer.

**Techniques**

Cataloguing.

**Practical example**

In task 3.1 we identified two translation systems from which we could reuse many translation and software design decisions: the WebODE export services to RDF(S) and DAML+OIL. The objective of this task is to select the exact pieces of code that will be reused, so that they can be used in the next implementation tasks.

We will show some of the reusable pieces of code selected from the DAML+OIL export service:

With regard to translation decisions, one of the aspects to be reused was the transformation of the scope of instance attributes and ad hoc relations from being local to concepts to being global to the ontology. As expressed elsewhere, this transformation can be done in at least two different ways: creating an identifier for each of them, or trying to find commonalities in their domain and ranges and building complex expressions from them. We will opt for the second one, which was also implemented in the export service to DAML+OIL, and will reuse the following piece of code from the ODELex specification of that export service.

```plaintext
%WebODE.InstanceAttribute IDENTIFIER WebODE.Concept IDENTIFIER WebODE.Type IDENTIFIER INIT: (S1=convertToURI($1))
TABLE: {{WebODE.InstanceAttribute,$1,$2,$3,[OWL.DatatypeProperty,$1]}}
REPEATED: {{WebODE.InstanceAttribute,$1,-,$3} => (S1=get(WebODE.InstanceAttribute,$1,-,$3)),
default => (S1=addNumber(S1))}
OVERLAP: (S1=addNumber(S1))

%WebODE.AdHocRelation IDENTIFIER WebODE.Concept IDENTIFIER WebODE.Concept IDENTIFIER /* The 2nd and 3rd identifiers are of the domain and range concepts */
INIT: (S1=convertToURI($1))
```

With regard to the implementation code, we selected for reuse the process needed to output the files that contain the translated ontology as a ZIP file. The following function is reused from the WebODE export service to DAML+OIL (we only show its signature):

```java
private byte[] writeOntologyAndInstancesToFile
(String fileDirectory, boolean bConceptualization,
 String[] instanceSets, String namespace)
```

### 4.4.4.2 Task 4.2. Implement transformations in the pragmatic level

The objective of this task is to implement the ontology translation decisions proposed at the pragmatic layer, which were included in the translation system design document. The implementation will be mainly coded with ODESem, a language that has been created specifically for this task (and also for the semantic layer, as will be described later). In case that a transformation cannot be expressed in this language, a general-purpose programming language can be used to express them.

The task will be performed by knowledge and software engineers: the former will implement the transformations in ODESem, and the latter will implement the additional transformations needed. Besides, the pieces of code selected previously will be reused in this task.

**Input products and output results**

- **Input:** translation system design document (catalogue of translation decisions at the pragmatic level)

- **Output:** ODESem implementation.

**Participants**

Knowledge and software engineer.

**Techniques**

ODESem (cf. section 4.3.3).

**Practical example**

Some pieces of the declarative code needed to express the transformations at the pragmatic level for the WebODE export service to OWL DL were already shown in section 4.3.3.
4.4.4.3 Task 4.3. Implement transformations in the semantic level

The objective of this task is to implement the ontology translation decisions proposed at the semantic layer, which were included in the translation system design document. The implementation will be mainly coded with ODESem, as occurred with the implementation of transformations in the pragmatic layer. The reason for using the same language to implement transformations in both layers (semantic and pragmatic) is that the transformations to be performed in both layers are usually treated similarly. Furthermore, like in task 4.2, in case that a transformation cannot be expressed in this language, a general-purpose programming language can be used to express them.

The task will be performed by knowledge and software engineers: the former will implement the transformations in ODESem, and the latter will implement the additional transformations needed. Besides, the pieces of code selected in task 4.1 will be reused.

Input products and output results

- **Input**: translation system design document (catalogue of translation decisions at the semantic level)
- **Output**: ODESem implementation.

Participants

Knowledge and software engineer.

Techniques

ODESem (cf. section 4.3.3).

Practical example

Some pieces of the declarative code needed to express the transformations at the semantic level for the WebODE export service to OWL DL were already shown in section 4.3.3.

4.4.4.4 Task 4.4. Implement transformations in the syntax level

The objective of this task is to implement the ontology translation decisions proposed at the syntax layer, which were included in the translation system design document. The implementation will be mainly coded with ODESyntax, a language specifically created to implement this type of transformations. Like in the previous implementation tasks, in case that a transformation cannot be expressed in ODESyntax, a general-purpose programming language can be used to express them.
The task will be mainly performed by software engineers, although knowledge engineers may also help in this task. Besides, the pieces of code selected in task 4.1 will be reused.

**Input products and output results**

*Input:* translation system design document (catalogue of translation decisions at the syntax level)

*Output:* ODESyntax implementation.

**Participants**

Software and knowledge engineer.

**Techniques**

ODESyntax (cf. section 4.3.2).

**Practical example**

Some pieces of the declarative code needed to express the transformations at the syntax level for the WebODE export service to OWL DL were already shown in section 4.3.2.

### 4.4.4.5 Task 4.5. Implement transformations in the lexical level

The objective of this task is to implement the ontology translation decisions proposed at the lexical layer, which were included in the translation system design document. The implementation will be mainly coded with ODELex, a language specifically created to implement this type of transformations. Like in the previous implementation tasks, in case that a transformation cannot be expressed in ODELex, a general-purpose programming language can be used to express them.

The task will be mainly performed by software engineers, although knowledge engineers may also help in this task. Besides, the pieces of code selected in task 4.1 will be reused.

**Input products and output results**

*Input:* translation system design document (catalogue of translation decisions at the lexical level)

*Output:* ODELex implementation.

**Participants**

Software and knowledge engineer.

**Techniques**
ODELex (cf. section 4.3.1).

Practical example

Some pieces of the declarative code needed to express the transformations at the lexical level for the WebODE export service to OWL DL were already shown in section 4.3.1.

4.4.4.6 Task 4.6. Implement additional transformations

The objective of this task is to implement the rest of transformations needed for the ontology translation system, and which have not been expressed in any of the four previous tasks. These additional transformations were designed during task 3.6, by means of general-purpose design techniques, and must be implemented in this task similarly, using general-purpose implementation techniques.

This task will be mainly performed by software engineers, although knowledge engineers may also help to implement some of the transformations. Besides, pieces of code selected in task 4.1 can be reused.

Input products and output results

Input: translation system design document (catalogue of additional transformations)

Output: Additional transformations implementation (normally, Java code to be appended to the translation system before its integration).

Participants

Software and knowledge engineer.

Techniques

--

Practical example

As a result of this task, the implementation code needed to integrate the ontology translation system as a WebODE export service is generated. We also reused this part of the code from the WebODE export service to DAML+OIL.

4.4.4.7 Task 4.7. Declarative specification processing and integration

In this task, the multi-layered declarative implementation of the ontology translation system, together with the additional functions that might have been implemented, are processed in order to generate the final code of the ontology translation system, and to integrate it, in case that it is necessary.
This task is performed by software engineers, who follow a set of recommendations for generating this final code. These recommendations are different for each layer, since the problems to be solved and the code to be generated in each layer are different and hence need different programming techniques. Given that most of the transformations have been implemented in formal languages, and that the additional ones have been specified in Java, most of the processes involved in this task can be automated.

If a problem appears during this task, and we detect that it is due to a problem in any of the previous implementation tasks, then we must go back to those tasks in order to solve them and generate the translation system correctly.

**Input products and output results**

**Input:** translation decision implementations in ODESem, ODESTSyntax, ODELex, and additional transformations.

**Output:** Java source code of the ontology translation system, and integrated software product.

**Participants**

Software engineer.

**Techniques**

ODEDialect.

**Practical example**

---

**4.4.4.8 Task 4.8. Test suite execution**

The objective of this task is to determine the correct functioning of the ontology translation system generated and integrated in the previous task. To check the ontology translation system's validity, we will use the test suite that was defined during the analysis activity, which was created as a set of system tests that the system should pass successfully in order to consider it valid. As we described in the task 2.6, the test suite contains a set of input ontologies in the source format of the ontology translation system and their corresponding output ontologies in the target format. The set usually covers exhaustively the possibilities where the translation system must take translation decisions.

To pass the test, the output ontologies must correspond to the ones expected in each case. As we commented in the introduction to this implementation activity, this check will normally consist
in comparing whether the output file(s) and the expected file(s) are identical. However, there will be cases where this kind of comparison will not be possible, since the results can be generated in any order.

If any of the test fails, we must go back to the previous implementation activities to detect the source of the problems and propose corrective actions. Furthermore, we must consider that the method allows moving to previous activities if problems are detected at any point of our development.

This task must be performed by software and knowledge engineers, since they have to detect, in case of unsuccessful cases, which are the sources of the problems detected, and propose the corrective actions.

**Input products and output results**

- **Input**: integrated ontology translation system, test plan and test suite.
- **Output**: test results and proposed corrective actions.

**Participants**

Software and knowledge engineer.

**Techniques**

Test systems.

**Practical example**

The result of task 2.6 was a test plan that contained a test suite with synthetic ontologies to be tested in order to check the conformance of the WebODE export service to OWL DL to the expected transformation results. We show below an example of one of the errors found when running these tests, and the corrective action taken to solve it:

One of the WebODE ontologies included in the test contained the concept \( C \), the instance attribute \( C \) defined in the concept \( C \) and with datatype \( String \), and the instance \( C \) of concept \( C \). This is possible in WebODE since the sets of identifiers for these ontology components are not necessarily disjoint, though they refer to different components. The expected result of the test was a class, a datatype property and an instance, all of them with different identifiers (e.g., \( C \), \( C\_1 \), and \( C\_1\_1 \) respectively).

When the test was run the first time, we discovered that there were only two different identifiers, because the concept and the instance had the same identifier. This is not valid OWL DL, but OWL Full, and hence it was the result of an incorrect transformation. We analysed the lexical transformation code from the ontology translation system and we discovered that we had not explicitly stated that there could not be overlap between OWL class identifiers and OWL individual identifiers. Hence the proposed corrective action was to express explicitly that "no-overlapping" relationship and the next time that the test was executed, it successfully passed.
Chapter 5
Semantic and Pragmatic Preservation in Ontology Translation

This chapter presents four ontology translation approaches used so far in different contexts, taking into account the two perspectives related to the semantic and pragmatic ontology translation layers. The chapter describes how these approaches preserve the knowledge represented in an ontology and its intended meaning when the ontology is transformed between two formats. These descriptions put special emphasis in the lifecycle of knowledge in cyclic ontology translation processes.

5.1 Transformation Approaches and Their Implications in Semantic and Pragmatic Preservation

The ontology translation model proposed in this thesis considers that ontology translation problems can appear, and hence have to be solved, at four different but interrelated layers: lexical, syntax, semantic, and pragmatic. The problems that appear at these four layers are equally important: all of them must be solved to allow translating successfully an ontology from the source to the target format. However, as described in chapter 4, the decisions taken at the semantic and pragmatic layers are the ones that usually drive the transformations to be made by an ontology translation system.

The overall objective of an ontology translation system is normally to obtain an ontology in the target format that is equivalent to the original ontology available in the source format. The concept of equivalence in ontology translation could be considered ambiguous. To overcome this ambiguity, the objective can be reformulated as follows: obtaining an ontology in the target format with maximal preservation properties in each of the four ontology translation layers.
Let \( \tau \) be a transformation function between the formats \( L \) and \( L' \) \((\tau: L \rightarrow L')\), which is implemented by an ontology translation system. [Euzenat, 2001] describes the following maximal preservation properties at each translation level:

- At the lexical layer, \textit{synset preservation} (a synset is the set of connected components of the synonymy graph \( S \)). That is, \( \forall t, t' \in L, S(t) = S(t') \Rightarrow S(\tau(t)) = S(\tau(t')) \).

- At the syntactic level, \textit{order preservation}. That is, given two order relations for the formats \( L \) and \( L' \) (\( \leq_L \) and \( \leq_{L'} \) respectively), \( \forall r, s \in L, r \leq_L s \Rightarrow \tau(r) \leq_{L'} \tau(s) \).

- At the semantic level, \textit{consequence preservation}. That is, \( \forall r, s \in L, r \models_L \delta \Rightarrow \tau(r) \models_{L'} \tau(\delta) \).

- At the pragmatic level, \textit{interpretation preservation}. That is, let \( \Sigma \) be the interpretation rules, \( P \) the set of persons or systems that must interpret the expressions defined in the format \( L \), and \( \models^i \) the interpretation relation for person or system \( i \), \( \forall r, s \in L, \forall i, j \in P, (r, \Sigma \models^i \delta \Rightarrow \tau(r), \Sigma \models^j \tau(\delta)) \).

However, in many cases not all of these maximal preservation properties can be achieved simultaneously in an ontology translation system, since there is usually an important trade-off among the preservation properties that can be obtained at each layer. For instance, it is not always easy to achieve maximum semantic preservation (consequence preservation) and maximum pragmatic preservation (intended meaning or interpretation preservation) at the same time. Let us show an example of a transformation from the ontology language OWL DL to the ontology tool Protégé-2000 in order to illustrate this trade-off:

Let us suppose that we have an OWL DL ontology with the class Person and the object property \textit{hasParents}, whose domain is Person, whose cardinality is 2, and whose range is the union of the classes Man and Woman. Besides, the class Person has two existential restrictions regarding this property, stating that any instance of this class must be connected at least to one instance of the class Man, and to one instance of the class Woman. In the DL formalism, this is expressed as follows:

\[
\text{Person} \subseteq (\geq 2 \text{hasParents}) \cap (\leq 2 \text{hasParents}) \cap (\forall \text{hasParents.(Man} \cup \text{Woman})) \cap \\
(\exists \text{hasParents.Man}) \cap (\exists \text{hasParents.Woman})
\]

There are at least two alternatives to transform this expression to Protégé-2000:

- Preserving the semantics of the original expression, while making it more difficult for human users and applications to understand it in the target format. We create the slot \textit{hasParents} whose domain is the class Person, whose range is the class
ManOrWoman, and whose cardinality is 2. This slot is attached to the class Person. The class ManOrWoman is declared as an abstract class, with the classes Man and Woman as its subclasses. Besides, we create an additional PAL constraint stating that if an instance of the class Person has two values for the slot hasParents, then one of them is instance of the class Man and the another is instance of the class Woman, or viceversa. A diagram of the classes and relations obtained, together with the PAL constraint, is shown in figure 5.1.

![Figure 5.1. Semantic preservation transformation of the class Person to Protégé-2000.](image)

Not preserving the exact semantics of the original expression, while making the transformed ontology easy to understand. We create two additional slots: hasFather (with domain Person and range Man) and hasMother (with domain Person and range Woman), both of them subslots of the slot hasParent and both of them with cardinality 1. These slots are attached to the class Person, as shown in figure 5.2.

![Figure 5.2. Pragmatic preservation transformation of the class Person to Protégé-2000.](image)
One of these alternatives has to be selected for transforming the OWL DL ontology. The selection to be made will depend on the preservation properties needed for the ontology translation system: maximal preservation in the semantic layer or maximal preservation in the pragmatic layer.

In this chapter we explore different approaches that have been used so far for ontology translation between heterogeneous formats, and comment on their main features (advantages and disadvantages) from the points of view of semantic and pragmatic preservation. We will pay special attention to the preservation of the semantics and of the intended meaning of knowledge in cyclic transformations, that is, transformations where ontologies are finally obtained in the source format after combining several intermediate transformations.

This analysis is included as a separate chapter of this thesis, instead of being included as part of the state of the art, because it presents a deep analysis and comparison of the four translation approaches from the perspectives of semantic and pragmatic preservation, hence contributing to the current state of the art.

The four alternatives explored are:

- Indirect ontology translation between the source and target formats by means of a common interchange format (section 5.2).
- Direct ontology translation between the source and target formats without using additional components or workarounds for representing the knowledge that cannot be directly represented in the target format (section 5.3.1).
- Direct ontology translation between the source and target formats by instantiating the source format KR ontology in the target format (section 5.3.2).
- Direct ontology translation between the source and target formats by using the knowledge components of the target format as much as possible, and by using only additional components or workarounds for representing the knowledge that cannot be directly represented in the target format (section 5.3.3).

Finally, we will compare in section 5.4 these transformation approaches according to the two parameters used for the description: semantic and pragmatic preservation properties.
5.2 INDIRECT ONTOLOGY TRANSLATION BY MEANS OF COMMON INTERCHANGE FORMATS

In the beginning of the 1990s, several standard knowledge representation formats were proposed to exchange knowledge between heterogeneous systems. Among them, we can cite: CKRL (Common Knowledge Representation Language) [Morik et al., 1991] for exchanging knowledge between machine learning tools; EXPRESS [Spiby, 1992] for exchanging product descriptions; SUMM (Semantic Unification Meta-Model) [Fulton et al., 1992] for enterprise integration; KIF (Knowledge Interchange Format) [Genesereth and Fikes, 1992] and Ontolingua [Farquhar et al., 1997] for exchanging knowledge between generic knowledge representation systems; PIF (Process Interchange Format) [Lee et al., 1998] for exchanging knowledge about processes; KRSL (Knowledge Representation Specification Language) [Allen and Lehrer, 1992] for exchanging knowledge about planning, etc.

A common feature of most of the previous knowledge exchange languages is that they were not created nor intended to be used as the internal format of KR systems, but only for knowledge communication and exchange tasks. Another common feature is that these standardization efforts were aimed at reducing the number of translators needed to achieve interoperability among a heterogeneous group of languages or systems (from $O(n^2)$ to $O(n)$ translators for a number $n$ of systems). This approach corresponds to the pivot architecture described in section 2.5.

Recently, the World Wide Web Consortium (W3C) has proposed three languages as recommendations of the emerging Semantic Web, namely RDF, RDF Schema, and OWL. As described in chapter 2, these languages have been created with the purpose of becoming widely used standards for knowledge representation in the Semantic Web. As a consequence, they can be also considered as potential common exchange languages, although this is not their main objective.

Given our focus on the ontology field, in this section we will only describe and analyse the use of KIF, RDF(S), and OWL as common knowledge interchange formats (aka exchange format or interlinguas). There are two main advantages in using them for ontology translation processes. First, the number of translation systems needed to transform ontologies between two formats is reduced. Second, if the source and target formats already provide export and import services to and from any of these interlinguas, the implementation effort needed to perform the ontology translation is dramatically reduced.
Taking into account the previous consideration, we will describe the main advantages and drawbacks of each format, with examples of how actual translation systems translate to and from these formats. In our analysis of the semantic and pragmatic preservation properties of this approach, we will show that in these interchange formats there is a trade-off between being strict enough to allow exchanging ontologies easily and being extensible enough to ensure knowledge preservation.

5.2.1 KIF as an interchange format

As described in chapter 2, the KIF language was developed in the beginning of the 1990s to facilitate knowledge sharing. KIF’s main features were its declarative semantics and its provision for expressing arbitrary sentences in first order logic, using four kinds of expressions: terms, sentences, rules, and definitions.

Despite its development as an interchange format, KIF has not been widely used in ontology exchange between ontology tools and languages. The interoperability tables appearing in [Gómez-Pérez, 2002] confirm that after one decade of KIF existence there are only two ontology tools able to translate ontologies to and from KIF: the Ontolingua Server and OntoSaurus. Besides, WebOnto translates ontologies to Ontolingua, which can be transformed later to KIF by the Ontolingua Server, but WebOnto does not import Ontolingua nor KIF ontologies. The main conclusion to be extracted from this lack of translation systems is that KIF has not succeeded as a de facto standard in the ontology community. Let us see some of the reasons for this lack of success.

1. KIF allows representing the same knowledge in many different ways, making knowledge exchange difficult. There are many examples that show this problem, derived from the flexibility of KIF’s knowledge model. For instance, the subclass relationship between the classes A and B can be stated with the following two expressions:

\[(\text{Subclass-Of A B})\]

\[(=> (A ?x) (B ?x))\]

The first expression uses the relation Subclass-Of between the classes A and B. The second expression, which is equivalent, expresses that an instance of the class A is also an instance of the class B1.

\[\text{Note that } \forall x (A(x) \Rightarrow B(x)) \text{ in classic first order logic does not have exactly the same meaning as Subclass-Of}(A,B) \text{ in the paradigm of frames. For example, non-monotonic reasoning is usually considered when reasoning with frames, and disregarded in classic first order logic. To learn more about the relationships between frames and first order logic, consult [Brewka, 1987].}\]
[Pease et al., 2000] show a more complex example related to how to express the type of the KIF variables used inside a KIF expression:

1. With KIF-style typed quantifiers, before the KIF expression:

   \[(\forall ((?x \text{ Class1}) (?y \text{ Class2})) ...)\]

2. With \text{instance-of} relations, inside the KIF expression:

   \[(\forall (?x ?y) (\text{and} (\text{instance-of} ?x \text{ Class1}) (\text{instance-of} ?y \text{ Class2}) ...))\]

3. With predicates whose name is the class to which the variable belongs, inside the KIF expression:

   \[(\forall (?x ?y) (\text{and} (\text{Class1} ?x) (\text{Class2} ?y) ...)).\]

The flexibility to express the same knowledge in many different ways provokes many ontology translation problems, especially during the process of importing ontologies from the interlingua. To solve these problems, it is usually advisable to accompany knowledge exchange standards by a style guide that imposes how to represent specific pieces of knowledge in the interlingua for exchange purposes. For instance, such a style guide may impose that the type of the variables used in KIF axioms must be defined inside the quantifier specification, as in the first option of the previous example.

2. **Translations to KIF are usually written with regard to a specific format.** The current KIF translation systems commonly use in their translations the vocabulary of the KR ontology of the source format from which they are transformed.

As an example, figure 5.3 shows two definitions in Ontolingua and LOOM, which are equivalent. Both of them define the attribute engine-count of the concept Moving-Object\(^2\). The attribute type is integer and its cardinality is one (consequently, in Ontolingua it is represented as a function). The figure also shows the corresponding KIF code generated by the Ontolingua Server and OntoSaurus, which have many differences between each other.

\(^2\) The LOOM definition is extracted and adapted from the ontology Aircraft, which is available in the demo version of the OntoSaurus tool.
A declarative approach to ontology translation with knowledge preservation

**LOOM**

\[
\text{(defrelation \(\text{ENGINE-COUNT}\))}
\text{domain \(\text{MOVING-OBJECT}\) - range \(\text{INTEGER}\) - characteristics \(\text{SINGLE-VALUED}\))}
\]

**Ontolingua**

\[
\text{(define-function \(\text{Engine-Count}\))}
\text{(domain \(\text{Engine-Count}\) - range \(\text{INTEGER}\))}
\text{(arity \(\text{Engine-Count}\) 2)}
\text{(binary-relation \(\text{Engine-Count}\))}
\]

![LOOM → KIF](image1)

![Ontolingua → KIF](image2)

**Figure 5.3. Different KIF representations of the same single-valued attribute.**

Each tool generates KIF expressions according to their own vocabularies. The Ontolingua Server generates KIF expressions with the terms Function, Domain, Range, Arity, etc., which are relations defined in the Frame Ontology, and which have their corresponding KIF axioms in that ontology. OntoSaurus generates KIF expressions using "pure" first-order logic expressions, except for the \(\text{LOOM:SINGLE-VALUED}\) expression, which represents the restriction of the relation cardinality to one.

The previous example illustrates a problem that is not only found in translations from the Ontolingua Server and OntoSaurus. This problem is also rather common in KIF translations from other source formats. For instance, Raschid and Vidal [Raschid and Vidal, 1996] describe how they translate F-Logic ontologies to KIF in order to create a KIF mediator to interoperate with multiple databases. The generated KIF expressions use the F-Logic vocabulary: \text{is-a} predicates to express the \text{Subclass-Of} relationship, \text{fun-value} predicates to express the attribute values, etc. The KIF axioms corresponding to these F-Logic-style predicates, which are necessary to deal with the KIF expressions obtained, are also translated into a Meta-Knowledge Base.

In an ideal scenario, KIF theories should be translatable from and to multiple languages. However, in practice, there is a trade-off between the neutrality of KIF translations and their operational use in different operational languages, as shown as a result of the Neutral Representation project [Barley et al, 1997]. If KIF theories are written without regard to a specific source or target language, then the theories become untranslatable and hence of no operational use; on the other hand, if KIF theories are written for a specific source or target language (using only KIF constructs that are easily translatable to and from it) then the
translation loses all the benefits of neutrality and theory’s contents will not be correctly
delivered to systems based on other languages.

3. Interoperability with KIF has been only proved with the same origin and target
languages. Translation to and from KIF has been demonstrated for individual languages
(LOOM and Ontolingua). For instance, if we translate from Ontolingua to KIF, and then back to
Ontolingua, the KIF translation system is able to recognize easily the Ontolingua-based
vocabulary previously generated. However, these experiments are of little use to prove the use
of KIF for interoperability [Ginsberg, 1991]. As an example, figure 5.4 shows how the
Ontolingua Server imports the KIF codes of the attribute engine-count from figure 5.2.

![Ontolingua](image)

Figure 5.4: Results of importing to the Ontolingua Server the KIF code generated from LOOM and
Ontolingua.

The Ontolingua definition obtained from the LOOM-generated KIF (figure 5.4.a) loses the
cardinality knowledge. Besides, the domain and range of engine-count are expressed by
means of a first order logic axiom; that is, the definition does not use the Ontolingua relations
Domain and Range, as presented in figure 5.3. Although both definitions express the same
knowledge, the one presented in figure 5.4.a is less readable than the one presented in figure
5.3.

This problem even appears in the translation to Ontolingua of the Ontolingua-generated KIF
code (figure 5.4.b). The relations Domain and Range are used inside an :axiom-def expression
(they do not use variables in the expressions, but the function and class names), which is
equivalent to the :def expression (which uses variables). And other primitives are used to express redundant knowledge, such as Function, Arity, Function-Arity, and Binary-Relation.

The reason for these bad results is due to the variety of styles used to generate KIF expressions. This variety makes it difficult for automatic translation systems to recognize the KIF constructs generated by other systems. How can an automatic translation system to recognize, for instance, the Subclass-Of predicate? What if is-a is used instead? Or is the translator into the interlingua expected to provide a declarative description of the predicates used (that Subclass-Of is transitive and so on, included in a Meta-Knowledge Base such as in the F-Logic example) and the translator out is supposed to recognize that?

In summary, all the problems that appear when using KIF as an interchange language are mainly related to the pragmatic layer: KIF was designed so as to allow representing the knowledge that could be represented in a wide variety of knowledge representation languages. Due to its flexibility, the same knowledge can be expressed in many different and equivalent ways, which makes it difficult for other systems to understand it correctly. Due to this problem in the pragmatic layer, some other problems appear also in the semantic layer: some knowledge can be lost in the translation process, especially in the case where the source and target formats are not the same.

5.2.2 RDF(S) as an interchange format

In contrast to KIF, [Gómez-Pérez, 2002] shows that RDF(S) has been widely adopted by ontology tools as an import/export format. That is, the combination of RDF and RDF Schema has become a de facto standard for exchanging ontologies. Furthermore, not only ontology tools have adopted this language combination for ontology exchange, but also other general software tools have adopted RDF(S) as their content exchange format (Internet browsers like Mozilla, search engines and content providers like Amazon, Alta Vista, etc.).

However, the availability of RDF(S) export/import services in most of the ontology tools does not mean that these tools can interoperate. In other words, they do not ensure that ontologies can be actually exchanged without modifying manually their source code or without losing knowledge in the translation process. This conclusion was recently reached by the OntoWeb Special Interest Group on Enterprise-Standard Ontology Environments[^1], as a result of the

[^1]: http://delicias.dia.fi.upm.es/ontoweb/sig-tools/index.html
interoperability experiment carried out for the ISWC2003 Workshop on Evaluation of Ontology Tools (EON2003)[4] [Sure et al., 2003], and is also presented in [Polikoff and Allemang, 2003].

The experiment consisted in analysing how ontologies could be exchanged between different tools and/or languages using RDF(S) and/or OWL as interchange formats. The experiment results can be found in [Isaac et al., 2003; Corcho et al., 2003a; Fillies, 2003]. They deal with the following ontology tools: Differential Ontology Editor[5], KAON [Maedche et al., 2003], OilEd [Bechhofer et al., 2001], OntoEdit [Sure et al., 2002], Protégé-2000 [Noy et al., 2000], SemTalk [Fillies et al., 2002], and WebODE [Arpírez et al., 2003]. Let us summarise some of the conclusions that can be extracted from this experiment and from our analysis of the main features of RDF(S):

1. **RDF(S) allows representing the same knowledge in different ways, making knowledge exchange difficult.** RDF(S) documents can be serialised with three different syntaxes: RDF/XML [Beckett, 2004], Notation-3[6] (N3), and its subset N-Triples[7]. The first syntax is mainly intended for machine-consumption and is normative; however, there are several ways to express the same knowledge with it. The second and third ones are mainly intended for human-consumption. The figure 5.5 shows the definition of the class A, which is a subclass of the class B, with two different syntaxes for RDF/XML, with Notation-3 and N-Triples.

<table>
<thead>
<tr>
<th>Abbreviated RDF/XML</th>
<th>RDF/XML</th>
<th>Notation-3</th>
<th>N-Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;/rdfs:Class&gt;</code></td>
<td><code>&lt;rdfs:subClassOf rdf:resource=&quot;#B&quot;/&gt;</code></td>
<td><code>&lt;http://examples.org/sampleOntology#A&gt;</code></td>
<td><code>&lt;http://www.w3.org/2000/01/rdf-schema#subClassOf&gt;</code></td>
</tr>
</tbody>
</table>

Figure 5.5. Different syntaxes used to represent the same knowledge in RDF(S).

---

Most of the existing RDF(S)-aware APIs and developer tools\(^8\) can manage the previous notations. They can create the object model that corresponds to an RDF(S) ontology, independently of its syntax, and can serialize an object model in at least one of these notations. However, some ontology tools still use directly RDF(S) source code instead of intermediate object models, which cause problems, as shown in [Isaac et al., 2003].

2. The standard knowledge model of RDF(S) is not expressive enough to represent most of the knowledge that can be represented with existing ontology languages and tools. As described in chapter 2, the standard RDF(S) knowledge model, also known as "plain RDF(S)", is based on the combination of two knowledge representation paradigms: semantic networks and frames. It allows defining classes, organized in class taxonomies, and properties, also organized in property hierarchies, and with their domain and range specified. However, it cannot express types of knowledge that can be usually represented in ontology tools and in other ontology languages, such as cardinality constraints of properties, disjoint and exhaustive knowledge in class taxonomies, first order logic axioms, etc.

For instance, the attribute `engine-count` that we dealt with in the previous section is implemented in plain RDF(S) as follows, without expressing its cardinality restriction:

```xml
<rdf:Property rdf:ID='engine-count'>
  <rdfs:domain rdf:resource='tVehicle'/>
  <rdfs:range rdf:resource='http://www.w3.org/2001/XMLSchema#integer'/>
</rdf:Property>
```

As a consequence of its limited expressiveness, plain RDF(S) allows exchanging only very lightweight ontologies without losing knowledge in the transformation process. In the case of heavyweight ontologies, much knowledge is lost.

3. Translations to RDF(S) are usually written with regard to a specific language or tool. As stated above, if we translate a heavyweight ontology from a more expressive language or tool to plain RDF(S) we will lose knowledge in the resulting RDF(S) ontology. To overcome this problem, two solutions have been proposed until now. These solutions have the goal of preserving knowledge in the transformation process through RDF(S):

a) **Represent the knowledge that could be lost inside rdfs:comment tags, using a specific structure for this information.** For instance, plain RDF(S) cannot express class attributes. To preserve this knowledge when translating to RDF(S), [Isaac et al., 2003] propose to represent inside a comment the values of the class attributes of a class: similarity with parent (SWP),

---

\(^8\) A list of RDF-aware tools can be found at http://www.ilrt.bris.ac.uk/discovery/rdf/resources/
similarity with siblings (SWS), difference with siblings (DWS) and difference with parent (DWP). For example, the following comment describes the concept SpatialEntity in the travel ontology used for the EON2003 experiment [Isaac et al., 2003]:

```xml
<rdfs:comment>
  SWP=The entity has a spatio-temporal location
  SWS=The entity is considered mostly in regard of its spatial or temporal location
  DWS=The entity is considered mostly in regard of its spatial location
  DWP=The entity is considered mostly in regard of its spatial or temporal location : The entity is considered mostly in regard of its spatial location
</rdfs:comment>
```

b) Extend RDF(S) ontology component descriptions with properties not defined in RDF Schema.
RDF is based on the semantic networks KR paradigm and allows using any label for the graph arcs. These labels do not necessarily belong to the RDF or the RDF Schema vocabularies. Hence we can define the knowledge that could be lost in the translation with specific properties attached to the RDF(S) class and property definitions.

For example, Protégé-2000 implements this solution in its RDF(S) backend. Protégé-2000 users can decide whether exporting their ontologies to plain RDF(S) or to a Protégé-specific RDF(S) extension. With the first export format, they lose the knowledge that cannot be represented directly in plain RDF(S): property cardinalities, PAL constraints, etc. With the second one, part of the ontology is implemented in plain RDF(S) and the other part is preserved with that extension, as described in [Noy et al., 2001].

For instance, the attribute engine-count is exported to Protégé-specific RDF(S) as follows (the a namespace used in a:maxCardinality corresponds to the URI http://protege.stanford.edu/system#9):

```xml
<rdf:Property rdf:about="#engine-count"
  a:maxCardinality="1"
  rdfs:label="engine-count">
  <rdfs:domain rdf:resource="#Vehicle"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdfs:Property>
```

This case is hence similar to the KIF case: interoperability with RDF(S) can be easily achieved using the same source and target formats, but it becomes difficult when using different source

---

9 The RDF(S) backend for Protégé-2000 does not take into account some of the last modifications of RDF(S), such as the use of XML Schema datatypes. Hence the type integer is transformed to &rdfs;Literal.
and target formats. For example, KAON provides two different RDF(S) import services: one for plain RDF(S) and another for Protégé RDF(S). In summary, we move back from the situation with $O(n)$ translators to the situation where we needed $O(n^2)$ translators, hence losing the main advantages of using a common interlingua.

4. Current translation systems do still have many errors when exporting and/or importing RDF(S). RDF and RDFS have been created rather recently, and they have evolved during their existence. Besides, there has been evolution in related technologies in which RDF and RDFS are based, such as XML namespaces, XML Schema datatypes, etc. As a consequence, some of the existing RDF(S) import and export services are not stable enough yet.

For example, Protégé-2000 uses old namespaces to refer to the RDF and RDFS namespaces [Corcho et al., 2003a; page 108], and transforms attribute datatypes incorrectly, as shown in the previous example.

We can give some more examples regarding the translation of concept and property identifiers [Isaac et al., 2003, Corcho et al., 2003a]. For instance, some translation systems use an incorrect XML encoding for the RDF(S) files when any of the identifiers inside the ontology uses letters with accents; some of them use digits at the beginning of identifiers, which produce incorrect XML URI identifiers, or non allowed symbols like &, ;, etc.

In summary, RDF(S) shares most of the problems that we identified for KIF: from a pragmatic point of view, there are many possible ways to represent the same knowledge, and consequently not all the ontology translation systems developed for RDF(S) are able to manage correctly the knowledge generated in this language by other tools. From a semantic preservation point of view, the expressiveness of RDF(S) is very limited. However, its knowledge model is extensible, and hence any knowledge can be represented in it, with the problems already identified. Finally, RDF(S) also poses problems in the syntactic level, since it allows using different syntaxes to represent the same information, as shown above.

5.2.3 OWL as an interchange format

Although OWL was not proposed as a W3C Recommendation until very recently (February 2004), there are already some early-adopter tools able to manage OWL ontologies. In contrast to RDF(S), OWL-aware tools are primarily ontology tools and not general ones. The reason for this is that OWL is more complex than RDF(S), whose expressiveness is enough for most of the knowledge exchange needs of the other types of tools.
As occurred with RDF(S), some OWL-aware tools do still show some minor problems when managing this "young" language, especially related to namespaces, concept and property identifiers, etc. [Isaac et al., 2003].

As we have done with KIF and RDF(S), we will summarize below the main aspects related to the use of OWL as an exchange language:

1. **OWL allows representing the same knowledge in different ways, making knowledge exchange difficult.** OWL is built on top of RDF(S), which allows using different syntaxes to express knowledge. Therefore, OWL suffers from the same interoperability problem that we described for RDF(S). As in the case of that language, this problem can be solved with the use of OWL-aware APIs and developer tools that abstract the syntax of OWL ontologies.

2. **OWL Lite and OWL DL are not expressive enough to represent some of the knowledge that can be expressed in other ontology languages and tools.** OWL Lite has limited expressiveness: it adds some primitives to RDF(S), such as cardinality restrictions (restricted to the numbers 0 and 1), object and datatype properties, different types of range restrictions, etc. OWL DL adds some more primitives to OWL Lite, such as disjointness between classes, union of classes, complement of classes, etc.

   However, the knowledge that we can represent with these two sublanguages is restricted so as to ensure the existence of decidable reasoning engines for them. Hence they are not expressive enough to represent all types of first order logic axioms, which are part of the knowledge models of tools like OntoEdit, Protégé-2000, etc., and of languages like Ontolingua, OCML, F-Logic, etc.

The OWL layered approach allows using OWL Lite and/or OWL DL to express most of the knowledge of the ontology. It ensures an easy ontology exchange between languages and/or tools, provided that the minor problems commented at the beginning of this section are solved. For instance, the attribute engine-count would be represented in OWL (both Lite and DL) as follows:

```xml
<owl:DatatypeProperty rdf:ID='engine-count'>
  <rdf:type rdf:resource='&owl;FunctionalProperty'/>
  <rdfs:domain rdf:resource='#Vehicle'/>
  <rdfs:range rdf:resource='&xsd;integer'/>
</owl:DatatypeProperty>
```

3. **Part of the OWL translations must be written with regard to a specific format.** To overcome the previous knowledge losses and ensure knowledge preservation in transformations, the OWL Full sublanguage can be used. OWL Full uses the full potential of RDF(S), which is not permitted in OWL Lite nor OWL DL: it allows using metaclasses.
Another possible extension is the use of OWL annotation properties to represent this additional knowledge. Annotation properties are not used by OWL inference engines and so they do not have an impact in the reasoning properties of OWL ontologies.

However, as occurred with RDF(S), in both cases the extensions used to represent this additional knowledge depend strongly on the source format, and are not easy to "understand" by the import services of other tools or by ontology translation systems to other languages.

In summary, OWL is more expressive than RDF(S). Consequently, it allows representing in a standardised way much more knowledge than RDF(S) allowed, and hence the number of problems related to the pragmatic and semantic levels is reduced. However, it is still less expressive than many other tools, and its extensibility facilities have to be used in order to avoid losing knowledge in transformations. However, these translation solutions are created ad-hoc for specific source or target formats, and hence using OWL without losing knowledge is difficult. Related to the syntax layer, OWL has the same problems than RDF(S), although this problem is usually ameliorated by the presence of APIs that abstract the syntax variants of the language.

5.3 DIRECT ONTOLOGY TRANSLATION

As shown in the previous section, one of the main advantages of using common interchange formats for translating ontologies between two formats is that they do not usually require additional implementation efforts. This is due to the fact that in many cases there are already ontology translation systems available from the source to the interchange format, and vice versa. However, we have shown in the previous section that translations through interchange formats do not usually maintain the semantics nor the pragmatics of the original knowledge.

For this reason, in some cases it may be better to implement an ad hoc translation system between the source and the target formats, following a peer-to-peer translation architecture, as described in section 2.5. Even though the implementation effort is higher than with the previous pivot approach, the main advantage of such a system is that it can take into account the exact translation needs required, driven by the semantic and pragmatic preservation properties needed. Consequently, the results obtained with direct transformation approaches are usually better than those obtained with indirect transformation approaches.

The following sections describe three types of direct (peer-to-peer) transformation approaches:
Section 5.3.1 describes direct transformations that are mainly driven by pragmatic preservation, but not by semantic preservation. They consist in transforming only the knowledge that can be expressed with the ontology components available in the knowledge model of the target format. With this approach, the translation system does not use any knowledge model extensibility facility that the target format may provide, such as metaclasses, annotation properties, etc.

Section 5.3.2 describes direct transformations that are mainly driven by semantic preservation, but not by pragmatic preservation. They consist in creating the knowledge representation of the source format in the target format, as if it was a domain ontology of the target format. Then, ontologies in the source format are translated by creating instances of the KR ontology in the target format.

Section 5.3.3 describes a hybrid approach that is driven by both semantic and pragmatic preservation. It consists in transforming the ontology components of the source ontology that have a direct correspondence with other ontology components in the target format. The rest of knowledge is transformed to the target format by using its extensibility facilities, such as metaclasses, annotation properties, etc.

For each of these direct transformation approaches we have implemented an ontology translation system between the ontology tools WebODE and Protégé-2000, and vice versa. Their main features will be described in each section.

5.3.1 Direct translation without additional components

This direct transformation approach is the one more widely used by existing ontology translation systems. It consists in transforming only the ontology components of the source format that have a direct correspondence with other ontology components of the target format, without using any additional components or workarounds for the source ontology components that cannot be directly transformed. As it will be presented below, this direct approach allows preserving the pragmatics of the ontology transformed. However, the ontology semantics sometimes is not preserved.

To implement such an ad hoc direct translation system, we must first analyse in depth which target ontology components correspond to the source ontology components. Once that the mappings between components have been proposed, the transformation process consists in translating each source component to the corresponding one(s) in the target format.

Let us analyse this approach with respect to the preservation of semantics and pragmatics in the transformation:
With regard to the first aspect (semantic preservation), we can conclude that some knowledge is usually lost in the translation process, unless the source and target formats are very similar. The knowledge lost corresponds to those ontology components that cannot be transformed directly to the target format.

With regard to the second aspect (pragmatic preservation), the main conclusion achieved is that the knowledge transformed is usually legible in the target format, because the components used to represent the target ontology are those belonging to the standard knowledge model of the target format. Obviously, the knowledge that has not been transformed cannot be analysed from a pragmatic point of view, since it is not present in the target format.

In cyclic transformation processes, the implications of this translation approach are clear: the knowledge finally obtained is the one that can be represented in both the source and the target formats. As a consequence of the first transformation, some knowledge will be lost, as described above. The knowledge lost corresponds to the ontology components of the source format that have not direct correspondences with other ontology components in the target format. In the transformation back to the original format, there will be no knowledge losses, since the ontology components that are represented in the target format are only those that had a direct correspondence in both formats.

5.3.1.1 Practical example: from WebODE to Protégé-2000, and vice versa

We have created an ontology translation system that transforms WebODE ontologies to Protégé-2000 ontologies according to the standard knowledge model of Protégé-2000. This means that, in a coarse-grained view, WebODE concepts are transformed into Protégé-2000 classes, WebODE exhaustive decompositions and partitions are transformed to Protégé-2000 class taxonomies and abstract classes, WebODE instance attributes and ad hoc relations are transformed to Protégé-2000 template slots, WebODE class attributes are transformed to Protégé-2000 own slots by means of template slots and metaclasses, and WebODE formal axioms are transformed to Protégé-2000 PAL constraints. The transformations for the Protégé-2000 to WebODE translation system are the inverse of these transformations.

As a result of the transformation of a WebODE ontology to Protégé-2000, the ontology obtained contains all the previous components, but losses completely the knowledge about synonyms and acronyms, bibliographic references, and disjoint knowledge in class taxonomies. In other words, there is not full semantic preservation. Besides, some small details are also lost with regard to some attribute datatypes (WebODE URL datatypes and XML Schema datatypes are
transformed to the Protégé-2000 *Symbol* datatype), to algebraic properties of relations (such as symmetry, transitivieness, etc.), to the existence of multiple natural language documentations for attributes with the same name, etc. In the transformation back to WebODE, all the knowledge successfully exported to Protégé-2000 can be transformed back, and the knowledge that was lost is not recovered.

With regard to pragmatics, end users and applications can easily “understand” the ontology obtained, since it uses the standard knowledge model of Protégé-2000: it contains Protégé-2000 classes, slots, PAL constraints, etc. The same applies to the ontology obtained in WebODE as a result of a cyclic transformation.

Figure 5.6 shows a snapshot of the result of transforming the WebODE travel ontology developed for the EON2002 workshop to Protégé-2000 following this proposal. The figure shows that the concepts hotel1Star, hotel2Star, hotel3Star, hotel4Star and hotel5Star, which form a partition of the class hotel in WebODE, are transformed into Protégé classes that are subclasses of the class hotel. Besides, the class hotel is defined as an abstract class in Protégé-2000, that is, it cannot have direct instances. The figure also shows that the metaclass hotelTypes has been created. This metaclass has the template slot numberOfStars attached, which is converted to an own slot in the classes hotel1Star, hotel2Star, etc., with its corresponding values for each of them (1, 2, etc., respectively).

---

10 http://km.aifb.uni-karlsruhe.de/eon2002
The previous example is one of the possible approaches for transforming directly a WebODE ontology to Protégé-2000, and vice versa, using their standard knowledge models. Obviously, there are more possibilities that could also make use of the standard knowledge models of both ontology tools while making transformations easily understandable for WebODE and Protégé-2000 users.

For instance, since the knowledge models of both tools are based on a combination of frames and first order logic, much knowledge can be expressed by means of first order logic axioms. As an example, the previous system would allow creating a class that is a direct or indirect subclass of the classes hotel1Star and hotel2Star, which is not correct because those classes are disjoint in the source ontology. To avoid this, a WebODE partition can be transformed not only by creating all the subclass of relationships between the classes in the partition and the superclass, and declaring the superclass as abstract, but also adding PAL constraints stating that all the classes in the partition are disjoint to each other. The following constraint would be added to the class hotel1Star:

\[(\text{forall } ?X)\]
This transformation does still use the Protégé-2000 standard knowledge model, although it creates a workaround to represent the disjointness between classes, which is something not directly available in Protégé-2000. However, we may consider that this axiom is still legible by Protégé-2000 users, and hence this type of transformation would fall inside the category of translation approaches described in this section.

5.3.2 Direct translation by instantiating the source format KR ontology in the target format

In this section we propose another direct (peer-to-peer) type of transformation where the opposite maximal preservation properties hold, that is, this approach will be able to preserve the semantics of the ontology transformed but not its pragmatics.

This approach consists in creating first the KR ontology of the source format in the target one, with the standard knowledge modelling components of the target format, that is, with its concepts or classes, its relations, slots, roles or properties, its formal axioms, etc. Basically, this case consists in developing that KR ontology as if it was a domain ontology of the target format. Once that this KR ontology has been created, the translation process consists in creating instances of that KR ontology according to the contents of the ontology to be transformed. For instance, a concept in the source format is transformed into an instance of the class Concept in the target format; an instance in the source format is transformed into an instance of the class Instance in the target format, and is connected to one or several instances of the class Concept by means of the relation isInstanceOf; etc.

In summary, the only translation decisions taken into account in this transformation are related to the lexical and syntax levels of the ontology translation model described in chapter 4. No translation decisions have to be taken in the semantic or the pragmatic layers. Therefore, the transformation process is extremely simple.

As commented above, the main advantage of this translation approach is that it allows preserving the semantics of the original ontology: all the knowledge inside all the original ontology components is transformed completely to the target format. Besides, this approach ensures knowledge preservation in cyclic transformations. That is, if the ontology obtained is transformed back to the source format, we will finally be able to obtain exactly the same ontology that we had originally.
With regard to pragmatics this approach obtains bad results. The KR ontology is created by means of the standard knowledge modelling components of the target format (classes or concepts; relations, roles, slots or properties; formal axioms, etc.), and all the knowledge of the original ontology is transformed to instances of the components represented in that KR ontology. The result is that the ontology transformed to the target format is not legible, neither for human users nor for systems.

However, in cyclic transformations an interesting preservation property arises: the original ontology can be recovered completely when it is transformed back from the target format. As a consequence, we can say that even though the pragmatics of the ontology is not preserved in the transformation to the target format, it is preserved in cyclic transformations.

In summary, this approach does not make a real translation of knowledge from one format to another. Basically, it only performs a transformation in the lexical and syntactic layers, avoiding the difficult task of semantic and pragmatic translation. Another drawback of this approach is that the number of ontology translation systems to be developed is doubled. For example, to perform cyclic transformations between two formats, four translation systems are needed: from L1 to L2, and from L2 to L1, for cyclic transformations starting in L1; and from L2 to L1, and from L1 to L2, for cyclic transformations starting in L2.

### 5.3.2.1 Practical example: from WebODE to Protégé-2000, and vice versa

To illustrate the ontology translation approach presented in this section, we have implemented another ontology translation system between WebODE and Protégé-2000, and vice versa, starting in WebODE. We have created the WebODE KR ontology by means of Protégé-2000 classes and slots.

This KR ontology contains 14 classes, which represent the ontology components of the WebODE knowledge model: :WebODEComponent, :Concept, :Attribute, :InstanceAttribute, :ClassAttribute, :AdHocRelation, :Axiom, :Instance, :Group, :Constant, :Property, :Synonym, :Acronym, and :Reference. The KR ontology also contains 36 slots that represent the relationships between the previous components: :name, :documentation, :hasAttribute, :hasDomain, :hasRange, :datatype, :maxCardinality, :minCardinality, :minValue, :maxValue, :measurementUnit, :precision, :hasValues, :isSubclassOf, :isDisjointDecompositionOf, :isExhaustiveDecompositionOf, :isPartitionOf, :hasFormalExpression, :isInstanceOf, etc. Finally, the ontology contains 32 PAL constraints that model restrictions of the WebODE knowledge model.
The figure 5.7 shows a screenshot of the WebODE KR ontology modelled in Protégé-2000. The figure shows the details of the :InstanceAttribute class, the slots inherited from its superclasses :WebODEComponent and :Attribute (:name, :documentation, :datatype, :minCardinality, :maxCardinality, :minValue, :maxValue, :measurementUnit, and :precision), and the slot attached to it: :hasValues.

Once that this KR ontology has been created in Protégé-2000, the following step consists in transforming the WebODE domain ontology components in instances of the classes and slots of the KR ontology already created. Therefore, the concepts of the original ontology are transformed to instances of the class :Concept, its instance attributes are transformed to instances of the class :InstanceAttribute, and attached to the corresponding instance of :Concept with the :hasAttribute slot, etc.

Figure 5.8 shows how all the concepts from the source ontology have been transformed into instances of the Protégé-2000 class :Concept, and shows also the details of the concept accommodation, with its documentation, the classes of which it is a subclass, etc. As explained above, the main drawback of this approach is related to pragmatic preservation. In this example, Protégé-2000 users (be them humans or other information systems) will not be able to use easily the resulting ontology. For instance, not even the class taxonomy of the original ontology is shown in Protégé-2000.
A declarative approach to ontology translation with knowledge preservation

Following this approach the inverse transformation (from the ontology generated in Protégé-2000 back to WebODE) is also a simple task. First of all, it is not necessary to create any KR ontology in WebODE, since the translation system will transform the ontology back to the WebODE standard knowledge model. Then, the transformation process would consist again in executing mappings at the lexical and syntax layers.

In the case that the source format was Protégé-2000, and we wanted to create ontology translation systems to WebODE, and from WebODE to Protégé-2000, the process needed is similar to the one explained in this section. First, the Protégé-2000 KR ontology can be created in WebODE by means of WebODE concepts, instance and class attributes, relations, and formal axioms. Then, the lexical and syntax mappings between Protégé-2000 and the Protégé-2000 KR ontology in WebODE are executed. Finally, to obtain the source ontology back again, new transformations at the lexical and syntax layers must be proposed between the Protégé-2000 KR ontology in WebODE and Protégé-2000.
5.3.3 Direct translation only with the necessary additional components of the source KR ontology

The high level conclusions that can be extracted from the descriptions of the previous two direct (peer-to-peer) translation approaches is that either they are normally able to preserve semantics or pragmatics in the transformation, but not both of them at the same time:

- In the direct approach of section 5.3.2, the source format KR ontology is implemented in the target format by means of the standard knowledge modelling components of the target format, and the ontology is transformed by creating instances of that KR ontology. As described in that section, there is no real translation between both formats, but merely lexical and syntax transformations. Consequently, there are pragmatic preservation problems in the target format, which avoid using the ontology translated in other contexts (either by other information systems or by human users).

- In contrast, the direct approach of section 5.3.1 performs a real translation at all translation layers. All the ontology components of the source format with a direct correspondence with components of the target format are translated, and the rest of components are lost in the transformation. Hence, the ontology is legible in the target format (pragmatic preservation), but there are knowledge losses (no semantic preservation).

Obviously, these transformations describe extreme cases that do not necessarily have to be followed strictly, that is, there can be middle (hybrid) approaches that combine their advantages. In this section, such a hybrid approach is presented: it will be able to preserve in the transformation most of the ontology semantics and of the ontology pragmatics.

This approach proposes to transform as much knowledge as possible using the components of the target format’s standard knowledge model (as described in section 5.3.1), so that the knowledge transformed can be easily understood and dealt with by human users and applications. Besides, it proposes to manage those pieces of knowledge that have not been transformed, so that they can be recovered in case that the ontology is transformed back to the original format (as described in section 5.3.2). This proposal avoids mixing the domain ontology components transformed with the knowledge modelling components used for semantic preservation.

To achieve this twofold objective, this approach proposes to implement part of the source format’s KR ontology in the target format, by means of meta-knowledge (which is usually expressed with metaclasses, annotation properties, etc.). The KR ontology implemented contains only the ontology components that cannot be represented directly in the standard
knowledge model of the target format, so that they are only used in case that some knowledge of
the original ontology cannot be directly transformed to the target format.

In summary, following this approach the translation of an ontology consists in performing the
following steps:

- Create part of the KR ontology of the source format in the target format, by means of meta-
knowledge. This part of the KR ontology models the knowledge modelling components of
the source format that have not a direct correspondence with the knowledge modelling
components of the target format. Meta-knowledge is usually represented by means of
metaclasses (as described by [Dahchour et al., 2001], one of the applications of metaclasses
is to define formalisms or development methods within another system). However, this does
not prevent us from using other elements, such as annotations (like OWL annotation
properties), structured natural language documentations, etc.

- Whenever there is a direct correspondence between the source format’s ontology
components and the target format’s ones, transform the component of the original ontology
to its corresponding components in the target format.

- Transform all the knowledge that was not transformed previously, according to the partial
KR ontology created in the first step.

This approach requires that the target format provides some means of representing the needed
fragments of KR ontologies in such a way that the domain ontology components transformed
are not mixed with the knowledge modelling components used for semantic preservation. This
way, we will be able to preserve most of the semantics and pragmatics of the ontology
transformed.

Some target formats, such as the ontology tools Protégé-2000 and WebODE, and the ontology
markup languages RDF(S) and OWL, provide interesting possibilities for implementing this
approach, as described below:

- Protégé-2000 allows representing metaclasses. Metaclasses can be used to create the
additional knowledge modelling components needed, so that the semantics can be
preserved in the transformation. Besides, the Protégé-2000 user interface can be easily
configured, so that the previous additional KR ontology components can be hidden both
from the class taxonomy of the ontology and from the forms used to created domain
ontology components. This is useful to preserve the ontology pragmatics in the
transformation, because the domain ontology components are not mixed with the other
components used for semantic preservation.
- WebODE allows representing metaclasses, as shown in [Fernández-López and Gómez-Pérez, 2002], and importing components from other ontologies. These KR features allow creating two parallel versions of the same ontology, one with the knowledge directly transformed to the standard knowledge model of WebODE, and the other one with the additional knowledge not transformed. In this way, the domain ontology components are not mixed with the other components used for semantic preservation.

- OWL allows attaching annotation properties to any ontology component. Annotation properties do not have an influence in the semantics of the ontology transformed, and can be used to store any extra knowledge that cannot be represented in the standard knowledge model of the language. Besides, OWL Full allows representing metaclasses.

- RDF(S) allows representing metaclasses, which have the advantages already commented above.

With regard to the number of translation systems needed to perform cyclic transformations, this approach has the same drawbacks than the one presented in section 5.3.2: it doubles its normal number. However, there is a difference with respect to that approach. If no specific translation system is developed between the language L2 to L1, but just a general one using the approaches presented in sections 5.2 or 5.2.1, part of the ontology will be at least transformed, while in the previous approach the transformation could not be done at all.

5.3.3.1 Practical example: from WebODE to Protégé-2000, and vice versa

To illustrate the ontology translation approach presented in this section, we have implemented another ontology translation system between WebODE and Protégé-2000, and vice versa. The objective of this translation system is to reuse as much as possible the standard knowledge model of Protégé-2000, so that the resulting ontology is easy to understand for Protégé-2000 users (either human users or systems), and use the extensibility facilities provided by Protégé-2000 to maintain the rest of knowledge that cannot be directly translated.

Therefore, we have created part of the WebODE KR ontology in Protégé-2000, as shown in figure 5.9, by extending the Protégé-2000 standard metamodel. The following metaclasses have been created: :WebODEConcept, :WebODESlot, :WebODEAttribute, :WebODEAdHocRelation, :WebODEPredefinedSlots, and :WebODEPredefinedFacet. Besides, the following classes have been defined as subclasses of the class :PAL-CONSTRAINT, which is used to generate all the PAL constraints in Protégé-2000: :WebODEDisjointConstraint, :WebODEAxiom, and :WebODEPropertyConstraint. Furthermore, subclasses of the predefined class :THING have
been used to represent some other components: :WebODEReference, :WebODESynonym, :WebODEAbbreviation, :WebODEConstant, and :WebODEProperty.

The figure shows the details of the metaclass :WebODEConcept, which extends the metaclass :STANDARD-CLASS, used by default for creating Protégé-2000 classes. The extension consists in some new WebODE specific slots that can store information about classes that is not represented in the standard knowledge model of Protégé-2000, such as information about transitive and intransitive parts of, about not subclass of, and bibliographic references, abbreviations, and synonyms. Besides, two other slots are attached to this metaclass, which are used also to ensure that in cyclic transformations the original ontology can be recovered: the original URI of the concept in WebODE and its original name, since the class name may have suffered from lexical transformations in the transformation process to Protégé-2000. The rest of information reuses the standard slots for standard classes in Protégé-2000.

![Figure 5.9. Partial WebODE KR ontology represented in Protégé-2000 mainly by means of metaclasses.](image)

This ontology is created as a separate ontology that is imported by every ontology that results from the application of the ontology translation system created. This translation system works as follows:

- First, it transforms to Protégé-2000 all the WebODE ontology components that have a direct representation in the standard knowledge model of Protégé-2000.
Second, it transforms the rest of the WebODE ontology components that have not a direct representation in the previous knowledge model. This transformation consists in creating instances of the WebODE KR ontology described above, hence preserving the knowledge that could have been lost in the translation when the ontology is transformed back to WebODE.

Figure 5.10 shows a screenshot of the travel ontology translated using this approach. In this example, the metaclass :WebODEConcept has a subclass hotelTypes, which is consequently another metaclass, used as the metaclass for the classes hotel1Star, hotel2Star, hotel3Star, hotel4Star, and hotel5Star, since they all have an own slot numberOfStars whose value is 1, 2, 3, 4, and 5, respectively. The rest of classes in this ontology are instances of the metaclass :WebODEConcept.

The figure shows that, in contrast with the solution presented in section 5.3.2, the classes and class taxonomy of the ontology can be easily seen and understood. The figure shows the details of the class hotel1Star, with its template slots and the own slot numberOfStars, which come from the instance attribute and class attribute definition of WebODE. There are also other own slots that do not appear commonly in Protégé-2000 ontologies: :ORIGINAL-URI, :WEBODE-NAME, :SYNONYMS, :ABBREVIATIONS, etc. These slots are used to include the extra knowledge from WebODE, as we discussed before.
A declarative approach to ontology translation with knowledge preservation

The main drawback of the previous screenshot is that an end-user who is not acquainted with the knowledge model of WebODE will not be able to understand what the terms :WebODEReference, :WebODESynonym, etc., mean, since this solution is merging the modeling of the domain ontology with the modeling of the KR ontology. We need a last step in this process, to ensure that the new components related to the WebODE KR ontology are not shown to Protégé-2000 users.

We use the hiding-class functionality provided by Protégé-2000, and remove some components from the forms showed to users. Consequently, end users do not have to learn anything about how to model ontologies with WebODE and can use the ontology using the common modeling conventions provided by Protégé. In the transformation back to WebODE, this hidden knowledge can be transformed without problems. Figure 5.11 shows the result of configuring Protégé-2000 to hide these classes and the forms shown for classes.
Figure 5.11. Result of a direct translation from WebODE to Protégé-2000 using the standard knowledge model of Protégé-2000 and part of the WebODE KR ontology, hiding classes and customising forms.

In summary, this proposal meets the two objectives that we had in mind at the beginning of this section: preserving all the knowledge in the transformation and allowing end users and applications to understand easily the resulting ontology.

Similarly, the same kind of solution can be applied when transforming ontologies from Protégé-2000 to WebODE, as was briefly explained previously: the new components are not shown to WebODE users since they are instances of the WebODE KR ontology and are not accessed directly from the ontology that is result of the transformation. However, they are also available for performing the transformation back to Protégé-2000.

The inverse case (translating from Protégé-2000 to WebODE) is equivalent to this one: we should create the Protégé KR ontology in WebODE, as explained in [Fernández-López and Gómez-Pérez, 2002] for the implementation of metaclasses for the ODEClean service, and transform all the components of the Protégé ontology according to it.
5.4 COMPARISON OF TRANSFORMATION APPROACHES

Once we have described in detail the four transformation approaches proposed in this section, we will compare them according to two different sets of parameters: preservation properties of the semantics and pragmatics of the ontology transformed, and required implementation effort of the approach.

Figure 5.12 shows the comparison of these approaches according to the first set of criteria. The horizontal axis represents the amount of semantic knowledge preserved in the transformation, while the vertical axis represents the amount of pragmatic knowledge preserved in the transformation. We do not aim at providing quantitative measures about the amounts of semantic and pragmatic knowledge preserved in the translation process, but we just provide approximate qualitative measures for them. Besides, not all the source and target format pairs behave in the same way with respect to this preservation.

As shown in the figure, the second and third approaches have extremely different behaviours, since the former is devoted to preserve pragmatics, with no guarantees about the semantics of the knowledge transformed, and the latter is devoted to preserve semantics, with even less guarantee about the preservation of the pragmatics of the ontology transformed. Hence, the most interesting approach seems to be the fourth one, which can maintain most of both properties, while for the indirect translation through common interchange languages the results are less precise and will depend on the quality and standardization of the transformations to the interlingua.
With regard to the implementation effort required to carry out each approach, the results are different. The use of common interchange formats does not usually require much implementation effort, since in most of the cases there are already ontology translation systems implemented for such tasks. With respect to the second and third approaches, they require more or less the same amount of effort: the effort required in the second approach is mainly related to the identification of mappings between the knowledge components of the source and target formats and the implementation of such mappings, while in the third approach the effort is mainly related to the creation of the source KR ontology in the target format, while the transformation is usually easy (it has to be done only in the lexical and syntax layers). Finally, the approach that requires more effort is the fourth one, since it has to combine the two previous ones. However, in general it provides better preservation properties than the other ones.
Chapter 6
Experimentation

This chapter describes two sets of experiments made to evaluate the contributions of this thesis. The first set of experiments consists in the creation of several ontology translation systems (import and export services) between the WebODE ontology engineering platform and several ontology languages and tools, as well as other general-purpose languages. Some of them have been used to obtain the ontology translation model described in chapter 4. Others have been used to evaluate the use of such ontology translation model. The second set of experiments presents some results and comparisons of the application of the different ontology translation approaches described in chapter 5.

6.1 ONTOLOGY TRANSLATION MODEL

The ontology translation model described in chapter 4 has been derived from our experience in the development of ontology translation systems between the ontology engineering workbench WebODE and different ontology and general-purpose languages, as follows:

- **Description logic ontology languages.** We have developed WebODE export services to DAML+OIL (based on SHOIQ), OIL (based on SHIQ), and XCARIN (an XML version of the CARIN language, which is based on ALN). Besides, we have also developed WebODE import services from the languages DAML+OIL and XCARIN.

- **Frame-based ontology languages.** We have developed the WebODE export service to FLogic, based on a combination of frames and first order logic.

- **General-purpose programming languages.** We have developed the WebODE export service to Java, converting the ontology concepts to Java beans.

- **Markup languages.** We have developed the WebODE export and import services to and from XML, using a DTD that resembles the WebODE knowledge model.
Once we generated the model based on our experience, we have evaluated its goodness and refined it by constructing the ontology translation systems between WebODE and RDF(S), DAML+OIL, OWL, and Protégé-2000, and vice versa. All these translation systems have been successfully integrated as export and import services of the WebODE platform and are currently available for WebODE users. The details of these export and import services are as follows:

- **WebODE export and import services to and from RDF(S).** The RDF(S) language is based on a combination of semantic networks and frames, and has been developed from scratch. With respect to the translation approach followed for these export and import services, the construction of these ontology translation systems is mainly driven by pragmatic preservation, and we have decided to perform a direct transformation without any additional components to preserve the semantics of the ontology transformed (second approach). We have decided to use this approach so as to allow a better interoperability with other ontology tools and ontology-based applications, which would be highly compromised if we used any of the other approaches.

  From a coarse-grained perspective, WebODE concepts are transformed to RDFS classes, WebODE attributes and ad hoc relations are transformed to RDF properties, and WebODE instances are transformed to RDF instances, and vice versa. However, much knowledge is lost in the transformation, such as cardinality constraints in properties, disjoint and exhaustive knowledge in class taxonomies, formal axioms, etc.

- **WebODE export and import services to and from OWL DL.** The OWL DL language is based on description logic, and has been constructed by reusing the export and import services to and from DAML+OIL, and by reusing the RDF export and import services for dealing with instances. The translation approach followed for these export and import services is also driven by pragmatic preservation, and we have decided to perform also a direct transformation without any additional components to preserve the semantics (second approach). As with RDF(S), this decision is made to allow a better interoperability with other ontology tools and ontology-based applications.

  From a coarse-grained perspective, WebODE concepts are transformed to OWL classes, WebODE attributes are transformed to OWL datatype properties, WebODE ad hoc relations are transformed to OWL object properties, and WebODE instances are transformed to RDF instances, and vice versa. Compared to the RDF(S) translation systems, the amount of knowledge lost in this transformation is much lower, since OWL is much more expressive than RDF(S).
- **WebODE export and import services to and from Protégé-2000.** The standard knowledge model of Protégé-2000 is based on a combination of frames and first order logic. These services have been developed from scratch. We have implemented these services according to all the ontology translation approaches described in chapter 5, although the service currently available has been developed with a hybrid approach, combining direct mappings between components and all the additional components needed in order to preserve knowledge in the transformation, by means of metaclasses and hidden classes (fourth approach). The details of this service are described in section 6.2.5.

Some of these import and export services have been developed in the context of R&D projects: the XCARIN ones were developed in the context of the EU project MKBEEM (IST-1999-10589), and the DAML+OIL ones in the context of the Spanish CICYT project ContentWeb (TIC-2001-2745).

The export and import services to and from RDF(S), OWL DL, and Protégé-2000 have been created in the context of the EU project Esperonto (IST-2001-34373). The objective of this project is to bridge the gap between the current Web and the Semantic Web: that is the reason why the W3C recommendations RDF(S) and OWL DL have been dealt with. Besides, the Protégé-2000 services have been created to allow both tools exchange their ontologies so as to permit using their varied and complementary ontology-based services and applications.

It is important to highlight that the transformations to be done among these formats are quite different to each other. As described above, the WebODE and Protégé-2000 knowledge models are based on a combination of frames and first order logic, OWL DL is based on description logic, and RDF(S) is based on semantic networks, although it provides some basic frame-based KR primitives. Hence the successful application of the ontology translation model for the development of these ontology translation systems shows that the model proposed in this chapter can be used in different contexts and for generating ontology translation systems with different degrees of difficulty.

All these services are available in the public server of the WebODE ontology engineering workbench¹, and have been widely used for uploading and implementing ontologies in these languages and in Protégé-2000. We have tested the correct behaviour of these export services with all the ontologies currently available in this server (215 ontologies of different sizes and complexities, and developed by different users), plus the set of synthetic ontologies from the benchmark suite described in [Corcho et al., 2003b]. Besides, 65 RDF(S) ontologies and 55

¹ [http://webode.dia.fi.upm.es/](http://webode.dia.fi.upm.es/)
OWL DL ontologies (also of different sizes and complexities, and developed by different users) have been successfully imported in the server by WebODE external users.

6.2 SEMANTIC AND PRAGMATIC PRESERVATION IN ONTOLOGY TRANSLATION

The second set of experiments consists in performing cyclic ontology translations between the ontology tools WebODE and Protégé-2000, and then comparing the results obtained at each step according to two parameters: semantic and pragmatic preservation. We have used a representative ontology for performing these experiments: the travel ontology built for the EON2002 workshop, constructed from a natural language definition in the travel domain, which aimed at showing how different tools can be used to build the same ontology.

In our experiments, we have taken into account the four ontology translation approaches described in chapter 5. For the first translation approach (indirect transformation through common knowledge interchange formats), we have used two different ontology markup languages, RDF(S) and OWL DL. The results for RDF(S) are described with more detail in [Corcho et al., 2003a]. In the next subsections we will present the results obtained in each experiment.

6.2.1 Experiment 1. Indirect ontology translation with RDF(S) as a common interchange format

This experiment consisted in performing a 4-step transformation of the travel ontology: (1) from WebODE to RDF(S), (2) from RDF(S) to Protégé-2000, (3) from Protégé-2000 to RDF(S), and (4) from RDF(S) to WebODE. Both tools already provided export and import services to and from the RDF(S) language. Therefore, no implementation effort has been needed to perform this experiment.

Table 6.1 shows a coarse-grained summary of the results obtained in this experiment. It presents the number of concepts available at each step, the number of subclass-of relationships, of disjoint and exhaustive decompositions of concepts, of concept partitions, of attributes and ad hoc relations, of formal axioms, of constants, and of concept instances. These ontology components will be also the ones used in the rest of experiments. The most representative results are highlighted in bold font, and commented below.
Table 6.1. Summary of the results obtained in the indirect translation between WebODE and Protégé-2000, with RDF(S) as a knowledge interchange format.

<table>
<thead>
<tr>
<th></th>
<th>WebODE</th>
<th>RDF(S) (step 1)</th>
<th>Protégé-2000 (step 2)</th>
<th>RDF(S) (step 3)</th>
<th>WebODE (step 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#concepts</td>
<td>62</td>
<td>62</td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>#subclass of</td>
<td>24</td>
<td>61</td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>#disjoint decompositions</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#exhaustive decompositions</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#partitions</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#attributes/relations</td>
<td>69</td>
<td>43</td>
<td>43</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>#axioms</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#constants</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#instances</td>
<td>20</td>
<td>20</td>
<td>22</td>
<td>22</td>
<td>20</td>
</tr>
</tbody>
</table>

As commented in section 6.1, the WebODE export service to RDF(S) does only transform the knowledge that can be represented in the standard knowledge model of this language. Concepts and instances are correctly transformed. The number of attributes and relations decreases because in RDF(S) their scope is global to the ontology, while in WebODE they depend on the concept where they are defined. The original ontology contains several attributes and relations with the same name defined in different concepts, and that is the reason why the number of global RDF properties decreases. Furthermore, their cardinality restrictions are lost, since RDF(S) cannot express such knowledge, and in the cases where a property was defined for two different concepts or with two different ranges, this information is also lost, since RDF(S) does not allow representing complex property domains and ranges. Besides, formal axioms and constants are lost. Similarly, disjoint and exhaustive knowledge in concept taxonomies is partially lost, and transformed into subclass-of relationships between concepts.

The second step is performed almost correctly by the Protégé-2000 import service from RDF(S). The classes and properties defined in RDF(S) are imported as Protégé-2000 classes and slots, the subclass-of relationships are also transformed correctly (although one extra relationship is generated from the root class in the class taxonomy to the predefined Protégé-2000 class :THING), and the instances are correctly transformed, although two new instances are created as a result of the values of some properties for some instances, which were resources.

Protégé-2000 gives support to two types of RDF(S) translation: plain RDF(S) and Protégé-2000 specific RDF(S). In the first case, the RDF(S) ontology resulting from the third translation step is similar to the one generated after the first one. The only differences between both files are related to the RDF syntax used in each case, and to the two previous new instances incorrectly generated. In the second case, the RDF(S) code contains also additional tags and attributes that
allow preserving all the knowledge available in the Protégé-2000 ontology. The following piece of code shows such additional information for the RDF property distanceToSkiResort, with the attributes a:maxCardinality, a:minValue, and a:range, which are Protégé-2000 specific:

```xml
<rdf:Property rdf:about="#distanceToSkiResort"
             a:maxCardinality="1"
             a:minValue="0.0"
             a:range="float"
             rdfs:comment="The distance from the hotel to a ski resort"
             rdfs:label="distanceToSkiResort">
  <rdfs:domain rdf:resource="#accommodation"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
```

Finally, the result of the last translation step is a WebODE ontology that contains all the concepts and instances available in the original ontology, and has lost all the formal axioms and constants. The disjoint and exhaustive knowledge in concept taxonomies has been lost, though transformed to simple subclass-of relationships between concepts. The properties that had a explicit domain defined are transformed into attributes or relations (depending on whether the range is a datatype or a class) attached to the corresponding concept. In contrast, the properties with no explicit domain defined are transformed into attributes or relations attached to the root concept of the taxonomy.

The table and the previous comments mainly summarize the results related to the preservation of semantics in the translation process. With regard to the preservation of the intended meaning of the ontology, the results obtained are better: the classes and the class taxonomy of the Protégé-2000 ontology are the same as those in the original one (except for the disjoint and exhaustive knowledge), and the instances are similar, except for the new ones created. With regard to the slots, two different attributes or relations with the same name in WebODE are only transformed to a single slot or relation if they have the same range. In another case, they are transformed to different slots or relations, with similar names. Finally, only some term names are different to the original one, but still easily understandable.

6.2.2 Experiment 2. Indirect ontology translation with OWL DL as a common interchange format

Like the previous experiment, this one consists also in four transformation steps, with OWL DL as a common interchange language. Both tools provided export and import services to and from OWL DL; hence no implementation effort has been needed to perform this experiment. We
must take into account that in the case of Protégé-2000 we had two options: using Protégé-2000 together with the OWL plugin and its own user interface, or using Protégé-2000 with the OWL plugin but the general-purpose Protégé-2000 user interface. We have selected the second option, since the general-purpose Protégé-2000 user interface is the one involved in the rest of experiments, and hence the results obtained here will not be biased by the interchange language used.

Table 6.2 shows a coarse-grained summary of the results obtained in this experiment, with the same knowledge components than in table 6.1, and with the most representative results highlighted in bold font.

Table 6.2. Summary of the results obtained in the indirect translation between WebODE and Protégé-2000, with OWL DL as a knowledge interchange format.

<table>
<thead>
<tr>
<th></th>
<th>WebODE</th>
<th>OWL DL (step 1)</th>
<th>Protégé-2000 (step 2)</th>
<th>OWL DL (step 3)</th>
<th>WebODE (step 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#concepts</td>
<td>62</td>
<td>62</td>
<td>62</td>
<td>62</td>
<td>66</td>
</tr>
<tr>
<td>#subclass of</td>
<td>24</td>
<td>61</td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>#disjoint decomp.</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#exhaustive decomp.</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#partitions</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#attributes/relations</td>
<td>69</td>
<td>43</td>
<td>43</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>#axioms</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>#constants</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>#instances</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

In the first transformation step, the figures presented in the table are similar to those obtained in the case of the RDF(S) language, except for the disjoint and exhaustive decompositions, and partitions. Disjoint knowledge is expressed by means of the `owl:disjointWith` primitive between all the pairs of concepts inside the disjoint decomposition or partition. Exhaustive knowledge is expressed by means of the `owl:unionOf` primitive (the superclass is defined to be equivalent to the union of all the classes in the exhaustive decomposition or partition. Besides, all the concepts in each decomposition or partition are defined as subclasses of their superclass.

However, there are other differences not summarized in the table, related to the higher expressiveness of OWL DL. For instance, OWL DL can represent property cardinalities, complex domains and ranges in properties, datatype properties and object properties, etc.

In the second transformation step, the disjoint and exhaustive knowledge in class taxonomies is lost, since it cannot be represented with the standard knowledge model of Protégé-2000. The rest of knowledge components are correctly transformed.

Consequently, the OWL DL ontology obtained as a result of the third step does not contain the knowledge that was lost in the previous transformation. However, since Protégé-2000 does not
provide specific components in its standard knowledge model for representing this kind of
knowledge, this knowledge would have not been transformed anyway.

The most interesting result that appears from the fourth translation process is the creation of
due new classes and formal axioms, which are interrelated. The classes correspond to anonym
classes that come from complex domains of datatype and object properties, and the formal
axioms define formally these complex domains. For example, the following formal axiom
defines the anonym class derived from the attribute number of Stars.

forall (?x) (OWL_ANONYM_CLASS_3(?x))<->
  ((hotel1Star(?x)) or (hotel2Star(?x)) or (hotel3Star(?x)) or
   (hotel4Star(?x)) or (hotel5Star(?x)))

From the pragmatic point of view, the results are similar to the ones obtained in the previous
step. The main differences are related to the transformation of disjoint and exhaustive
knowledge in concept taxonomies, as was the case for semantics. Disjoint decompositions are
transformed to rdfs:subClassOf relationships and owl:disjointWith relationships, which belong
to the OWL DL language, and consequently are easily understandable. In the case of the
transformation back to WebODE, the most representative aspects appear with the creation of
anonymous classes and formal axioms, since unions of classes cannot be easily expressed in
frame-based knowledge models. These classes and axioms make the resulting ontology more
difficult to understand in WebODE.

6.2.3 Experiment 3. Direct ontology translation without additional semantic
preservation components

The experiment shown in this section consists only in 2 translation steps: in the first one, the
ontology is transformed from WebODE to the standard knowledge model of Protégé-2000; in
the second step, that ontology is transformed back from Protégé-2000 to WebODE. The results
obtained are summarised in table 6.3. This experiment was also shown as a practical example in
section 5.3.1.

The implementation effort for this translation approach is bigger than in the previous indirect
approaches, since both translation systems have had to be developed.

Table 6.3. Summary of the results obtained in the direct translation between WebODE and Protégé-2000,
without additional components for semantic preservation.

<table>
<thead>
<tr>
<th></th>
<th>WebODE</th>
<th>Protégé-2000 (step 1)</th>
<th>WebODE (step 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#concepts</td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>#subclass of</td>
<td>24</td>
<td>61</td>
<td>61</td>
</tr>
</tbody>
</table>
The most relevant aspect to be commented for the first translation step rely on the transformation of disjoint and exhaustive knowledge in concept taxonomies. This knowledge is transformed into subclass-of relationships between classes and PAL constraints, which state either the disjointness between classes, the exhaustiveness of class decompositions or both (the use of PAL constraints was suggested in section 5.3.1, and has been implemented for this experiment). This is the reason why the number of formal axioms in Protégé-2000 is much higher than those available in the original ontology (37 PAL constraints represent this knowledge). The other 8 PAL constraints correspond to the 8 formal axioms defined in the original WebODE ontology, that can be transformed to PAL.

Other interesting aspects derived from the previous summary are the transformation of WebODE attributes and ad hoc relations into Protégé-2000 slots, which are also global to the ontology, as was the case with RDF(S) and OWL DL. Hence we obtain the same number of slots than we obtained in the previous experiments.

Other minor differences between the original ontology and the one obtained after the transformation are not shown in this summary. For example, the knowledge about precision and measurement units of attributes is lost, since it cannot be represented in the standard knowledge model of Protégé-2000, some datatypes are transformed and others are lost (e.g., XML Schema datatypes), the documentation attached to formal axioms is lost, etc.

In the transformation back from Protégé-2000 to WebODE, all the original knowledge is obtained back, except for the knowledge that was already lost in the first translation step. For example, all the concepts and instances are obtained back, and the attributes and ad hoc relations are correctly distinguished and transformed (hence moving from 43 slots to 69 attributes and ad hoc relations). The disjoint and exhaustive knowledge is also transformed back, but instead of using the corresponding WebODE KR primitives, it is transformed back as formal axioms. The reason for this strange behaviour is that the translation service from Protégé-2000 to WebODE is not robust enough to identify the formal axioms that deal with disjoint and exhaustive knowledge and convert them into disjoint and exhaustive decompositions, or partitions. Although such an identification can be done, it would require a large additional implementation effort, which would be only useful in the case that the ontology...
has been transformed to Protégé-2000 from WebODE. Similar situations were described in chapter 5 with KIF and RDF(S) expressions.

As a consequence of these results, we can say that the translation has preserved most of the semantics of the ontology transformed, except for some components and parts of them. We can also state that most of the pragmatics have been preserved in the transformation, except for the case of disjoint and exhaustive knowledge in concept taxonomies. This kind of knowledge is transformed back to formal axioms in WebODE, hence making it more difficult to understand what they represent in WebODE, where there are other specific primitives to represent that knowledge. Some other differences appear related to the differences in the scope of attributes and ad hoc relations, and slots in both tools, respectively. They have been already commented in the previous experiments, since RDF(S) and OWL DL gave also the same treatment to these knowledge components.

6.2.4 Experiment 4. Direct ontology translation instantiating the WebODE KR ontology in Protégé-2000

This experiment consists also in 2 translation steps: in the first one, the ontology is transformed from WebODE to Protégé-2000 by instantiating the WebODE KR ontology that has been implemented in Protégé-2000; in the second step, that ontology is transformed back from Protégé-2000 to WebODE. The results obtained are summarised in table 6.4. This experiment was also shown as a practical example in section 5.3.2.

The implementation effort for this translation approach is lower than in the previous direct approach, since it mainly consists in performing transformations at the lexical and syntax layers. In contrast, the previous approach considered transformations in the four layers identified in chapter 4.


<table>
<thead>
<tr>
<th>#concepts</th>
<th>WebODE</th>
<th>Protégé-2000 (step 1)</th>
<th>WebODE (step 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#subclass of</td>
<td>24</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>#disjoint decompositions</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>#exhaustive decompositions</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>#partitions</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>#attributes/relations</td>
<td>69</td>
<td>36 (predefined)</td>
<td>69</td>
</tr>
<tr>
<td>#axioms</td>
<td>8</td>
<td>32 (predefined)</td>
<td>8</td>
</tr>
<tr>
<td>#constants</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>#instances</td>
<td>20</td>
<td>179</td>
<td>20</td>
</tr>
</tbody>
</table>
As commented in section 5.3.2, the first step to be done in this approach, before making any ontology transformation, is to create the KR ontology of the source format in the target one. In this specific case, the transformation consists in creating the WebODE KR ontology in Protégé-2000, by means of Protégé-2000 classes and slots. This KR ontology contains 14 classes and 36 slots, together with 32 PAL constraints that express constraints for the instances of such a KR ontology.

The transformation process from WebODE to Protégé-2000 consists in creating instances of the 14 classes, and adding the corresponding values for their slots. Therefore, the Protégé-2000 ontology contains the 179 instances shown in the table.

The transformation back to WebODE is also rather simple, since it consists also in lexical and syntax transformations. As a consequence of such a transformation, the ontology obtained back is identical to the original WebODE ontology. Hence, we can state that we have preserved the semantics in the cyclic transformation.

Not only the semantics are preserved in the cyclic transformation, which can be easily proved since the original and the final ontologies are identical. The semantics are also preserved in the Protégé-2000 ontology obtained from the first translation step. The PAL constraints created as part of the WebODE KR ontology implemented in Protégé-2000 ensure that the semantics of the ontology are the same than those of the original WebODE one.

However, we cannot say the same about pragmatic preservation. Although in cyclic transformations the pragmatics is preserved (again, because we obtain an identical ontology), the ontology obtained in Protégé-2000 is not legible neither by human users nor by most of the ontology-based applications, since every ontology component is represented as an instance. Consequently, this ontology cannot be easily reused, adapted, etc.

Finally, the translation system from Protégé-2000 to WebODE cannot be used for all the ontologies available in Protégé-2000. It can be used only for those ontologies that have been generated previously in WebODE and transformed to Protégé-2000 with the previous translation system.

6.2.5 Experiment 5. Direct ontology translation with additional semantic preservation components

The final experiment consists also in 2 translation steps: in the first one, the ontology is transformed from WebODE to the standard knowledge model of Protégé-2000, instantiating only those knowledge components that are needed to preserve all the semantics of the original ontology; in the second step, that ontology is transformed back from Protégé-2000 to WebODE.
The results obtained are summarised in table 6.5. This experiment was also shown as a practical example in chapter 5.

The implementation effort for this translation approach is higher than in any of the previous approaches. First, we must create in Protégé-2000 the knowledge components that are not available in its standard knowledge model, and hide them so as not to affect the legibility of the Protégé-2000 ontologies transformed. The transformation consists in mapping the WebODE components to their corresponding Protégé-2000 standard knowledge components, and instantiating the additional ones only for expressing the knowledge that cannot be represented in this standard knowledge model.

The transformation from Protégé-2000 to WebODE can reuse the translation decisions of the third experiment. Besides, other transformations at the lexical and syntax level have to be done to translate the additional knowledge that could not be represented in the standard knowledge model of Protégé-2000.

Table 6.5. Summary of the results obtained in the direct translation between WebODE and Protégé-2000, with additional components for semantic preservation.

<table>
<thead>
<tr>
<th></th>
<th>WebODE</th>
<th>Protégé-2000 (step 1)</th>
<th>WebODE (step 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#concepts</td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>#subclass of</td>
<td>24</td>
<td>61</td>
<td>24</td>
</tr>
<tr>
<td>#disjoint decompositions</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>#exhaustive decompositions</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>#partitions</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>#attributes/relations</td>
<td>69</td>
<td>43</td>
<td>69</td>
</tr>
<tr>
<td>#axioms</td>
<td>8</td>
<td>45</td>
<td>8</td>
</tr>
<tr>
<td>#constants</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>#instances</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

The results shown in the table summarize the previous comments: Protégé-2000 users (either human users or systems) will obtain a Protégé-2000 ontology with the same 62 classes that were represented in WebODE, the same class taxonomy, 43 slots, some of which are attached to several classes, and 20 instances. Besides, this ontology has 45 PAL constraints, 8 of which come from the 8 WebODE formal axioms. The other PAL constraints express the disjoint and exhaustive knowledge in the class taxonomy. The rest of knowledge needed to preserve all the semantics in the transformation are hidden to Protégé-2000 users, such as the 6 predefined metaclasses and the 9 predefined classes used to preserve semantics in the transformation, as described in section 5.3.3.

With regard to the translation process from Protégé-2000 to WebODE, we obtain an ontology that is identical to the original one. Therefore, we can talk about semantic and pragmatic
preservation in cyclic transformations. This approach is the one used for the import and export services from and to Protégé-2000 that are currently available in WebODE.

With regard to the semantics and pragmatics of the ontology obtained in the intermediate step (that is, in Protégé-2000), they are also preserved completely, thanks to the direct mappings between components, and to the hiding of the additional knowledge transformed to ensure the preservation.
Chapter 7
Conclusions and Future Research Problems

This chapter presents the conclusions of our work, focusing on the main advances to the state of the art on ontology translation, namely: the proposal of a layered declarative ontology translation model and the analysis of different translation approaches with respect to their semantic and pragmatic preservation properties. In this chapter, we also identify open research issues in ontology translation between ontology languages and/or tools.

7.1 CONCLUSIONS

As described in chapter 3, the objective of the work presented in this thesis has been to advance the state of the art on ontology translation with the following two main subobjectives:

- Provision of an integrated model to build ontology translation systems, characterised by two main aspects: it takes into account different layers in the translation problems: lexical, syntax, semantic, and pragmatic, and the translation decisions can be expressed declaratively.

- Analysis of current translation approaches with respect to their semantic and pragmatic preservation properties in the translation processes, especially for the case of cyclic transformations.

Several open research problems have been taken into account in the proposal of solutions that solve both subobjectives:

- Lack of consensus about the classification of ontology translation problems. Many classifications exist, with different objectives (semantic interoperability, ontology merge and integration, etc.), but the translation levels inside each classification do not always coincide.
- Lack of integrated methods and techniques that support the complex task of building ontology translation systems. Many of these methods and techniques are applied in an isolated way for some aspects of the ontology translation building process, such as for comparing the source and target formats, for implementing the translation decisions, etc.

- The existing formal comparison approaches of KR languages rely on the use of translation functions between the source and target formats, but do not specify how these translation functions should be built. Therefore, they are useful only as a starting point for the development of ontology translation systems.

- The two existing systems that give technological support to the task of building ontology translation systems (OntoMorph and Transmorpher) impose important limits regarding the format of the source and target formats.

- Ontology building methodologies do not describe in detail how knowledge evolves when it is being translated between different formats.

- There are not detailed studies about the different approaches used so far for ontology translation, and about their impact on the preservation of the semantics and pragmatics of the transformed ontologies, especially in the case of cyclic transformations.

In summary, the following contributions to the state of the art have been provided as a result of the work presented in this thesis:

- **New characterisation of ontology translation problems**: lexical, syntax, semantic, and pragmatic. This classification aims at reaching consensus in the characterisation of ontology translation problems, since it is based on the work done in other existing theories, such as pragmatics, and provides a catalogue of problems that can appear in each layer, while the rest of works only provide examples.

- **Integrated model for building ontology translation systems**, which takes into account all the ontology translation layers previously identified and proposes to implement declaratively the ontology translation decisions for each layer.

- **A language (ODEDialect) composed of three declarative languages (ODELex, ODESyntax, and ODESem)**, which provide means to express and build the translation functions defined for the formal comparison approaches of KR languages and tool knowledge models.
- **Description of the knowledge lifecycle during translation processes**, paying special attention to cyclic transformations.

- **Analysis of the ontology translation approaches used for exchanging ontologies between ontology tools**, paying special attention to how each of these approaches preserves semantics and pragmatics during the translation process.

### 7.2 Open Research Problems

Even though this thesis presents important contributions to the current state of the art on ontology translation, there are still some open research problems that have not been taken into account for this thesis or that have appeared as a consequence of the advances proposed in this thesis. These open research problems, which mainly correspond to the thesis restrictions already identified in chapter 3, are described below:

- The ontology translation model is aimed at being used to build systems that transform domain ontologies between any pair of ontology languages and/or tools. The model is not specifically designed for other formats that allow representing other types of knowledge, such as knowledge about planning, knowledge about machine learning, etc., nor for other types of ontologies, such as knowledge representation ontologies or upper-level ontologies. Different ontology translation models could be necessary for such specific types of knowledge.

- The ontology translation model does not consider the optimisation of the translation process of the system built, neither in terms of space requirements nor in terms of time requirements. This negative aspects are due to the fact that a main requisite for the creation of this model was that it should be easy to create and maintain ontology translation systems, and they should be implemented declaratively. One of the negative impacts of such declarative approaches is that they usually have worse performance (in terms of time and space) than ad hoc solutions.

- The ontology translation model is only aimed at performing translations of full ontologies, and not of part of them. In other words, the model does not consider the possibility of performing incremental translations. These types of translations can be useful when reusing evolving ontologies in a language or tool that is different from the language or tool where they are available. In some of these cases, an ontology reengineering process must be performed, as described in [Gómez-Pérez and Rojas, 2000], and the ontology transformed is modified after the transformation. Incremental transformations can be used to translate
only the parts of the evolving ontology that has changed since the last translation, hence helping to maintain the restructured ontology.

- The three declarative languages that compose ODEDialect (ODELex, ODESyntax, and ODESem) are not expressive enough to represent every possible type of transformation that may appear in any transformation. They only cover the most common types of transformations identified at each layer. If a more complex transformation must be performed, user-defined functions must be used. Additional effort could be done to make these languages more expressive so as to cover more types of transformations.

- The analysis of semantic and pragmatic preservation during translation processes described in chapter 5 is restricted to the “representational aspects” of the source and target formats. However, it does not cover other aspects such as the reasoning tractability of the knowledge generated in the target format. This is an important aspect to be considered for many ontology-based applications, which also require that the ontologies transformed are tractable from a reasoning point of view.
Bibliography


on Artificial Intelligence (KI'01). Vienna, Austria. (Lecture Notes in Artificial Intelligence
LNAI 2174) Springer-Verlag, Berlin, Germany, pp 396–408

Bechhofer S, Goble C, Horrocks I (2001b) DAML+OIL is not enough. In: Decker S, Fensel D,
Sheth A, Staab S (eds) First Semantic Web Working Symposium (SWWS'01). Stanford,
California, pp 151–159

http://www.w3.org/TR/rdf-syntax-grammar

Berk E (1997) JLex: A lexical analyzer generator for Java, Technical Report, Department of 
Computer Science, Princeton University. Latest version available at: 

applications. In: Wahlster W (ed) European Conference on Artificial Intelligence 
(ECAI'96). Budapest, Hungary. John Wiley and Sons, Chichester, United Kingdom, pp 
298–302

Berners-Lee T (1999) Weaving the Web: The Original Design and Ultimate Destiny of the 

http://www.w3.org/TR/xmlschema-2/

Ontologies at the Knowledge Level using the Ontology Design Environment. In: Gaines BR,
Musen MA (eds) 11th International Workshop on Knowledge Acquisition, Modeling and 
Management (KAW'98). Banff, Canada, SHARE4:1–15

Artificial Intelligence 82(1-2):353–367

model for objects. In: Clifford J, Lindsay BG, Maier D (eds) ACM SIGMOD International 

Borst WN (1997) Construction of Engineering Ontologies. Centre for Telematica and 
Information Technology, University of Twente. Enschede, The Netherlands
Bibliography


   New York. IEEE Std. 1430-1996


Lenat DB, Guha RV (1990) Building Large Knowledge-based Systems: Representation and Inference in the Cyc Project. Addison-Wesley, Boston, Massachusetts


Woods WA (1983) *What’s important about knowledge representation*. IEEE Computer, 16(10)
I.1 ONTOLINGUA AND KIF

Ontolingua (Gruber, 1992; Farquhar et al., 1997) was released in 1992 by the Knowledge Systems Laboratory of Stanford University. It is an ontology language based on KIF (Genesereth and Fikes, 1992; NCITS, 1998) and on the Frame Ontology (Gruber, 1993a). Ontolingua is the ontology-building language used by the Ontolingua Server\(^1\) (Farquhar et al., 1997).

KIF (Knowledge Interchange Format) was developed in the context of the ARPA Knowledge Sharing Effort (Neches et al., 1991). The purpose for KIF's development was to solve the problem of language heterogeneity in knowledge representation, and to allow the interchange of knowledge between diverse information systems. KIF is a prefix notation of first order predicate calculus with some extensions. It permits the definition of objects, functions and relations with functional terms and equality. KIF has declarative semantics (it is possible to understand the meaning of expressions without an interpreter to manipulate them) and permits the

\(^1\) http://ontolingua.stanford.edu/
representation of meta-knowledge, reifying functions and relations, and non-monotonic reasoning rules.

As KIF is an interchange format, to implement ontologies with it is a very tedious task. But with the Frame Ontology (Gruber, 1993a), built on top of KIF, this task becomes easier. The Frame Ontology is a KR ontology for modelling ontologies under a frame-based approach and provides primitives such as Class, Binary-Relation, Named-Axiom, etc. Since it was built on the basis of KIF and a series of extensions of this language, this ontology can be completely translated into KIF with the Ontolingua Server's translators. In 1997, the Frame Ontology was modified because another representation ontology, the OKBC Ontology, was included between KIF and the Frame Ontology, as shown in Figure 4.1. As a result, several definitions from the Frame Ontology were moved to the OKBC Ontology.

Both the Frame Ontology and the OKBC Ontology are less expressive than KIF. This means that not all the knowledge that can be expressed in KIF can be expressed only with the primitives provided by these KR ontologies. Thus Ontolingua allows adding KIF expressions to the definitions implemented with the Frame Ontology and the OKBC Ontology. With the Ontolingua language we can build ontologies according to any of the four following approaches: (1) using the Frame Ontology vocabulary; (2) using the OKBC Ontology vocabulary; (3) using KIF expressions; and (4) combining the Frame Ontology vocabulary, the OKBC Ontology vocabulary and KIF expressions simultaneously.

Recently a theorem prover (JTP) has been developed for KIF expressions. Since all the expressions implemented in Ontolingua can be translated into KIF, JTP can be used to reason with Ontolingua ontologies.

Ontolingua ontologies are kept at the Ontolingua Server, which besides storing Ontolingua ontologies it allows users to build ontologies using its ontology editor, to import ontologies implemented in Ontolingua, and to translate ontologies from Ontolingua into KIF (Genesereth and Fikes, 1992), OKBC (Chaudhri et al., 1998), LOOM (MacGregor, 1991), Prolog, CORBAs IDL (Mowbray and Zahavi, 1995), Epikit (Genesereth, 1990) and CML (Stanley, 1986).

---

2 Java Theorem Prover (http://www.ksl.stanford.edu/software/JTP/)
I.1.1 Knowledge representation

In this section we present how ontologies are implemented in Ontolingua (either by combining the Frame Ontology vocabulary and KIF, or by using KIF exclusively). Now we present examples with the OKBC Ontology vocabulary since this protocol is described in Section 4.3.3. Several examples that have been extracted from our travel ontology are shown, and the main features of this language are pointed out.

The header of an Ontolingua ontology defines the ontology name, natural language (NL) documentation, level of maturity, and generality. The last two measures, maturity and generality, are assigned arbitrarily by the ontology developer, and their value ranges from very low to very high. Besides, the header refers to the ontologies imported. In our travel ontology, we import the following: the Frame-Ontology, kif-numbers, standard-units and time.

```
(define-ontology Travel-Ontology
  (Frame-Ontology kif-numbers standard-units time)
  "Sample ontology for travel agencies"
  :maturity :medium
  :generality :high)
```

The definition above is valid only for Ontolingua. KIF does not provide special means for defining the header of an ontology nor for importing terms from other ontologies. So if we want to import these terms, we must repeat their formal definitions in the new ontology.

Let us now describe in detail the components that can be implemented in Ontolingua. We start with the definition of concepts, which are called classes in Ontolingua. Classes are defined by their name, by an optional documentation and by one or several sentences defining constraints on the attributes of the class or on the class itself:

- The keywords :def, :sufficient and :iff-def establish necessary, sufficient, and necessary and sufficient conditions that any instance of the class must satisfy respectively.

- The keyword :equivalent establishes conditions for the equivalence of classes.

- The keywords :constraints and :default-constraints establish necessary conditions on attributes of the class and on the default values of the attributes of the class respectively. These constraints are inherited by its subclasses.

- The keywords :axiom-def, :axiom-constraints and :axiom-defaults allow defining second-order axioms about the class itself.
• The keywords :class-slots and :instance-slots can be used for defining class and instance attributes respectively. We can also define them inside the keyword :def and :iff-def, as presented below in the definition of the class Travel.

• The keyword :default-slot-values defines default values for attributes of the class.

• The keyword :issues is used for adding extra documentation to the class.

Below, we show the Ontolingua code that defines the concept Travel. In this definition, the documentation of the class has been included and some necessary constraints (:def) of the types of values of the instance attributes companyName and singleFare have been defined. Necessary and sufficient constraints (:iff-def) of the types of values of the instance attributes arrivalDate and departureDate have also been defined. Finally, with :axiom-def we have defined those expressions related to the class and not to its instances, such as the taxonomic relationships (Subclass-Of and Superclass-Of) and the exact cardinalities and maximum cardinalities of some of the attributes of the concept (with Template-Facet-Value). Note that the Superclass-Of primitive is not commonly used in Ontolingua class definitions, since it would be redundant to define that the class X is a superclass of the class Y, and that the class Y is a subclass of the class X. Hence, the primitive Subclass-Of is more frequently used.

(define-class Travel (?X)
   "A journey from place to place"
   :iff-def
   (and (arrivalDate ?X Date) (departureDate ?X Date))
   :def
   (and (companyName ?X String)
        (singleFare ?X CurrencyQuantity))
   :axiom-def
   (and (Superclass-Of Travel Flight)
        (Subclass-Of Travel Thing)
        (Template-Facet-Value Cardinality arrivalDate Travel 1)
        (Template-Facet-Value Cardinality departureDate Travel 1)
        (Template-Facet-Value Maximum-Cardinality singleFare Travel 1)))

Class attributes are defined with the keyword :class-slots. In the next example, transportMeans is defined as a class attribute of the class Flight, whose value is "plane". This attribute is not inherited to the subclasses nor to the instances of the class Flight. To be able to express that an attribute of a concept has some values that are inherited to its subclasses and instances, we would need default values, which cannot be represented in Ontolingua.
Besides, Flight is a subclass of Travel that has exactly one flightNumber. Note that in the previous definition the concept Travel was defined as a superclass of the concept Flight.

\[
\text{define-class} \quad \text{Flight} \quad (?X) \\
\text{"A journey by plane"} \\
\text{axiom-def} \quad \text{(and} \quad \text{(Subclass-Of} \quad \text{Flight} \quad \text{Travel)} \\
\quad \text{(Template-Facet-Value} \quad \text{Cardinality} \\
\quad \text{flightNumber} \quad \text{Flight} \quad 1)) \\
\text{class-slots} \quad \{(\text{transportMeans} \quad \text{"plane"})\}
\]

We have noted in the previous examples that we can restrict the type, cardinality, and maximum cardinality of attributes. We can also restrict their minimum cardinality and state their default values. Moreover, we can define arbitrary constraints on attributes, which are expressed by KIF sentences.

The concepts aforementioned (Travel and Flight) have been defined with primitives from the Frame Ontology. These concepts are transformed into unary relations in KIF. The definitions that follow next present the concepts Travel and Flight expressed in KIF, as provided by the Ontolingua Server^3. Note that some of the KIF definitions shown in this section have redundant information. For instance, classes are defined both as classes and as relations with arity 1.

\[
\text{defrelation} \quad \text{Travel} \quad (?X) \\
:= \quad \text{(and} \quad \text{(arrivalDate} \quad ?X \quad \text{Date)} \quad \text{(departureDate} \quad ?X \quad \text{Date))} \\
\quad \text{(Subclass-Of} \quad \text{Travel} \quad \text{Thing)} \\
\quad \text{(Superclass-Of} \quad \text{Travel} \quad \text{Flight)} \\
\quad \text{(Class} \quad \text{Travel)} \\
\quad \text{(Arity} \quad \text{Travel} \quad 1) \\
\quad \text{(Documentation} \quad \text{Travel} \quad \text{"A journey from place to place")} \\
\quad \Rightarrow \quad (\text{Travel} \quad ?X) \\
\quad \quad \text{(and} \quad \text{(companyName} \quad ?X \quad \text{String)} \\
\quad \quad \text{(singleFare} \quad ?X \quad \text{CurrencyQuantity})) \\
\quad \text{(Template-Facet-Value} \quad \text{Cardinality} \\
\quad \text{departureDate} \quad \text{Travel} \quad 1) \\
\quad \text{(Template-Facet-Value} \quad \text{Cardinality} \quad \text{arrivalDate} \quad \text{Travel} \quad 1) \\
\quad \text{(Template-Facet-Value} \quad \text{Maximum-Cardinality} \\
\quad \text{singleFare} \quad \text{Travel} \quad 1) \\
\]

\[
\text{defrelation} \quad \text{Flight} \\
\quad \text{(Class} \quad \text{Flight)} \\
\quad \text{(Arity} \quad \text{Flight} \quad 1) \\
\quad \text{(Documentation} \quad \text{Flight} \quad \text{"A journey by plane")} \\
\quad \text{(Subclass-Of} \quad \text{Flight} \quad \text{Travel)} \\
\quad \text{(Template-Facet-Value} \quad \text{Cardinality} \\
\quad \text{flightNumber} \quad \text{Flight} \quad 1) \\
\quad \text{(transportMeans} \quad \text{Flight} \quad \text{"plane")}
\]

\(^3\) All the KIF definitions presented in this section are presented here as they are automatically translated by the Ontolingua Server.
Both concepts are represented as instances of the class Class that are also unary relations: (Arity Flight 1) (Arity Travel 1). Their documentation is defined using the Documentation relation. In the first definition, the necessary and sufficient conditions for the concept Travel are expressed with KIF sentences after the symbol \( := \). Necessary conditions are expressed as logical implication \( \Rightarrow \). In the second one Flight is defined as a subclass of Travel, and it expresses the restriction on the cardinality of the attribute flightNumber. The value for the class attribute transportMeans is stated in the last sentence.

The Ontolingua definition of the concept Flight shows how to represent concept taxonomies using the Subclass-Of relation. We want to emphasize that in Ontolingua, a class must be always a subclass of, at least, another class, except for the generic class Thing. This class is the root class of the HPKB-upper-level ontology, which is a top level ontology converted from the Cyc upper-level ontology. If a class is not explicitly defined by the ontologist as a subclass of Thing, this knowledge is set by default by the Ontolingua Server.

Knowledge about disjoint and exhaustive concept decompositions and about partitions can also be represented in Ontolingua, with the primitives Disjoint-Decomposition, Exhaustive-Decomposition, and Partition respectively. These three primitives are defined in the Frame Ontology and can be found in the Ontolingua Server. Figure 1.1 illustrates their KIF definitions. The axioms included in both definitions deserve special attention as they define the meaning of these relations. The definition of Exhaustive-Decomposition imposes only a necessary condition (represented by \( \Rightarrow \) instead of \( := \)), that is, it does not define all the constraints of the relation, but only some of them.

If we reuse these definitions, we can build disjoint and exhaustive decompositions, and partitions. We show below two definitions that present the disjoint decomposition of the concept AmericanAirlinesFlight into AA7462, AA2010 and AA0488, and the partition of Location into the concepts EuropeanLocation, AsianLocation, AfricanLocation, AustralianLocation, AntarcticLocation, NorthAmericanLocation, and SouthAmericanLocation:

```kif
(define-class AmericanAirlinesFlight (?X)
 :def (Flight ?X)
 :axiom-def
  (Disjoint-Decomposition AmericanAirlinesFlight
   (Setof AA7462 AA2010 AA0488))
)

(define-class Location (?X)
 :axiom-def
  (Partition Location
   (Setof EuropeanLocation NorthAmericanLocation
    SouthAmericanLocation AntarcticLocation
    AsianLocation
    AfricanLocation AustralianLocation AntarcticLocation
    NorthAmericanLocation SouthAmericanLocation
    AustralianLocation SouthAmericanLocation
    AntarcticLocation)
  )
)```
Annex I. Detailed description of ontology languages

Figure I.1. KIF definition of Disjoint-Decomposition, Exhaustive-Decomposition, and Partition.

The previous definitions are translated completely into KIF as follows:

(defrelation AmericanAirlinesFlight)
(Class AmericanAirlinesFlight)
(Arity AmericanAirlinesFlight 1)
(Subclass-Of AmericanAirlinesFlight Flight)
(Disjoint-Decomposition AmericanAirlinesFlight
(setof AA7462 AA2010 AA0488))

(defrelation Location)
(Class Location)
(Arity Location 1)
As we can see in the KIF definition of the concept AmericanAirlinesFlight, this concept is defined as a subclass of the concept Flight. If we observe its definition in Ontolingua, we notice that we have used the expression :def (Flight ?X) instead of the expression :axiom-def (Subclass-Of AmericanAirlinesFlight Flight), used in the definition of the concept Travel. The KIF translation shows that both expressions mean the same, hence they can be used indistinctly. In the first one, we say that all the instances of the concept AmericanAirlinesFlight are necessarily instances of Flight, while in the second we say that the concept AmericanAirlinesFlight is a subclass of the concept Flight.

Binary relations are used to describe both the attributes of a concept and the relationships between two concepts. The following example, which combines the Frame Ontology vocabulary (define-relation and :def) and KIF expressions (the one attached to the :def primitive), shows the definition of the instance attribute singleFare and the binary relation arrivalPlace. The domains (concepts to which they are applied) and ranges (destination concepts or types of values) of both terms are explicitly described in these definitions.

The equivalent KIF definitions for these relations are presented below. In KIF, we explicitly declare twice that the relations are binary (by stating that they are binary relations and that their arity is 2). We also explicitly define the domain and range of both relations and their documentation.
Annex I. Detailed description of ontology languages

Ontolingua presupposes the unique name assumption for the names of all the terms of an ontology. For instance, in the case of relations this means that relation names must be unique in an ontology. We may want that two different concepts share the name of a relation, then the domain of the relation must be defined as being the union of both concepts (or (Ship ?X) (Flight ?X)), and the range of the relation must be restricted depending on the concept to which the relation is applied (specifying the value type with Template-Facet-Value value-type). Below there is an example where the relation departurePlace is different for the concepts Flight and Ship. The departure place of a flight must be an airport, while the departure place of a ship must be a port. In our travel ontology, both Airport and Port are defined as subclasses of Location.

```kif
(define-relation departurePlace (?X ?Y)
  :def (or (Ship ?X) (Flight ?X))
  :axiom-def
  ((Template-Facet-Value
    value-type departurePlace Ship Port)
   (Template-Facet-Value
    value-type departurePlace Flight Airport)))
```

This definition is presented below in KIF. It is explicitly stated that it is a binary relation, that its domain is Ship or Flight (expressed with :=>), and that its value type is Port or Airport, depending on whether its domain is Ship or Airport respectively (expressed with Template-Facet-Value).

```kif
(defrelation departurePlace (?X ?Y)
  :=> (or (Ship ?X) (Flight ?X))
  (Relation departurePlace)
  (Arity departurePlace 2)
  (Binary-Relation departurePlace)
  (Template-Facet-Value
    Value-Type departurePlace Ship Port)
  (Template-Facet-Value
    Value-Type departurePlace Flight Airport))
```

Higher arity relations are used to represent relationships between more than two concepts. Also in Chapter 1 we presented an Ontolingua definition for the ternary relation connects, which states that a road connects two cities provided they are different. This definition combines the Frame Ontology vocabulary and KIF expressions.

```kif
(define-relation connects (?city1 ?city2 ?road)
  "A road connects two different cities"
  :def
  (and
   (Location ?city1) (Location ?city2) (Road-Section ?road))
```
In KIF we have a relation whose arity is 3. The types of arguments are defined using the Nth-Domain primitive, which states the position of the relation argument and its type. Finally, all the logical sentences of the relation definition in Ontolingua are the same as those in KIF. Let us see the KIF definition of the relation connects:

```
(defrelation connects (?cityl ?city2 ?road) 
  (Documentation connects 
    "A road connects two different cities") 
  (Relation connects) 
  (Arity connects 3) 
  (Nth-Domain connects 1 Location) 
  (Nth-Domain connects 2 Location) 
  (Nth-Domain connects 3 RoadSection) 
  (not (part-of ?cityl ?city2)) 
  (not (part-of ?city2 ?cityl)) 
  (or (and (start ?road ?cityl) (end ?road ?city2)) 
    (and (start ?road ?city2) (end ?road ?cityl))))
```

Functions in Ontolingua are special kinds of relations, where the last argument (the output argument of the function) can be obtained from the set of input arguments. This relationship between input arguments and the output argument is expressed within :lambda-body. The next definition of the function Pays was presented. This function is used to obtain a room price after applying a discount.

```
(define-function Pays (?room ?discount) :-> ?finalPrice 
  "Price of the room after applying the discount" 
 :def (and (Room ?room) 
             (Number ?discount) 
             (Number ?finalPrice) 
             (Price ?room ?price)) 
 :lambda-body 
  (- ?price (/ (* ?price ?discount) 100)))
```

The equivalent KIF definition appears below. It is similar to the previous aforementioned of the relation connects, although it uses the primitive deffunction. In fact, in KIF functions are also a type of relations where the value of the output is always the same for a fixed set of the n-1 input arguments.

```
(deffunction Pays (?room ?discount ?finalPrice) 
  (Documentation Pays 
    "Price of the room after applying the discount") 
  (Function Pays) 
  (Arity Pays 3) 
  (Nth-Domain Pays 1 Room) 
  (Nth-Domain Pays 2 Number) 
  (Nth-Domain Pays 3 Number) 
  (= (Pays ?room ?discount) 
    (- ?price (/ (* ?price ?discount) 100)))
```
With regard to formal axioms, we can add that they can be defined in Ontolingua either as independent elements in the ontology or as embedded in other terms (inside concept definitions, relations, etc.). These axioms are represented directly in KIF but cannot be expressed with the Frame Ontology vocabulary. The following first order axiom declares that trains cannot go from the USA to Europe.

```
(define-Axiom NoTrainfromUSAtoEurope
  "It is not possible to travel from the USA to Europe by train"
  := (forall (?travel)
      (forall (?city1)
       (forall (?city2)
        (=> (and (Travel ?travel)
                 (arrivalPlace ?travel ?city1)
                 (departurePlace ?travel ?city2)
                 (EuropeanLocation ?city1)
                 (USALocation ?city2))
          (not (TrainTravel ?travel)))))))
```

Finally, an instance (which is also called an individual) is defined with the instance name, the class it belongs to and the values of the instance attributes. These instance attributes must be defined within the class to which the instance belongs or in any of the superclasses. The following example shows the definition of an instance of AA7462 flight in Ontolingua. Our instance has a single fare of 300 US Dollars, departs on February 8, 2002, and arrives in Seattle. 300USDollars, Feb8-2002 and Seattle are instances of CurrencyQuantity, Date and Location respectively.

```
(define-instance AA7462-Feb-08-2002 (AA7462)
  :def ((singleFare AA7462-Feb-08-2002 300USDollars)
        (departureDate AA7462-Feb-08-2002 Feb8-2002)
        (arrivalPlace AA7462-Feb-08-2002 Seattle)))
```

In KIF, instances are called objects and are defined in the same way as instances in Ontolingua. Here, the class to which the instance belongs and the values of the instance attributes must be defined explicitly.

```
(defobject AA7462-Feb-08-2002
  (AA7462 AA7462-Feb-08-2002)
  (singleFare AA7462-Feb-08-2002 300USDollars)
  (departureDate AA7462-Feb-08-2002 Feb8-2002)
  (arrivalPlace AA7462-Feb-08-2002 Seattle))
```

Other components, such as procedures, can be expressed in Ontolingua though their examples are not shown since they are not included in the common definition of heavyweight ontologies, as was explained in chapter 2.
I.1.2 Reasoning mechanisms

JTP\(^4\) is the theorem prover implemented for Ontolingua ontologies (actually, it is implemented for KIF and Ontolingua ontologies can be translated completely into KIF). JTP can be used for deducing information from axioms or for checking constraints in ontologies. It converts arbitrary first order logic sentences into clausal form, as presented in Russell and Norvig (1995), and it uses an iterative deepening search to find proofs extending the resolution techniques employed by Prolog, and avoiding getting lost in endless paths, as Prolog. In addition, JTP deals with inheritance.

The Ontolingua Server does not include JTP though it includes the following reasoning mechanisms: single and multiple inheritance and constraint checking. With regard to multiple inheritance, conflicts are not resolved in the Ontolingua Server. Furthermore, in the Ontolingua Server, inheritance is basically monotonic (that is, the Ontolingua Server does not deal with exceptions in concept taxonomies). The only non-monotonic capabilities of the Ontolingua Server are related to default values for slots and facets. The Ontolingua Server also checks the types and cardinalities of concept attributes and relation arguments.

I.2 RDF(S): RDF AND RDF SCHEMA

RDF (Lassila and Swick, 1999) stands for Resource Description Framework. It is being developed by the W3C to create metadata for describing Web resources, and it has been already proposed as a W3C recommendation. The RDF data model is equivalent to the semantic networks formalism and consists of three object types: resources, properties and statements.

The RDF data model does not have mechanisms for defining the relationships between properties and resources. This is the role of the RDF Vocabulary Description language (Brickley and Guha, 2004), also known as RDF Schema or RDFS, which is also a W3C Recommendation.

RDF(S) is the term commonly used to refer to the combination of RDF and RDFS. Thus, RDF(S) combines semantic networks with frames but it does not provide all the primitives that are usually found in frame-based knowledge representation systems. In fact, neither RDF, nor RDFS, and nor their combination in RDF(S) should be considered as ontology languages per se, but rather as general languages for describing metadata in the Web. From now on, we will always refer to RDF(S) in our examples and explanations.

\(^4\) http://www.ksl.stanford.edu/software/JTP/
RDF(S) is widely used as a representation format in many tools and projects, and there exists a huge amount of resources for RDF(S) handling, such as browsing, editing, validating, querying, storing, etc. In the section about further readings, we provide several URLs where updated information about RDF(S) resources can be found.

I.2.1 Knowledge representation

RDF(S) provides the most basic primitives for ontology modelling, achieving a balance between expressiveness and reasoning. It has been developed as a stable core of primitives that can be easily extended. In fact, as we will discuss later, languages such as OIL, DAML+OIL, and OWL reuse and extend RDF(S) primitives. In the following pages, we will review which types of ontology components may be represented in RDF(S).

In our examples we will use the XML syntax of RDF(S) (Beckett, 2004), and will also present the corresponding graphs for some of them. There is a syntax for RDF(S) based on triples (also called N-Triples notation) but it will not be presented here.

Before analysing which ontology components can be expressed in RDF(S), let us see how to define a RDF(S) ontology. An ontology in RDF(S) must start with the root node RDF. In this root node the namespaces for the RDF and RDFS KR ontologies must be included. These namespaces are usually identified with the prefixes rdf and rdfs, and they point to the standard URLs of both RDF and RDFS respectively. This definition is shown below:

```xml
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
```

The use of these namespaces allow us to use the prefixes rdf and rdfs for those KR primitives that belong to RDF and RDFS respectively.

The file containing our ontology will be located in a specific URL (for instance, http://www.ontologies.org/travel). However, the definitions inside it could refer to a different URL (for instance, http://delicias.dia.fi.upm.es/RDFS/travel). In that situation, we must add an xml:base definition to the root node, pointing to the second URL, as follows:

```xml
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xml:base="http://delicias.dia.fi.upm.es/RDFS/travel">```

Besides, if we want to import terms from other ontologies into our ontology, we should refer to them in any place of our ontology using the URL of those terms. For instance, if we want to

---

5 http://www.w3.org/TR/rdf-testcases/#ntriples
import the concept CurrencyQuantity from a RDFS ontology of units that is placed at http://delicias.dia.fi.upm.es/RDFS/units, we should use the following identifier for the term when referring to it:

http://delicias.dia.fi.upm.es/RDFS/units#CurrencyQuantity

As it is very tedious to write the previous expression every time we refer to this concept, we can create a XML entity units in the document type declaration that precedes the ontology definitions, as presented below, and use &units;CurrencyQuantity to refer to the concept CurrencyQuantity from that ontology. In our examples, we will suppose that the following entities have been created:

```xml
<ENTITY units 'http://delicias.dia.fi.upm.es/RDFS/units#'>
<ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
<ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
<ENTITY xsd 'http://www.w3.org/2001/XMLSchema#'>
```

The entity xsd is normally used to refer to XML Schema datatypes (Biron and Malhotra, 2001), which are used as the range of many ontology properties. In the definitions we present in this section, the ontology terms will be referred to with three attributes: rdf:ID, rdf:about and rdf:resource. There are slight differences in the use of these attributes. The first (rdf:ID) is used only once for each ontology term to create an anchor-id for the term in the document where the ontology is defined. The second (rdf:about) is used to extend the definition of the resource to which it refers. The third (rdf:resource) is used to refer to an ontology term, without adding more knowledge. In the second and third cases, if the ontology term is defined in the same document as the reference is, we must precede its name by #.

Other attributes can be used to refer to terms in a RDF(S) file (rdf:nodeID and rdf:bagID) but they are out of the scope of this description. We refer to Beckett (2004) for a description of these attributes.

**Concepts** are known as classes in RDF(S). Though there exist several patterns to specify a class in RDF(S) with XML syntax, here we will use the most compact syntax. Classes are referenced either by their name or by a URL to a Web resource. They can also include their documentation and their superclasses. This is shown in the following code for the concept Travel.

```xml
<rdfs:Class rdf:ID="Travel">
  <rdfs:comment>A journey from place to place</rdfs:comment>
</rdfs:Class>
```

Figure 1.2 presents the graphical notation for the concept Travel and its attributes. This figure contains classes and properties that belong to the RDF KR ontology (rdf:Property and rdf:type),
to the RDFS KR ontology (rdfs:Literal, rdfs:Class, rdfs:domain and rdfs:range), to a domain ontology about units (units:currencyQuantity), to XML Schema datatypes (xsd:date), and to our travel ontology (Travel, companyName, singleFare, departureDate and arrivalDate). The primitive rdf:type determines if a resource (represented by an ellipse) is a class or a property. For the properties of our ontology we should define their domain and range.

![Diagram of the class Travel in RDF(S)](image)

Instance attributes of classes are defined as properties in RDFS. The domain of these properties is the class to which the attribute belongs, and the range for those properties is the type of the attribute values. No cardinality constraints nor default values can be defined for attributes.

In the definitions for properties of the example that follows, we use XML entities for referring to XML Schema datatypes (&xsd;date and &xsd;integer) and terms in other ontologies (&units;CurrencyQuantity and &rdfs;Literal).

```xml
<rdf:Property rdf:ID="arrivalDate">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&xsd;date"/>
</rdf:Property>
<rdf:Property rdf:ID="departureDate">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&xsd;date"/>
</rdf:Property>
<rdf:Property rdf:ID="companyName">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
```
Class attributes can be represented in RDF(S) similarly, although by defining the domain of the property as rdfs:Class, and including the property value in the class definition. For example, the following definitions represent the class attribute transportMeans and the concept Flight:

```xml
<rdfs:Property rdf:ID="transportMeans">
    <rdfs:domain rdf:resource="&rdfs;Class"/>
    <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdfs:Property>

<rdfs:Class rdf:ID="Flight">
    <rdfs:comment>A journey by plane</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#Travel"/>
    <transportMeans rdf:datatype="&rdfs;Literal">plane</transportMeans>
</rdfs:Class>
```

Concept taxonomies are built in RDF(S) by defining a class as a subclass of one or more classes. However, neither disjoint nor exhaustive knowledge in concept taxonomies can be expressed in this language.

Binary relations between classes are defined in RDF(S) as properties (in fact, all of the previous examples could be considered binary relations, as they relate two concepts). Therefore, as in the case of attributes, we may constrain their domain and range.

However, relations of higher arity cannot be represented directly in RDF(S). They must be defined with the same technique used for other languages, that is, by creating a class that represents the relation and whose attributes correspond to each of the arguments of this relation.

```xml
<rdfs:Class rdf:ID="connects">
    <rdfs:comment>A road connects two different cities</rdfs:comment>
</rdfs:Class>

<rdfs:Property rdf:ID="firstArgument">
    <rdfs:domain rdf:resource="#connects"/>
    <rdfs:range rdf:resource="#Location"/>
</rdfs:Property>

<rdfs:Property rdf:ID="secondArgument">
    <rdfs:domain rdf:resource="#connects"/>
    <rdfs:range rdf:resource="#Location"/>
</rdfs:Property>

<rdfs:Property rdf:ID="thirdArgument">
    <rdfs:domain rdf:resource="#connects"/>
    <rdfs:range rdf:resource="#RoadSection"/>
</rdfs:Property>
```
It is also important to mention here that in Section 7.3 of the RDF specification document (Lassila and Swick, 1999), the authors describe how to model non-binary relations between instances in RDF. Their description only shows how to represent that a non-binary relation holds among several instances (or values). However, if we follow those guidelines, we will not be able to represent that non-binary relations can hold between classes, as we did in the above definition. Our definition could be instantiated as follows: ("the road 101 connects San Francisco and Los Angeles):  

```
<connects rdf:ID="connects-SF-LA">
  <firstArgument rdf:resource="#SanFrancisco/>
  <secondArgument rdf:resource="#LosAngeles/>
  <thirdArgument rdf:resource="#101/>
</connects>
```

An important feature of RDF(S) is that it is possible to create hierarchies of relations using the `subPropertyOf` primitive. For example, we can express that the relation `travelsByPlane` (whose range is `Flight`) specializes the relation `travels` (whose range is `Travel`). This means that if the relation `travelsByPlane` holds between two instances, then the relation `travels` holds between the same two instances.

```
<rdfs:subPropertyOf rdf:resource="#travels"/>
<rdfs:range rdf:resource="#Flight"/>
```

**Functions** are not components of the RDF(S) knowledge model. And nor are formal axioms, though Maedche and colleagues (2000) have carried out some studies in order to include axioms in RDF(S) ontologies. These studies propose to embed sentences in FLogic into RDF(S) axioms, so that a FLogic inference engine can read the information of RDF(S) documents and reason with the knowledge described in the ontology. This approach could be followed with other languages as well.

Finally, and regarding instances, there are several patterns to define instances in an ontology. To do so we only need primitives from RDF (not RDFS). The definition of an instance includes the class to which the instance belongs and the corresponding set of property-value pairs. Depending on the range of each property, we have different representations of their value: a literal for `singleFare`, a typed literal for `departureDate` (its type is expressed with the `rdf:datatype` attribute), and a reference to a resource for `arrivalPlace`. The corresponding graphical notation of this instance is shown in Figure 1.3. The value representations of each property are also different depending on their range: a rectangle with the value 300 US Dollars for `singleFare`, a rectangle with the value 2002-02-08^xsd:date for
departureDate, which expresses that its type is xsd:date and its value 2002-02-08, and an ellipse with the instance Seattle for arrivalPlace.

![Diagram](image)

Figure 1.3. Definition of the instance AA7462Feb082002 in RDF(S).

 Assertions made by instances (aka claims) can be represented in RDF(S) using reification. Reification in RDF consists in transforming the value of a property into a statement. In the following example, our instance travelAgency1 claims that the flightNumber of the instance AA7462Feb082002 is 7462. This is represented in RDF(S) as a statement whose subject, predicate, and object are AA7462Feb082002, flightNumber and 7462 respectively, and whose property claim:attributedTo has the value travelAgency1. Observe that the property claim:attributedTo neither belongs to the RDF nor to the RDFS KR ontologies; it has been created as an example.

This definition is also shown in Figure 1.4, where the statement representing the claim is represented as an empty ellipse since we have not provided any identifier for it.

---

For a deeper insight of this mechanism, refer to Section 4 of the RDF specification document (Lassila and Swick, 1999).
Finally, some comments have to be made on RDF(S) semantics. At present, some studies are being carried out by the W3C on the definition of the RDF(S) model theory (Hayes, 2004). Previous to this work, the logical interpretation of this language (based on semantic networks) was explored by Conen and Klapsing (2000; 2001). The lack of formal semantics to create RDF(S) has caused several problems to define the semantics of other languages that extend RDF(S) like OIL, DAML+OIL, and OWL, which will be dealt with in the next sections.

1.2.2 Reasoning mechanisms

Currently, most of the inference systems for RDF(S) are mainly devoted to querying information about RDF ontologies as if they were a deductive database. In this context, some languages for querying RDF databases have appeared such as RQL (Karvounarakis and Christophides, 2002), RDFQL\(^7\), and RDQL\(^8\). Standardization efforts will be needed to create a common query language for RDF(S).

In the following list we enumerate several RDF(S) inference systems (although it is by no means an exhaustive list): SilRI (Simple Logic RDF Inference) (Decker et al., 1998), included in Ontobroker; RIL\(^9\) (RDF Inference Language), and TRIPLE (Sintek and Decker, 2001). Each of them has their own features though they are not dealt with here. There are also many other

---

\(^7\) http://www.intellidimension.com/RDFGateway/Docs/rdfqlmanual.asp
\(^8\) http://www.hpl.hp.com/semweb/rdql.html
RDF(S) tools and APIs available for downloading, such as Amaya\textsuperscript{10}, the Jena toolkit\textsuperscript{11}, the ICS-FORTH RDFSuite\textsuperscript{12}, etc. These tools can be used to store RDF(S) ontologies, parse RDF(S), query RDF(S) ontologies, etc. Some of them are available in Java, others in C++, Lisp, Python, etc.

1.3 DAML+OIL

DAML+OIL (Horrocks and van Harmelen, 2001) was developed by a joint committee from the USA and the European Union (mainly OIL developers) in the context of the DARPA project DAML (\textit{DARPA Agent Markup Language}). The main purpose of this language is to allow semantic markup of Web resources.

DAML+OIL has passed through several stages in its development. The first version of the language, called DAML-ONT, was created just as an extension of RDF(S) and provided frame-based KR primitives. It was released in October 2000. The following version, released in December 2000, was called DAML+OIL. This second version moved away from frames to DL, as shown by Bechhofer and colleagues (2001b). The last version of DAML+OIL was released in March 2001. Basically, this last version fixed some problems that were detected in the prior specification and changed some of the primitives of that version. None of these DAML+OIL versions used a layered structure for the language as OIL did.

The DAML+OIL KR ontology contains primitives that are equivalent to those of the RDF KR ontology (\texttt{rdf:Property}, \texttt{rdf:type}, and \texttt{rdf:value}), primitives that are equivalent to those of the RDFS KR ontology (\texttt{rdfs:Literal}, \texttt{rdfs:subPropertyOf}, \texttt{rdfs:subClassOf}, \texttt{rdfs:domain}, \texttt{rdfs:range}, \texttt{rdfs:label}, \texttt{rdfs:comment}, \texttt{rdfs:seeAlso} and \texttt{rdfs:isDefinedBy}), and primitives that are new (the rest of them). In this section, we will use the prefix that indicates the language to which it belongs (\texttt{rdf} for RDF, \texttt{rdfs} for RDF Schema and \texttt{daml} for DAML+OIL) for all the primitives.

DAML+OIL ontologies are written in XML (no plain text syntax, as in the case of OIL). And they can also be written with the triple notation for RDF. We will use the XML syntax for all the examples in this section so that readers can get a flavor of this syntax.

\textsuperscript{9} http://rdfinference.org/index.html
\textsuperscript{10} http://www.w3.org/Amaya/Amaya.html
\textsuperscript{11} http://www.hpl.hp.com/semweb/jena-top.html
\textsuperscript{12} http://www.ics.forth.gr/proj/iss/RDF/
There are many tools, systems and applications to manage and use DAML+OIL ontologies. Many of them are being adapted to the OWL language, since this language will supersede DAML+OIL.

I.3.1 Knowledge representation

DAML+OIL ontologies are based on RDF(S). Therefore, an ontology in DAML+OIL must start with the declaration of the RDF root node. In this node we will include the namespaces for the RDF, RDFS and DAML+OIL KR ontologies. Besides, the namespace $xsd$ is usually included for XML Schema datatypes. Please note that $xsd$ refers to a different URL than the one used for the same namespace in RDF(S). When DAML+OIL was created RDF(S) did not support XML Schema datatypes. In March 2001, the valid URL for XML Schema datatypes was http://www.w3.org/2000/10/XMLSchema. Later, RDF(S) gave support to XML Schema datatypes and that namespace had already changed to http://www.w3.org/2001/XMLSchema. However, DAML+OIL has not evolved and existing DAML+OIL ontologies use the old reference for XML Schema datatypes.

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#"
 xmlns:daml="http://www.daml.org/2001/03/daml+oil#">
```

As in RDF(S), the order in which definitions appear in DAML+OIL ontologies is not relevant, but DAML+OIL ontologies usually define the header of the ontology first and then the ontology terms.

The header of a DAML+OIL ontology includes: the ontology version (optional), the ontology documentation, which uses the primitive rdfs:comment, and the imported ontologies. In our case, we will import ontologies about units and datatypes (XML Schema definitions). We will also import the DAML+OIL KR ontology.

```
<daml:Ontology rdf:about="">
 <daml:versionInfo>1.0</daml:versionInfo>
 <rdfs:comment>Sample ontology for travel agencies</rdfs:comment>
 <daml:imports
  rdf:resource="http://delicias.dia.fi.upm.es/DAML/units"/>
 <daml:imports
  rdf:resource="http://www.w3.org/2000/10/XMLSchema"/>
 <daml:imports
  rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
</daml:Ontology>
```

As in RDF(S), we will use XML entities to refer to terms from other ontologies rather than use directly the URL of those terms. In the RDF(S) section we indicated how to define entities in XML documents.
Concepts are known as classes in DAML+OIL, and are created with the primitive `daml:Class`. Besides its name, a class might contain its documentation (with `rdfs:comment`) and expressions with the following list of primitives:

- `rdfs:subClassOf` contains class expressions. It allows defining the superclasses of the class.
- `daml:sameClassAs` and `daml:equivalentTo` also contain class expressions. These primitives define necessary and sufficient conditions for the class (i.e., they are used for redefining concepts already defined), and can be used indistinctly.
- `daml:oneOf`, defines a class by enumerating exhaustively all its instances. This is a way to define a class extensionally.
- `daml:intersectionOf`, `daml:unionOf` and `daml:complementOf` define an expression as a conjunction, a disjunction, or a negation of other expressions respectively.

According to the DL terminology, DAML+OIL is a SHIQ language. Sometimes it appears as a SHOIQ language, where the $O$ expresses the possibility of defining “nominals” (individuals that are used in `daml:oneOf` expressions). This means that class expressions can be built using the following constructors:

- Conjunction (`daml:intersectionOf`), disjunction (`daml:unionOf`) and negation (`daml:complementOf`).
- Collections of individuals (`daml:oneOf`).
- Property restrictions (`daml:Restriction`). They contain a reference to the property to which the restriction is applied with the primitive `daml:onProperty` and another element for expressing the restriction. The following restrictions can be applied to properties: value restriction (`daml:toClass`), existential restriction (`daml:hasClass`), role fillers (`daml:hasValue`), number restriction (`daml:cardinality`, `daml:maxCardinality`, `daml:minCardinality`) and qualified number restriction (`daml:cardinalityQ`, `daml:maxCardinalityQ`, `daml:minCardinalityQ`).
- Additionally, role expressions can express inverse roles (`daml:inverseOf`), and role hierarchies can be defined (`daml:subPropertyOf`).

The grammar to build class expressions is shown next:

```
class-name | <daml:intersectionOf> class-exp class-exp+ </daml:intersectionOf> | <daml:unionOf> class-exp class-exp+ </daml:unionOf> | <daml:complementOf> class-exp </daml:complementOf> | <daml:oneOf rdf:parseType="daml:collection">```


instance-expr represents an instance of a class. It is defined as follows:

```
<daml:Thing rdf:resource="#{instanceName}"/>
```

restriction-expr represents a restriction on a property when applied to the class. It can be any of the following:

```
<daml:toClass> class-expr </daml:toClass> |
<daml:hasClass> class-expr </daml:hasClass> |
<daml:hasValue> instance-name </daml:hasValue> |
<daml:cardinality> non-neg-integer </daml:cardinality> |
<daml:maxCardinality> non-neg-integer </daml:maxCardinality> |
<daml:minCardinality> non-neg-integer </daml:minCardinality> |
<daml:cardinalityQ> non-neg-integer </daml:cardinalityQ> |
<daml:hasClassQ> class-expr </daml:hasClassQ> |
<daml:maxCardinalityQ> non-neg-integer </daml:maxCardinalityQ>
```

The definition of the primitive concept Travel is shown below. The class expression inside `rdfs:subClassOf` establishes restrictions on the instance attributes of this concept: a travel has exactly an arrival date and a departure date (whose type is date, which is a predefined XML Schema datatype), at most, a single fare (whose type is currencyQuantity, which is defined in the units ontology) and this travel may be made with zero, one or several companies.

```
<daml:Class rdf:ID="Travel">
  <rdfs:comment>A journey from place to place</rdfs:comment>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#arrivalDate"/>
      <daml:toClass rdf:resource="&xsd;date"/>
      <daml:cardinality>1</daml:cardinality>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#departureDate"/>
      <daml:toClass rdf:resource="&xsd;date"/>
      <daml:cardinality>1</daml:cardinality>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#companyName"/>
      <daml:toClass rdf:resource="&xsd;string"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#singleFare"/>
      <daml:toClass rdf:resource="&units;currencyQuantity"/>
      <daml:maxCardinality>1</daml:maxCardinality>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```
The definition of the defined concept Flight is shown below. A flight is a type of travel. It has an instance attribute called flightNumber, which must have exactly one value of type integer. Besides, it has also an attribute called transportMeans, whose value is "plane". This value is inherited by all the subclasses and instances of Flight.

```
<daml:Class rdf:ID="Flight">
  <rdfs:comment>A journey by plane</rdfs:comment>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Travel"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#flightNumber"/>
      <daml:toClass rdf:resource="&xsd;integer"/>
      <daml:cardinality>1</daml:cardinality>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#transportMeans"/>
      <daml:hasValue>
        <xsd:string rdf:value="plane"/>
      </daml:hasValue>
      <daml:cardinality>1</daml:cardinality>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>
```

Class attributes can be represented as we explained in RDF(S). Although this type of definition is valid in RDF(S), DAML+OIL inference engines will not take it into account, since it is not included in the DAML+OIL specification.

Besides, the primitives `daml:equivalentTo` and `daml:sameClassAs` may be used indistinctly for defining equivalences between classes. These expressions are normally used for defining synonyms. For instance, TravelByAir could be a synonym of Flight in our ontology, as can be seen below:

```
<daml:Class rdf:ID="TravelByAir">
  <daml:sameClassAs rdf:resource="#Flight"/>
</daml:Class>
```

In the previous definitions, we have explored how to represent restrictions on instance attributes in DAML+OIL by expliciting their type and cardinality. We have also explored how to represent restrictions on class attributes expliciting their values. Concept attributes must be defined as properties in the ontology. In DAML+OIL, there are two types of properties: `daml:ObjectProperty`, whose range is a class, and `daml:DatatypeProperty`, whose range is a datatype.

To define a property we may explicit its domain and range with the primitives `rdfs:domain` and `rdfs:range`. We can also state that a property is a subproperty of other properties with the primitive `rdfs:subPropertyOf`. Finally, we can also express equivalence between properties with `daml:samePropertyAs` or `daml:equivalentTo` indistinctly, and inverse properties with...
Let us now see the attribute definitions arrivalDate, departureDate, companyName and singleFare of the concept Travel, and the attribute definitions flightNumber and transportMeans of the concept Flight.

```xml
<daml:DatatypeProperty rdf:ID="arrivalDate">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&xsd;date"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="departureDate">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&xsd;date"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="companyName">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</daml:DatatypeProperty>
<daml:ObjectProperty rdf:ID="singleFare">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&units;currencyQuantity"/>
</daml:ObjectProperty>
<daml:DatatypeProperty rdf:ID="flightNumber">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&xsd;integer"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="transportMeans">
  <rdfs:domain rdf:resource="#Travel"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</daml:DatatypeProperty>
```

Concept taxonomies in DAML+OIL are created with the rdfs:subClassOf primitive in the case of primitive concepts, and with the daml:intersectionOf, daml:unionOf, daml:sameClassAs or daml:equivalentTo primitives in the case of defined concepts. We have already seen some examples when defining the concepts Travel and Flight.

Disjoint and exhaustive knowledge can be expressed in DAML+OIL. Disjointness between concepts is expressed with the daml:disjointWith primitive. We present below the definitions of the concepts AA7462, AA2010 and AA0488 stating that they are disjoint with each other. Note that due to the symmetry of the daml:disjointWith primitive, it would not be necessary to write all the daml:disjointWith statements, though it would be enough to write only three (for example, AA7462 daml:disjointWith AA2010, AA7462 daml:disjointWith AA0488, and AA2010 daml:disjointWith AA0488).

```xml
<daml:Class rdf:ID="AA7462">
  <rdfs:subClassOf rdf:resource="#AmericanAirlinesFlight"/>
  <daml:disjointWith rdf:resource="#AA2010"/>
  <daml:disjointWith rdf:resource="#AA0488"/>
</daml:Class>

<daml:Class rdf:ID="AA2010">
  <rdfs:subClassOf rdf:resource="#AmericanAirlinesFlight"/>
  <daml:disjointWith rdf:resource="#AA7462"/>
</daml:Class>
```
Note that the disjoint decomposition has been represented by explicitly including a \texttt{rdfs:subClassOf} statement in each of the concept definitions, which link each concept with the concept \texttt{AmericanAirlinesFlight}.

Partitions are expressed with the \texttt{daml:disjointUnionOf} primitive. Let us see now the definition of the partition of the concept \texttt{Location} in \texttt{EuropeanLocation}, \texttt{AsianLocation}, \texttt{NorthAmericanLocation}, \texttt{SouthAmericanLocation}, \texttt{AfricanLocation}, \texttt{AustralianLocation}, and \texttt{AntarcticLocation}.

There are no specific primitives in DAML+OIL to represent exhaustive decompositions of a concept, although they can be represented with the \texttt{daml:unionOf} primitive, similarly to the use of \texttt{daml:disjointUnionOf}.

We have already presented some binary \texttt{relations} for building taxonomies in DAML+OIL. In this language, binary relations are defined with the primitive \texttt{daml:ObjectProperty}. It is not compulsory to define explicitly the domain and range of a property in DAML+OIL. Thus, if two different concepts have properties with the same name, the property domain will contain two classes. If the attribute types are different for each class, it is recommended not to define the range of the property in the property definition but to add the restriction about its type to the definition of the corresponding classes. For instance, in the case of the relation \texttt{departurePlace} applied to the concept \texttt{Ship} and the concept \texttt{Flight}, we have the following definitions:
The definition of a relation can include some additional mathematical properties. We declare that a relation is transitive (\texttt{daml:TransitiveProperty}), that it has just one value when applied to an instance (\texttt{daml:UniqueProperty}), or that it is unambiguous (\texttt{daml:UnambiguousProperty}). The last two cases define global cardinality restrictions on the property no matter which class they are applied to.

Higher arity relations must be defined as concepts in DAML+OIL, as they are defined in other languages. The corresponding definition for the ternary relation \texttt{connects} is as follows:

```xml
<daml:Class rdf:ID="connects">
  <rdfs:comment>A road connects two different cities</rdfs:comment>
</daml:Class>
<daml:ObjectProperty rdf:ID="firstArgument">
  <rdfs:domain rdf:resource="#connects"/>
  <rdfs:range rdf:resource="#Location"/>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="secondArgument">
  <rdfs:domain rdf:resource="#connects"/>
  <rdfs:range rdf:resource="#Location"/>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="thirdArgument">
  <rdfs:domain rdf:resource="#connects"/>
  <rdfs:range rdf:resource="#RoadSection"/>
</daml:ObjectProperty>
```

As in other markup languages, in DAML+OIL we cannot express the constraint that cities are not part of each other and that the road starts in one of the cities and ends in the other one.
**Functions** are not components of the DAML+OIL knowledge model. However, binary functions may be represented with the `daml:UniqueProperty` primitive previously discussed. Higher arity functions cannot be represented in this language.

Neither are formal **axioms** components of the DAML+OIL knowledge model either. In this case, the same studies carried out on RDF(S) by Maedche and colleagues (2000) could be applied to DAML+OIL since DAML+OIL is built on top of RDF(S).

With regard to **instances**, they are defined using only RDF vocabulary. Thus the definition of our instance of flight AA7462 is similar to the one presented in the previous section. We have changed the range of the property `singleFare`, which refers to an instance of the concept `CurrencyQuantity` in the ontology of units, while in the previous section it refers to a literal value (300 US Dollars). Since the attribute `rdf:datatype` was included in RDF(S) after DAML+OIL was created, the value of the attribute `departureDate` is also expressed differently from that one, as proposed in XML Schema. Claims could also be represented in the same fashion as they were in RDF(S).

```xml
<AA7462 rdf:ID="AA7462Feb082002" >
  <singleFare rdf:resource="&units;USDollar300"/>
  <departureDate>
    <xsd:date rdf:value="2002-02-08"/>
  </departureDate>
  <arrivalPlace rdf:resource="#Seattle"/>
</AA7462>
```

### 1.3.2 Reasoning mechanisms

Model-theoretic semantics (van Harmelen et al., 2001) and a KIF axiomatization (Fikes and McGuinness, 2001) are provided for DAML+OIL. Both approaches give meanings to those primitives that are specifically defined in the DAML+OIL KR ontology. Any additional statements, resulting in other RDF triples, are perfectly allowed in this language. However, DAML+OIL is silent on the semantic consequences (or lack thereof) of such additional triples.

Both specifications of the semantics of DAML+OIL permit building inference engines or using existing ones. In fact, there are several inference engines available for DAML+OIL. For instance, FaCT and RACER (described when dealing with OIL description) can be used to reason with DAML+OIL ontologies since they allow reasoning with SHIQ languages.

TRIPLE (Sintek and Decker, 2001) is an inference engine that allows defining the semantics of any RDF-based language by means of rules. Hence it can be used with OIL, DAML+OIL and
OWL ontologies\textsuperscript{13}. In the case of DAML+OIL ontologies, TRIPLE may also use external classifiers such as FaCT and RACER. This inference engine is available in Java and can be downloaded free\textsuperscript{14}.

The JTP theorem prover, referenced in the section of Ontolingua and KIF, provides support for reasoning with DAML+OIL ontologies based on the KIF axiomatization of DAML+OIL primitives.

DAML+OIL ontologies can be used in Jess\textsuperscript{15}. For example, the DAML+OIL API provided by GRCl\textsuperscript{16} allows loading DAML+OIL statements from Jess. This permits writing Jess programs that can reason with domain ontologies written in DAML+OIL.

The DAML Query Language\textsuperscript{17} (DQL) also uses DAML+OIL ontologies. DQL is a formal language and protocol for a querying agent and an answering agent to use in conducting a query-answering dialogue.

Finally, all the inference engines available for RDF(S) can be used for DAML+OIL. However, these inference engines do not exploit the semantics of DAML+OIL primitives but take them as if they were other RDF triples without any specific semantics.

As explained in the OIL section, the use of DL classifiers (such as FaCT or RACER) permits performing automatic classifications of the ontology concepts, and detecting inconsistencies in this concept taxonomy.

Independently of the inference engine that we use for reasoning with our DAML+OIL ontologies, multiple inheritance is allowed. However, conflicts in multiple inheritance are not yet solved. Constraint checking can be performed on the values of properties and their cardinalities.

Finally, many tools can be used currently for authoring DAML+OIL ontologies\textsuperscript{18} such as OILEd, OntoEdit, WebODE and DUET, among others. Any other tool capable of working with RDF(S) ontologies can also be used for developing DAML+OIL ontologies provided that the ontology developer uses the DAML+OIL KR primitives. RDF(S) query engines, storage systems and parsers can be used to manage DAML+OIL ontologies since they can be serialized in RDF(S).

\textsuperscript{13} As TRIPLE does not provide any implementation of the semantics of OIL, it was not presented in the previous section, though it could be used for reasoning with OIL ontologies.

\textsuperscript{14} http://triple.semanticweb.org/\textsuperscript{14}

\textsuperscript{15} http://herzberg.ca.sandia.gov/jess/\textsuperscript{15}

\textsuperscript{16} http://codip.grei.com/\textsuperscript{16}

\textsuperscript{17} http://www.daml.org/2003/04/dql/dql\textsuperscript{17}

\textsuperscript{18} http://www.daml.org/tools/\textsuperscript{18}
1.4 OWL

As was commented in the introduction to this chapter, OWL (Dean and Schreiber, 2004) is the result of the work of the W3C Web Ontology (WebOnt) Working Group, which was formed in November 2001. This language derives from and supersedes DAML+OIL. It covers most of DAML+OIL features and renames most of its primitives. As the previous languages, OWL is intended for publishing and sharing ontologies in the Web.

OWL has a layered structure, as OIL had. OWL is divided into three layers: OWL Lite, OWL DL and OWL Full. In this section, we will explain how to build an ontology in OWL DL, and will point out how it differs from OWL Lite.

Like DAML+OIL, OWL is built upon RDF(S). Therefore, some RDF(S) primitives are reused by OWL, and OWL ontologies are written either in XML or with the triples notation for RDF. We will use the XML syntax for all the examples in this section.

As OWL is derived from DAML+OIL it shares many features with that language. The main differences between OWL and DAML+OIL are the following:

- OWL does not include qualified number restrictions (daml:hasClassQ, daml:cardinalityQ, daml:maxCardinalityQ, and daml:minCardinalityQ).

- OWL permits defining symmetric properties, which were not considered in DAML+OIL, with the primitive owl:SymmetricProperty.

- OWL does not rename the RDF(S) primitives reused by the language, as happened in DAML+OIL. For instance, rdfs:subClassOf, rdfs:subPropertyOf, etc.

- In OWL many DAML+OIL primitives have been renamed. For example, the primitive damhtoClass has been renamed as owl:allValuesFrom.

- OWL does not include the primitive daml:disjointUnionOf, since it can be effected by combining owl:unionOf and owl:disjointWith.

Other minor differences exist between both languages and they are explained in detail by Dean and Schreiber (2004).

1.4.1 Knowledge representation

An ontology in OWL starts with the declaration of the RDF root node. In this node we must include the namespaces for the RDF, RDFS and OWL KR ontologies. If XML Schema datatypes are used, it may be helpful to include a namespace for XML Schema, which is usually prefixed as xsd (and which points to the newest URL of XML Schema, as in RDF(S)).
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:owl="http://www.w3.org/2002/07/owl#">

As in other languages, the order in which definitions appear in OWL ontologies is not relevant since it is based in RDF(S). However, OWL ontologies usually define the header of the ontology first and then the ontology terms.

The header of an OWL ontology may include: the ontology documentation (rdfs:comment); the ontology version (owl:versionInfo); and the imported ontologies (owl:imports). As we did with our DAML+OIL ontology, we will import ontologies about units and datatypes. It is not necessary (nor recommended) to import the OWL KR ontology. The ontology header may also include information about version control with the primitives owl:backwardCompatibleWith, owl:incompatibleWith, and owl:priorVersion, which were described in Chapter 2. Inside the ontology we can also find definitions of deprecated classes and properties with the owl:DeprecatedClass and owl:DeprecatedProperty primitives.

<owl:Ontology rdf:about=""/>
<owl:versionInfo>1.0</owl:versionInfo>
<rdfs:comment>Sample ontology for travel agencies</rdfs:comment>
<owl:imports
rdf:resource="http://delicias.dia.fi.upm.es/owl/units"/>
<owl:imports
rdf:resource="http://www.w3.org/2001/XMLSchema"/>
</owl:Ontology>

Concepts are known as classes in OWL and are created with the primitive owl:Class. Besides its name, a class may also contain its documentation (with rdfs:comment) and any number of expressions with the following list of primitives:

- rdfs:subClassOf contains class expressions. It allows defining the superclasses of the class.
- owl:disjointWith asserts that the class cannot share instances with the class expression in this primitive.
- owl:equivalentClass also contains class expressions. This primitive defines necessary and sufficient conditions for the class (i.e., it is used for redefining concepts already defined).
- owl:oneOf defines a class by enumerating exhaustively all its instances. This is a way to define a class extensionally.
- owl:intersectionOf, owl:unionOf and owl:complementOf define a class expression as a conjunction, a disjunction, or a negation of other class expressions respectively.
The first two primitives (*rdfs:subClassOf* and *owl:disjointWith*) define necessary conditions for the class (they can be used in the definition of primitive concepts), while the rest of primitives define necessary and sufficient conditions for the class (that is, they are used to define defined concepts). In OWL Lite the only primitives that can be used are *rdfs:subClassOf*, *owl:equivariantClass*, and *owl:intersectionOf*. In all the cases they can only be used with class identifiers and property restrictions.

According to the DL terminology, OWL is a SHOIN language. This means that class expressions can be built with the following constructors:

- Conjunction (*owl:intersectionOf*), disjunction (*owl:unionOf*), and negation (*owl:complementOf*) of class expressions.
- Collections of individuals (*owl:oneOf*).
- Property restrictions (*owl:Restriction*). They contain a reference to the property to which the restriction is applied with the primitive *owl:onProperty* and an element for expressing the restriction. The following restrictions can be applied to properties: value restriction (*owl:allValuesFrom*), existential restriction (*owl:someValuesFrom*), role fillers (*owl:hasValue*), and number restriction (*owl:cardinality*, *owl:maxCardinality*, *owl:minCardinality*).
- Besides, role expressions can express inverse roles (*owl:inverseOf*) and role hierarchies (*rdfs:subPropertyOf*).

Therefore the grammar to build class expressions in OWL is very similar to that of DAML+OIL, as shown below. Note that Boolean expressions and enumerations are enclosed in `<owl:Class>`..`</owl:Class>` tags.

```
class-name |<owl:Class> boolean-expr </owl:Class> |<owl:Class>
<owl:oneOf rdf:parseType="Collection">
  instance-expr+
</owl:oneOf>
</owl:Class> |

<owl:Restriction>
  <owl:onProperty> property-name </owl:onProperty>
  restriction-expr
</owl:Restriction>
```

*boolean-expr* is defined as follows:

```
<owl:intersectionOf>
  class-expr class-expr+
</owl:intersectionOf> |<owl:unionOf> class-expr class-expr+ </owl:unionOf> |<owl:complementOf> class-expr </owl:complementOf>
```
instance-expr represents an instance of a class. It is defined as follows:

\[ \text{<owl:Thing rdf:resource="\"instanceName\"/}> \]

restriction-expr represents a restriction on a property when applied to the class. It can be any of the following (note that qualified number restrictions cannot be defined in OWL, in contrast with DAML+OIL):

\[ \text{<owl:allValuesFrom class-expr></owl:allValuesFrom> | } \]
\[ \text{<owl:someValuesFrom class-expr></owl:someValuesFrom> | } \]
\[ \text{<owl:hasValue instance-name></owl:hasValue> | } \]
\[ \text{<owl:cardinality rdf:datatype="\&xsd;nonNegativeInteger">non-neg-integer</owl:cardinality> | } \]
\[ \text{<owl:maxCardinality rdf:datatype="\&xsd;nonNegativeInteger">non-neg-integer</owl:maxCardinality> | } \]
\[ \text{<owl:minCardinality rdf:datatype="\&xsd;nonNegativeInteger">non-neg-integer</owl:minCardinality>
\]

In OWL Lite class expressions can only contain class names and property restrictions. The primitive \text{owl:hasValue} cannot be used in property restrictions. Besides, the primitives \text{owl:allValuesFrom} and \text{owl:someValuesFrom} only contain class identifiers or named datatypes, and cardinality restrictions only take the values 0 or 1. OWL DL does not impose any of these restrictions.

The definition of the primitive concept Travel is shown below. The concept Travel is defined as an intersection of several restrictions using \text{rdfs:subClassOf}: a travel has exactly one arrival date and one departure date (whose type is date, which is defined as an XML Schema datatype), at most, a single fare (whose type is currencyQuantity, which is defined in the units ontology) and this travel may be made with zero, one, or several companies.

\[ \text{<owl:Class rdf:ID="Travel">} \]
\[ \text{<rdfs:comment>A journey from place to place</rdfs:comment>} \]
\[ \text{<rdfs:subClassOf}> \]
\[ \text{<owl:Restriction>} \]
\[ \text{<owl:onProperty rdf:resource="\#arrivalDate"/}> \]
\[ \text{<owl:allValuesFrom rdf:resource="\&xsd;date"/>} \]
\[ \text{</owl:Restriction> } \]
\[ \text{<rdfs:subClassOf>} \]
\[ \text{<owl:Restriction>} \]
\[ \text{<owl:onProperty rdf:resource="\#departureDate"/}> \]
\[ \text{<owl:allValuesFrom rdf:resource="\&xsd;date"/>} \]
\[ \text{</owl:Restriction> } \]
\[ \text{</rdfs:subClassOf> } \]
\[ \text{<owl:Restriction>} \]
\[ \text{<owl:onProperty rdf:resource="\#arrivalDate"/}> \]
\[ \text{<owl:cardinality rdf:datatype="\&xsd;nonNegativeInteger">1</owl:cardinality> } \]
\[ \text{</owl:Restriction> } \]
\[ \text{</rdfs:subClassOf> } \]
\[ \text{<owl:Restriction>} \]
\[ \text{<owl:onProperty rdf:resource="\#departureDate"/}> \]
\[ \text{<owl:allValuesFrom rdf:resource="\&xsd;date"/>} \]
\[ \text{</owl:Restriction> } \]
\[ \text{</rdfs:subClassOf> } \]
The definition of the defined concept Flight is shown below. A flight is a kind of travel. It has an instance attribute called flightNumber, which must have exactly one value of type integer. Finally, it has also another instance attribute called transportMeans, whose value is "plane". This last restriction cannot be expressed in OWL DL, since owl:hasValue cannot be used in this version of the language. As occurred with DAML+OIL, we could have represented this attribute as a class attribute by attaching it directly to the definition of this concept. However, this part of the definition would not have been interpreted by OWL tools.
Equivalence between classes can be defined with the primitive `owl:equivalentClass`. For instance, `TravelByAir` is a synonym of `Flight` in our ontology, as can be seen below:

```xml
<owl:Class rdf:ID="TravelByAir">
    <owl:equivalentClass rdf:resource="#Flight"/>
</owl:Class>
```

In the previous definitions, we have explored how to represent restrictions on instance attributes in OWL, expliciting their type and cardinality. We have also explored how to represent restrictions on class attributes expliciting their values. Concept attributes must be defined as properties in the ontology. Like in DAML+OIL, in OWL there are two types of properties: `owl:ObjectProperty`, whose range is a class, and `owl:DatatypeProperty`, whose range is a datatype.

To define a property we may explicit its domain and range with the primitives `rdfs:domain` and `rdfs:range`. The primitive `rdfs:range` can refer to a class expression in OWL DL and only to a class identifier or to a named datatype in OWL Lite. We can state that a property is a subproperty of other properties with the primitive `rdfs:subPropertyOf`. Finally, we can express equivalence between properties with `owl:equivalentProperty`, and inverse properties with `owl:inverseOf`. Now we present the attribute definitions `arrivalDate`, `departureDate`, `companyName` and `singleFare` of the concept `Travel`, and the attribute definitions `flightNumber` and `transportMeans` of the concept `Flight`. We can see that these definitions have the same structure than the ones for DAML+OIL.

```xml
<owl:DatatypeProperty rdf:ID="arrivalDate">
    <rdfs:domain rdf:resource="#Travel"/>
    <rdfs:range rdf:resource="&xsd;date"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="departureDate">
    <rdfs:domain rdf:resource="#Travel"/>
    <rdfs:range rdf:resource="&xsd;date"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="companyName">
    <rdfs:domain rdf:resource="#Travel"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:ID="singleFare">
    <rdfs:domain rdf:resource="#Travel"/>
</owl:ObjectProperty>
```
We have already seen some examples of how to create concept taxonomies in OWL using `rdfs:subclassOf` for primitive concepts, and `owl:intersectionOf`, `owl:unionOf`, or `owl:equivalentClass` for defined concepts. Disjoint knowledge can be expressed with the `owl:disjointWith` primitive. We present below the definitions of the disjoint concepts AA7462, AA2010, and AA0488. Like in DAML+OIL, the `owl:disjointWith` primitive is symmetric; hence, three statements would be enough to state the disjointness between these classes. Note that this knowledge cannot be represented in OWL Lite since the primitive `owl:disjointWith` is not defined for this version of the language.

```xml
<owl:Class rdf:ID="AA7462">
  <rdfs:subClassOf rdf:resource="#AmericanAirlinesFlight"/>
  <owl:disjointWith rdf:resource="#AA2010"/>
  <owl:disjointWith rdf:resource="#AA0488"/>
</owl:Class>

<owl:Class rdf:ID="AA2010">
  <rdfs:subClassOf rdf:resource="#AmericanAirlinesFlight"/>
  <owl:disjointWith rdf:resource="#AA0488"/>
</owl:Class>

<owl:Class rdf:ID="AA0488">
  <rdfs:subClassOf rdf:resource="#AmericanAirlinesFlight"/>
</owl:Class>
```

Similarly to DAML+OIL, exhaustive decompositions can be represented with the `owl:unionOf` primitive. Regarding partitions, we commented in the introduction to this language that the DAML+OIL primitive `daml:disjointUnionOf` has not been included in OWL. The reason for this is that partitions can be expressed by combining the `owl:disjointWith` and the `owl:unionOf` primitives, though they result in a much longer and less readable definition. As with disjoint decompositions, exhaustive decompositions and partitions cannot be represented in OWL Lite either. Let us see now the definition of the partition of the concept `Location` in the concepts `NorthAmericanLocation`, `SouthAmericanLocation`, `EuropeanLocation`, `AsianLocation`, `AfricanLocation`, `AustralianLocation`, and `AntarcticLocation`. We must define that `Location` is the union of those seven classes.

```xml
<owl:Class rdf:ID="Location">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:resource="#EuropeanLocation"/>
  </owl:unionOf>
</owl:Class>
```
We must also define the six classes as disjoint with each other. We show below, as an example, the definition of the class EuropeanLocation.

```xml
<owl:Class rdf:ID="EuropeanLocation">
  <rdfs:subClassOf rdf:resource="#Location"/>
  <owl:disjointWith rdf:resource="#NorthAmericanLocation"/>
  <owl:disjointWith rdf:resource="#SouthAmericanLocation"/>
  <owl:disjointWith rdf:resource="#AsianLocation"/>
  <owl:disjointWith rdf:resource="#AfricanLocation"/>
  <owl:disjointWith rdf:resource="#AustralianLocation"/>
  <owl:disjointWith rdf:resource="#AntarcticLocation"/>
</owl:Class>
```

Binary relations are defined with the primitive `owl:ObjectProperty`. The global domain and range of a relation can be explicitly defined with the `rdfs:domain` and `rdfs:range` primitives, which can contain any class expression (except for OWL Lite, in which they can only contain class identifiers, or named datatypes for `rdfs:range`). However, as in DAML+OIL, it is not compulsory to define explicitly the global domain and range of a property. Instead, these restrictions can be defined locally inside class definitions with property restrictions, as presented in some of the concept definitions above. Below we present the definition of the relation `departurePlace` that can be applied to the concept `Ship` and the concept `Flight`, and that has different ranges for each of the concepts to which it is applied (Port and Airport respectively):

```xml
<owl:ObjectProperty rdf:ID="departurePlace">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf>
        <owl:Class rdf:resource="#Flight"/>
        <owl:Class rdf:resource="#Ship"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="#Location"/>
</owl:ObjectProperty>
```
Besides, in OWL we can define property hierarchies (with the rdfs:subPropertyOf primitive), we can state equivalences between properties (with the owl:equivalentProperty primitive), and we can assert the inverse of a property (with the owl:inverseOf primitive).

In addition to using owl:ObjectProperty to define binary relations, we can provide additional logical information about it with the following primitives:

- owl:TransitiveProperty. This primitive states that the relation is transitive.
- owl:SymmetricProperty. This primitive states that the relation is symmetric.

We can also state global cardinality restrictions on all kinds of properties (either object properties or datatype properties):

- owl:FunctionalProperty. This primitive states that the relation or attribute has only one value when applied to an instance of a concept of this domain. This primitive can be used for object properties (binary relations) and for datatype properties (concept attributes).
- owl:InverseFunctionalProperty. This primitive declares that the property is unambiguous, that is, for each instance of its range there is, at most, one instance of its domain that can take it. Consequently, this primitive can be used only with object properties. However, in OWL Full it can also be used with datatype properties.
Concerning higher arity relations, they must be defined as concepts in OWL (like in DAML+OIL). Below we present the definition for the ternary relation connects:

```xml
<owl:Class rdf:ID="connects">
  <rdfs:comment>A road connects two cities</rdfs:comment>
</owl:Class>
<owl:ObjectProperty rdf:ID="firstArgument">
  <rdfs:domain rdf:resource="#connects"/>
  <rdfs:range rdf:resource="#Location"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="secondArgument">
  <rdfs:domain rdf:resource="#connects"/>
  <rdfs:range rdf:resource="#Location"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="thirdArgument">
  <rdfs:domain rdf:resource="#connects"/>
  <rdfs:range rdf:resource="#RoadSection"/>
</owl:ObjectProperty>
```

In OWL we cannot express the constraint that cities are not part of each other and that the road starts in one of the cities and ends in the other.

Functions are not components of the OWL knowledge model, though binary functions can be represented with the `owl:FunctionalProperty` primitive previously discussed. Higher arity functions cannot be represented in this language.

Formal axioms are also not components of the OWL knowledge model.

Finally, instances are defined using only RDF vocabulary, as occurred in DAML+OIL. In OWL, we must use the attribute `rdf:datatype` to express the value type of datatype properties, as explained in the RDF(S) section. Therefore, the definition of our instance of flight AA7462 is as follows:

```xml
<AA7462 rdf:ID="AA7462Feb082002">
  <singleFare rdf:resource="&units;USDollar300"/>
  <departureDate rdf:datatype="&xsd;date">2002-02-08</departureDate>
  <arrivalPlace rdf:resource="#Seattle"/>
</AA7462>
```

Claims could also be represented in the same way as they were in RDF(S), and we can also assert that two instances are equivalent (with `owl:sameIndividualAs` and `owl:sameAs` indistinctly) or different (with `owl:differentFrom`). In addition, we can express that a set of individuals are all different from each other (with `owl:AllDifferent`). Note that these primitives are needed because OWL does not assume the unique name assumption for identifiers. This means that two terms with different identifiers can represent the same individual. Hence if two terms refer to different individuals we need to explicitly declare that both terms are different. For instance, the following definition states that in our ontology the identifiers Seattle, NewYork and Madrid refer to different individuals:

```xml
<owl:AllDifferent>
```

```xml
```
I.4.2 Reasoning mechanisms

The model-theoretic semantics of OWL is described by Patel-Schneider and colleagues (2004). This semantics is described in two different ways: as an extension of the RDF(S) model theory and as a direct model-theoretic semantics of OWL. Both of them have the same semantic consequences on OWL ontologies, and they are based on the DAML+OIL model-theoretic semantics, taking into account the differences between both languages.

Like DAML+OIL, OWL allows including any additional statements (RDF triples) in its ontologies apart from those explicitly defined in the language. However, OWL is silent on the semantic consequences (or lack thereof) of such additional triples.

A set of test cases has been defined by Carroll and De Roo (2004) including entailment tests, non-entailment tests, consistency tests, inconsistency tests, etc. They illustrate the correct usage of the OWL and the formal meaning of its constructs.

Due to its similarities with OIL and DAML+OIL, inference engines used for these languages (FaCT, RACER, TRIPLE, etc.) can be easily adapted for reasoning with it. There are not many inference engines available yet for reasoning with OWL, but we foresee that there will be soon. A reasoning engine already available is Euler.19

As with other languages, these inference engines will permit performing automatic classifications of OWL ontology concepts, and detecting inconsistencies in OWL concept taxonomies.

Furthermore, we can say that multiple inheritance is allowed in OWL ontologies (as we have discussed when describing how to create class expressions). In the semantics of OWL, however, there is no explanation on how conflicts in multiple inheritance can be solved. Constraint checking can be performed on the values of properties and their cardinalities.

OWL assumes monotonic reasoning, even if class definitions or property definitions are split up in different Web resources. This means that facts and entailments declared explicitly or obtained with inference engines can only be added, never deleted, and that new information cannot negate previous information.

19 http://www.agfa.com/w3c/euler/
As with DAML+OIL, many tools will be available for authoring OWL ontologies; tools capable of editing RDF(S) ontologies can also be used for developing OWL ontologies provided that the ontology developer uses the OWL KR primitives. In addition, RDF(S) query engines, storage systems, and parsers can be employed to manage OWL ontologies since they can be serialized in RDF(S). Finally, we must add that systems that transform DAML+OIL ontologies into OWL ontologies are already available\textsuperscript{20}.

\textsuperscript{20} http://www.mindswap.org/2002/owl.html
Annex II
Detailed description of ontology tools: Protégé-2000 and WebODE

This annex presents a detailed description of the ontology tools that have been used for the experimentation of this thesis: Protégé-2000 and WebODE. These descriptions have been extracted from [Gómez-Pérez et al., 2003].

II.1 PROTÉGÉ-2000

Protégé-2000¹ (Noy et al., 2000) is the latest version of the Protégé line of tools, created by the Stanford Medical Informatics (SMI) group at Stanford University. The first Protégé tool was created in 1987 (Musen, 1989); its main aim was to simplify the knowledge acquisition process for expert systems. To achieve this objective, it used the knowledge acquired in previous stages of the process to generate customized forms for acquiring more knowledge. Since then, Protégé has gone through several releases and has focused on different aspects of knowledge acquisition (knowledge bases, problem solving methods, ontologies, etc.), the result of which is Protégé-2000. The history of the Protégé line of tools was described by Gennari and colleagues (2003). It has around 7000 registered users.

Protégé-2000 is oriented to the task of ontology and knowledge-base development. It is freely available for downloading under the Mozilla open-source license. There are currently two active versions with some differences between each other: 1.8 (April 2003) and 2.0 (November 2003).

Architecture

Protégé-2000 is a Java-based standalone application to be installed and run in a local computer. The core of this application is the ontology editor, described further.

¹ http://protege.stanford.edu/
Protégé-2000 has an extensible architecture for creating and integrating easily new extensions (aka plug-ins). These extensions usually perform functions not provided by the Protégé-2000 standard distribution (other types of visualization, new import and export formats, etc.), implement applications that use Protégé-2000 ontologies, or allow configuring the ontology editor. Most of these plug-ins are available in the Protégé Plug-in Library\(^2\), where contributions from many different research groups can be found.

We now describe the three groups of plug-ins that can be developed for Protégé-2000 with actual examples of such types of plug-ins:

- **Tab plug-ins.** These are the most common types in Protégé-2000, and provide functions that are not covered by the standard distribution of the ontology editor. To perform their task, tab plug-ins extend the ontology editor with an additional tab so that users can access its functions from it. The following functions are covered by some of the plug-ins available: ontology graphical visualization (Jambalaya tab and OntoViz tab), ontology merge and versioning (PROMPT tab), management of large on-line knowledge sources (UMLS and WordNet tabs), OKBC ontology access (OKBC tab), constraint building and execution (PAL tab), and inference engines using Jess (Friedman-Hill, 2003), Prolog, FLogic, FaCT, and Algernon\(^3\) (Jess, Prolog, FLORA, OIL, and Algernon tabs respectively).

- **Slot widgets.** These are used to display and edit slot values without the default display and edit facilities. There are also slot widgets for displaying images, video and audio, and for managing dates, for measurement units, for swapping values between slots, etc.

- **Backends.** These enable users to export and import ontologies in different formats: RDF Schema, XML, XML Schema, etc. There is a backend for storing and retrieving ontologies from databases so that not only ontologies can be stored as CLIPS files (the default storage format used by Protégé-2000) but they can also be stored in any database JDBC compatible. Recently a backend to export and import ontologies in XMI\(^4\) has been made available.

**Knowledge model**

Protégé-2000 knowledge model is based on frames and first order logic. It is OKBC compatible, which means that the main modeling components of Protégé-2000 are classes, slots, facets and instances. Classes are organized in class hierarchies where multiple inheritance is permitted and

\(^2\) [http://protege.stanford.edu/plugins.html](http://protege.stanford.edu/plugins.html)
\(^3\) [http://www.cs.utexas.edu/users/qr/algy/](http://www.cs.utexas.edu/users/qr/algy/)
\(^4\) XML Metadata Interchange, used to exchange UML models with XML.
slots can also be organized in slot hierarchies. The knowledge model allows expressing constraints in the PAL language, which is a subset of KIF, and allows expressing metaclasses, which are classes whose instances are also classes.

**Classes** in Protégé-2000 can be concrete or abstract. The former may have direct instances while the latter cannot have them; instances of the class must be defined as instances of any of its subclasses in the class taxonomy.

**Slots** are global to the ontology (two different slots cannot have the same name in an ontology) and can be constrained in the classes to which they are attached. For instance, we can define the global slot age, which refers to the age of a traveler. When attached to the class Young Traveler, the minimum and maximum values for this slot are 16 and 30 years-old respectively, and when applied to the class Adult Traveler, the minimum and maximum values for this slot are 30 and 65 years-old respectively.

Like in OKBC, Protégé-2000 also distinguishes between two types of slots: template and own slots. The standard facets for template slots in Protégé-2000 are: NL documentation, allowed values, minimum and maximum cardinality, default values, inverse slot and template-slot values. Other facets can be created and attached to the slots to describe them. For instance, in our travel ontology we could add the facet currency to the slot price to express the currency in which the value of that slot is specified.

Protégé-2000's knowledge model can easily be extended by means of metaclasses. Metaclasses are defined as classes whose instances are also classes. They are used as templates to create other classes in Protégé-2000. This means that the template slots defined at the metaclass are converted into own slots in the classes that are instances of the metaclass. These classes must fill the values for their own slots. For example, in our travel ontology we know that the class AA7462 has the own slot company Name with the value "American Airlines". A similar situation appears with the classes AA2010, AA0488, etc., and their own slot company Name. To represent such knowledge in Protégé-2000, we can create a metaclass Company Flight with the template slot company Name, and create the previous classes as instances of that metaclass. These classes will have an own slot company Name where we will input its value. Figure II.1 shows the metaclass Company Flight with its template slot company Name, and the form for the class AA7462, which is an instance of that metaclass and must fill the value of its own slot company Name.
Protégé-2000 ontologies can be imported into other Protégé-2000 ontologies in two different ways: (a) by including them as external terms, that is, as terms that have been defined in another ontology and cannot be edited; and (b) by merging them, that is, by copying their definitions in the current ontology so that they can be redefined.

**Ontology editor**

Protégé-2000's ontology editor browses and edits the ontology's class taxonomy using a tree structure, defines global slots, attaches slots to classes, creates class instances, etc. It provides common search, copy & paste and drag & drop functions, among others, and also different pop-up menus according to the type and features of the ontology component being edited.

In Figure II.2 we can see a snapshot of the ontology editor for editing the concept **Travel** of our travel ontology. The left frame shows the class taxonomy of the ontology. In this frame we can observe that **Travel** is a subclass of the predefined class **THING**. In Protégé-2000, all the top classes of a class taxonomy must be subclasses of this predefined class. The main frame shows a form where the ontology developer can fill in the information about the class that we are editing. Besides the class name, we can provide its role (concrete or abstract, depending on
whether we can create direct instances of the class or not), its documentation, its applicable constraints, and its template slots. The "Template Slots" box contains the slots attached to the class and the slots inherited by it through the class taxonomy. As shown in this figure, we have attached the following slots to the class Travel: arrival Date, departure Date, company Name, single Fare, arrival Place, departure Place, and :NAME (the last slot is used to identify instances of the class Travel).

Figure II.2. Edition of the concept Travel with Protégé-2000.

Figure II.3 is a snapshot of the ontology editor for editing the slot arrival Date of our travel ontology. The left frame shows the hierarchy of the slots already defined in the ontology (a slot can be defined as a subslot of another). Besides the domain-dependent slots defined by the ontology developer for a domain ontology there are other predefined slots that are applied to other slots and classes of the standard knowledge model of Protégé-2000 (:ANNOTATED-INSTANCE, :ANNOTATION-TEXT, etc.). The main frame shows the form to be filled in when we define a domain-dependent slot. It contains fields for its name, NL documentation, value type, minimum and maximum cardinalities, minimum and maximum values for numeric slots, template and default values, and the name of the inverse slot, if any.
Slots in Protégé-2000 can be used to specify attributes and ad hoc relations between classes, and this distinction is expressed in their slot value type. The former are defined as a simple datatype (integer, float, string, and Boolean) or as an enumerated datatype (symbol). The latter are defined as class instances or classes. If we define the slot value type as any, it can have any of the previous value types. If the value type of a slot is Instance or Class, a box appears in the slot definition form to include its destination classes.

When we attach slots to a class (this is performed in the class edition form, such as the one presented in figure II.2), we can override some of the features that have been defined globally for them, as we commented in the knowledge model section. The form used to redefine them is similar to that of figure II.3.

One of the outstanding features of the Protégé-2000 ontology editor, when compared with other ontology editors, is that we can design the screen layouts used to create instances. Ontology developers can select which kind of forms will be presented, where the form fields will be located for each slot, which slot widgets they want to use for each slot, etc. Figure II.4 is a screenshot of Protégé-2000 for editing the form corresponding to the instances of the class Travel.
The Protégé-2000 ontology editor also contains a Queries tab. With this tab users can create queries about instances that have or have not a specific value for a slot, about instances whose slot value is greater or lower than a specific number, etc. Queries are stored in a query library and can be combined.

The PAL tab checks which PAL constraints are violated by the ontology instances. Figure II.2 shows that the PAL constraint "No train from USA to Europe" is attached to the class Travel. Other inference engines integrated as tab plug-ins in Protégé-2000 are FLORA (a XSB\textsuperscript{5} engine for FLogic's ontologies), Prolog, and Jess.

The ontology editor generates different types of ontology documentation: HTML documents and ontology statistics. There are also plug-ins for visualizing ontologies graphically such as the OntoViz and Jambalaya tab plug-ins.

Other tab plug-ins cover other functions. We recommend visiting the Protégé-2000 plug-in library for updated and extra information about them.

An ontology library available for downloading can be found in the Protégé-2000 Web site with ontologies such as the UNSPSC ontology, the Dublin core, etc. Since Protégé-2000 can import ontologies in several formats, it can import ontologies from other ontology libraries such as the DAML ontology library\textsuperscript{6}, the Ontolingua Server ontology library, etc.

Regarding collaborative ontology edition, Protégé-2000 does not support it, though some work is being done in this direction to migrate Protégé-2000 to a multi-user environment.

\textsuperscript{5} XSB is a logic programming and deductive database system.
\textsuperscript{6} http://www.daml.org/ontologies/
Interoperability

Once we have created an ontology in Protégé-2000, there are many ways to access Protégé-2000 ontologies from ontology-based applications.

All the ontology terms can be accessed with the Protégé-2000 Java API. Hence it is easy for ontology-based applications to access ontologies as well as use other functions provided by different plug-ins. This API is also available through a CORBA-based server so that remote clients can access Protégé-2000 ontologies.

Protégé-2000 ontologies can be exported and imported with some of the backends provided in the standard release or as plug-ins: RDF(S), XML, XML Schema, and XMI. As explained in other sections, once the corresponding output file has been generated and saved locally, it can be used by any local application capable of managing that format. In the case of XMI, the UML model translated can be used to obtain Java classes from it.

II.2 WebODE

WebODE\(^7\) (Corcho et al., 2002; Arpírez et al., 2003) is an ontological engineering workbench developed by the Ontology Group at Universidad Politécnica de Madrid (UPM). The current version is 2.0. WebODE is the offspring of the ontology design environment ODE (Blázquez et al., 1998), a standalone ontology tool based on tables and graphs, which allowed users to customize the knowledge model used for conceptualizing their ontologies according to their KR needs. Both ODE and WebODE give support to the ontology building methodology METHONTOLOGY.

Currently, WebODE contains an ontology editor, which integrates most of the ontology services offered by the workbench, an ontology-based knowledge management system (ODEKM), an automatic Semantic Web portal generator (ODESeW), a Web resources annotation tool (ODEAnnotate), and a Semantic Web services editing tool (ODEWS).

Architecture

WebODE has been built as a scalable, extensible, integrated workbench that covers and gives support to most of the activities involved in the ontology development process.

\(^7\) http://webode.dia.fi.upm.es/
WebODE is platform-independent as it is completely implemented in Java. To allow scalability and easy extensibility, it is supported by an application server so that services can be easily created and integrated in the workbench by means of a management console. One important advantage of using this application server is that we can decide which users or user groups may access each of the services of the workbench.

Figure II.5 illustrates the services currently available in the WebODE workbench. The core of the WebODE’s ontology development services are: the cache, consistency and axiom services, and the ontology access service (ODE API), which defines an API for accessing WebODE ontologies. One of the main advantages of this architecture is that these services can be accessed remotely from any other application or any other instance of the workbench. WebODE ontologies are stored in a relational database so they can manage huge ontologies quite efficiently. WebODE also provides backup management functions for the ontologies stored in the server.

The figure shows that the interoperability services are running on top of the ontology access service. These services import ontologies from XML, XCARIN, RDF(S), DAML+OIL, and OWL; and export ontologies to XML, FLogic, XCARIN, RDF(S), OIL, DAML+OIL, and OWL. Ontologies are also exported to languages that are not specifically created for defining ontologies such as Prolog, Jess, and Java. For instance, the Prolog export service is used as a basis of the WebODE’s inference engine.
Other middleware services such as WebPicker, ODEClean, and ODEMerge, which are described further, also use the WebODE ontology access API or the XML export/import services.

**Knowledge model**

Ontologies in WebODE are conceptualised with a very expressive knowledge model (Arpírez et al., 2001). This knowledge model is based on the reference set of intermediate representations of the METHONTOLOGY methodology (Fernández-López et al., 1999). Therefore, the following ontology components are included in the WebODE’s knowledge model: concepts and their local attributes (both instance and class attributes, whose type can be any XML Schema datatype); concept groups, which represent sets of disjoint concepts; concept taxonomies, and disjoint and exhaustive class partitions; ad hoc binary relations between concepts, which may be characterised by relation properties (symmetry, transitiveness, etc.); constants; formal axioms, expressed in first order logic; rules; and instances of concepts and relations.

In addition to the previous components, bibliographic references, synonyms, and abbreviations can be attached to any of the aforementioned.

The WebODE’s knowledge model allows referring to ontology terms defined in other ontologies by means of imported terms. Imported terms are identified with URIs and these are of two types: those available in another WebODE ontology, either in the same or in a different WebODE server (referred to as webode://WebODE_host/ontologies/ontology#name), and those identified by a different type of URI.

WebODE instances are defined inside instance sets. Thus, we can create different instantiations for the same ontology, which are independent from each other. For instance, we can instantiate our travel ontology in different instance sets, one for each travel agency using the ontology.

**Ontology editor**

The WebODE ontology editor is a Web application built on top of the ontology access service (ODE API). The ontology editor integrates several ontology building services from the workbench: ontology edition, navigation, documentation, merge, reasoning, etc.

Three user interfaces are combined in this ontology editor: an HTML form-based editor for editing all ontology terms except axioms and rules; a graphical user interface, called
OntoDesigner, for editing concept taxonomies and relations graphically; and WAB (WebODE Axiom Builder), for editing formal axioms and rules. We now describe them and highlight their most important features.

Figure II.6 is a screenshot of the HTML interface for editing instance attributes of the concept Travel of our travel ontology. The main areas of this interface are:

- The browsing area. To navigate through the whole ontology and to create new elements and modify or delete the existing ones.
- The clipboard. To easily copy and paste information between forms.
- The edition area. It presents HTML forms to insert, delete and update ontology terms (concepts, attributes, relations, etc.), and tables with knowledge about existing terms. Figure II.6 shows four attributes of the concept Travel: arrival Date, company Name, departure Date, and single Fare, and an HTML form to create a new instance attribute for this concept.

OntoDesigner eases the construction of concept taxonomies and ad hoc relations between concepts and allows defining views to highlight or customize the visualization of fragments of the ontology for each user.
A declarative approach to ontology translation with knowledge preservation

Concept taxonomies are created with the following set of predefined relations: 
Subclass-Of, Disjoint-Subclass-Partition \textit{(Disjoint-Decomposition)}, Exhaustive-Subclass-Partition \textit{(Partition)}, Transitive-Part-Of and Intransitive-Part-Of. 

Figure II.7 shows a view of our travel ontology in OntoDesigner, where we have selected the concepts Travel package, Luxury Trip, Economy Trip, Business Trip, Lodging, Location, and Transport, and several ad hoc and taxonomic relations between them. 

OntoDesigner can be used as part of the ODEClean evaluation module, as will be explained further.

The WebODE Axiom Builder (WAB) (Corcho et al., 2002) is a graphical editor for creating first order logic axioms and rules. In WebODE, formal axioms are used to model sentences that are always true while rules are normally included in the ontology to infer new knowledge. Figure II.8 shows an axiom in WAB that states that “every train that departs from a European location must arrive in a European location”. The buttons below the text box help write logical expressions with quantifiers (universal and existential) and logical connectives (negation, conjunction, disjunction, logical implication, and logical equivalence); and the lists below them are to easily include ontology concepts, their attributes and their ad hoc relations, and ontology constants. Users can also write directly the axiom expression with WAB.
Annex II. Detailed description of ontology tools

When an axiom is completely written in the text box, WAB checks that it uses the vocabulary contained in the ontology (checks that the concepts appearing in the axiom exist in the ontology, that the relations can be applied to those concepts, etc.). Then WAB transforms the axiom into Horn clauses through a skolemization process. If the axiom cannot be transformed into Horn clauses, WAB warns the user of this. These Horn clauses are then transformed into Prolog using primitives defined in the OKBC knowledge model so that can be used by the Prolog inference engine attached to WebODE, as described below. The Horn clause that corresponds to the axiom is shown on the bottom left of the figure, and the Prolog rule, which uses the OKBC primitives instance_of and value_facet_of, is shown on the bottom right.

Rules are similarly created with WAB. Figure II.9 shows a rule that states that “every trip that departs from Europe is arranged by the company Costa Cruises”. Rules are also checked, transformed into a Horn clause (shown on the bottom left of the figure), and then into a Prolog rule (shown on the bottom right of the figure).
A declarative approach to ontology translation with knowledge preservation

Figure II.9. Edition of a rule with WAB.

We now describe other ontology building services integrated in the ontology editor: the documentation service, ODEMerge, the OKBC-based Prolog inference engine, and ODEClean.

The WebODE ontology documentation service generates WebODE ontologies in different formats that can be used to document: HTML tables representing the Methontology's intermediate representations described in Section 3.3.5 and HTML concept taxonomies. Figure II.10 illustrates a fragment of the concept dictionary of our travel ontology, and figure II.11 shows the result of one of the HTML documentation formats generated with this service; this figure contains a fragment of the concept taxonomy of the ontology and specifies the attributes of each concept (preceded by a hollow bullet) and the ad hoc relations whose domain (symbol =>) or range (symbol <=) is each concept.

Figure II.10. A fragment of an intermediate representation generated by the WebODE ontology documentation service.
The WebODE merge service (ODEMerge) performs a supervised merge of concepts, attributes, and relationships from two ontologies built for the same domain. It uses natural language resources to find the mappings between concepts of both ontologies so as to generate the resulting ontology.

WebODE includes an inference engine that consists of a Prolog implementation of a subset of the OKBC protocol primitives. This engine uses the Ciao Prolog interpreter (Hermenegildo et al., 2000). Since WebODE ontologies can be translated into Prolog, the inference engine obtains an ontology in Prolog from the Prolog export service and loads it into the Prolog interpreter. With this process, the implemented OKBC primitives can build more complex Prolog programs for being executed in the Prolog interpreter for any purpose.

The inference engine can be accessed from the WebODE ontology editor with the user interface shown in figure II.12. In the figure, the middle text box shows the results of querying the inference engine of all the subclasses of the concept Travel in our ontology. It returns two answers to this query: the first is a list that contains the concepts Flight, Train Travel...
and Ship, which are the direct subclasses of the concept Travel; the second answer is also a list that contains all the ontology concepts that are direct and indirect subclasses of the concept Travel. As the rest of WebODE services, the inference engine can be executed not only from the user interface of the ontology editor but also by means of its Java API.

Currently, the WebODE inference engine is used for several purposes: querying ontology terms either with the predefined OKBC primitives or with user-defined Prolog programs; asserting new knowledge with the Prolog expressions generated by WAB; detecting inconsistencies in the ontology; and evaluating the ontology with ODEClean.

ODEClean (Fernández-López and Gómez-Pérez, 2002) is a service for evaluating concept taxonomies based on the OntoClean method (Welty and Guarino, 2001). This method relies on some philosophical notions such as rigidity, identity, dependency, and unity. Users with access to this service can edit the meta-properties of each concept and evaluate ontologies according to this method. Figure II.13 shows a screenshot of OntoDesigner after evaluating with ODEClean a class taxonomy with several errors.

ODEClean evaluation axioms are defined declaratively in Prolog instead of being hard-wired in the code of this service and are loaded in the inference engine when this service is invoked. This provides flexibility so that evaluation axioms can be easily changed if the OntoClean method is changed.

---

8 Please note that these concepts appear with their first letter in lowercase because if they started in uppercase they would be considered variables by the Prolog interpreter. This transformation is performed by the Prolog export service.
The WebODE workbench also provides several ontology evaluation functions: the ontology consistency service and the RDF(S), DAML+OIL, and OWL evaluation services.

The ontology consistency service shown in Figure II.5 provides constraint checking capabilities for the WebODE ontologies and is used by the ontology editor during the ontology building process. It checks type constraints, numerical values constraints, and cardinality constraints, and verifies concept taxonomies (i.e., external instances of an exhaustive decomposition, loops, etc.).

The RDF(S), DAML+OIL, and OWL evaluation services evaluate ontologies according to the evaluation criteria identified by Gómez-Pérez (2001). They detect errors in ontologies implemented in these languages and provide suggestions about better design criteria.

Finally, user groups can be created to collaborate in the edition of ontologies. Several users can edit the same ontology without errors by means of synchronization mechanisms.

**Interoperability**

There are several ways of using WebODE ontologies inside ontology-based applications.

First, they can be accessed from its Java API via a local service or application running on the same computer where the ontology server is installed. This API avoids accessing directly the relational database where ontologies are stored and it includes cache functions to accelerate the
access to ontology terms. WebODE ontologies can be accessed not only from inside the local server but also remotely with RMI (Remote Method Invocation) and Web services.

Second, ontology export services available in the workbench permit generating WebODE ontologies in XML and in several other ontology languages such as: RDF(S), OIL, DAML+OIL, OWL, XCARIN and FLogic. Translations into Prolog can be used similarly.

Third, ontologies can be transformed into Protégé-2000. So we can use them inside the Protégé-2000 ontology editor or use the interoperability capabilities provided by this tool.

Finally, WebODE ontologies can be transformed into Java. In this process, concepts are transformed into Java beans, attributes into class variables, ad hoc relations into associations between classes, etc., with their corresponding constructors and methods to access and update class variables, etc. This Java code can be used to create other Java applications, uploaded in rule systems like Jess, etc.
Annex III
BNF grammars of ODELex, ODESyntax, and ODESem

This annex presents the BNF grammars of the three languages that compose ODEDialect: ODELex, ODESyntax, and ODESem. These languages were described in detail in chapter 4.

III.1 NOTATION

The following notation will be used to describe the derivation rules of the ODELex, ODESyntax and ODESem grammars: words in italics will be used for non-terminal symbols, words in bold font will be used for terminal symbols, alternatives will be represented with the | symbol, optional elements will be enclosed in square brackets [ ], iterations of 0 or more items will be enclosed in braces { }, and ranges of values will be enclosed in parenthesis and separated by hyphens ( - ).

III.2 ODELex Grammar

(1) ODELexDocument :: {comment} %% [userCode] %% [declarations] %% [lexRules]

(2) userCode :: {comment} %%(javaCode)%

(3) declarations :: {comment | componentDecl | overlapDecl}

(4) componentDecl :: idComponent [%transient] [scopeDecl]

(5) scopeDecl :: %scope ( idScope {, idScope} )

(6) idScope :: idComponent | id

(7) overlapDecl :: no-overlap ( idComponent, idComponent {, idComponent} )
(8) lexRules :: {lexRule}

(9) lexRule :: ruleHeader CR init CR table CR repeated CR overlap CR

(10) ruleHeader :: % idComponent IDENTIFIER {idComponent IDENTIFIER}

(11) init :: INIT:{javaCode}

(12) table :: TABLE:{tableDecl}

(13) tableDecl :: (tableColumn , tableColumn)

(14) tableColumn :: [ idComponent , numberPosition , numberPosition]

(15) numberPosition :: % number | $ number

(16) repeated :: REPEATED:{transformation , transformation} }

(17) transformation :: tablePatternColumn => {javaCode} | default => {javaCode}

(18) tablePatternColumn :: [idComponent , numberPatternPosition

{ , numberPatternPosition}]

(19) numberPatternPosition :: % number |

(20) overlap :: OVERLAP:{javaCode}

(21) idComponent :: idFormat . id

(22) idFormat :: (A-Z){A-Z}

(23) id :: (A-Z,a-z){a-z,A-Z,0-9}

(24) comment :: /* textIncludingCR */ || textWithoutCR

(25) textIncludingCR :: {a-z,A-Z,0-9,CR}

(26) textWithoutCR :: {a-z,A-Z,0-9}

(27) number :: (1-9){0-9}

(28) CR :: \n
III.3 ODESyntax Grammar

(1) ODESyntaxDocument :: {comment} %% [userCode] %% [declarations]

%% [accessDeclsl] %% [updateDeclsl] %% [datatype]

(2) userCode :: {comment} %%(javaCode)%
Annex IV. BNF grammars of ODELex, ODESyntax, and ODESem

(3) declarations :: {comment | namespaceDecl | componentDecl }

(4) namespaceDecl :: %NAMESPACE id javaPackage ;

(5) componentDecl :: idComponent [scopeDecl] : javaClassID CR

(6) scopeDecl :: %scope ( idComponent , idComponent )

(7) accessDecl :: {accessDecl}

(8) accessDecl :: header CR [all CR] [individual CR] [information CR]

(9) header :: % idComponent IDENTIFIER (idComponent IDENTIFIER)

(10) all :: ALL: {functionDecl ; functionDecl }

(11) individual :: INDIVIDUAL: {functionDecl ; functionDecl }

(12) functionDecl :: number : id (parameters) : javaClassID array

(13) parameters :: % number {, % number } | λ

(14) array :: [] | λ

(15) information :: INFORMATION: {informDecl ; informDecl }

(16) informDecl :: id : id [ (parameters) ] : javaClassID array

(17) updateDecl :: {updateDecl}

(18) updateDecl :: header CR [create CR] [remove CR] [add CR]

 [removeall CR] [removeindividual CR]

(19) create :: CREATE: {createremDecl ; createremDecl }

(20) remove :: REMOVE: {createremDecl ; createremDecl }

(21) createremDecl :: number : id (parameters)

(22) add :: ADD: {addremDecl ; addremDecl }

(23) removeall :: REMOVEALL: {addremDecl ; addremDecl }

(24) removeindividual :: REMOVEINDIVIDUAL: {addremDecl ; addremDecl }

(25) addremDecl :: id : id (parameters)

(26) datatype :: {datatypeTransf}

(27) datatypeTransf :: " datatypeID " : " datatypeID " ; |

   default %1 : (datatypeID | %1 | {javaCode})
(28) idComponent :: idFormat . id

(29) idFormat :: (A-Z)(A-Z)

(30) id :: (A-Z,a-z)(a-z,A-Z,0-9)

(31) comment :: /* textIncludingCR */ // textWithoutCR

(32) textIncludingCR :: {a-z,A-Z,0-9,CR}

(33) textWithoutCR :: {a-z,A-Z,0-9}

(34) number :: (1-9)(0-9)

(35) CR :: 

(36) javaClassID :: id | id

(37) javaPackage :: id . { id .}

(38) datatypeID :: {a-z,A-Z,0-9;/,\#}

III.4 ODESem Grammar

(1) ODESemDocument :: {comment} %% [userCode] %% [declarations] %% [semRules]

(2) userCode :: {comment} %%(javaCode)%

(3) declarations :: {comment | ruleDecl}

(4) ruleDecl :: number : % id ;

(7) semRules :: {semRule | comment}

(8) semRule :: header CR lhs --> { rhs }

(9) header :: % id

(10) lhs :: idComponent id | NULL

(11) rhs :: { create | add | remove | removeAll | exec | ifThen | forEach
       | error | assign | functionCall} ;

(12) assign :: id = { create | functionCall | getComponents}

(13) functionCall :: id ( parameters )

(14) parameters :: id { , id } | λ
Annex IV. BNF grammars of ODELex, ODESyntax, and ODESem

(15) ifThen :: if ( condition ) [{} rhs ] else [{} rhs ]

(16) condition :: javaComparison | functionCall

(17) forEach :: forEach id IN var [{} rhs ]

(18) create :: CREATE ( idComponent , number , var {, var} )

(19) add :: ADD ( id , id , (create | var | getComponents) )

(20) remove :: REMOVE ( id , id , var )

(21) removeAll :: REMOVEALL ( id , id )

(22) exec :: EXEC ( % id , var )

(23) error :: ERROR ( errorMessage )

(24) errorMessage :: " textWithoutCR "

(25) var :: /c/ | idComponent

(26) getComponents :: GETCOMPONENT ( idComponent , id ) | GETALLCOMPONENTS ( idComponent , id )

(27) idComponent :: idFormat . id

(28) idFormat :: (A-Z){A-Z}

(29) id :: (A-Z,a-z){a-z,A-Z,0-9}

(30) comment :: /* textIncludingCR */ | // textWithoutCR

(31) textIncludingCR :: {a-z,A-Z,0-9,CR}

(32) textWithoutCR :: {a-z,A-Z,0-9}

(33) number :: (1-9){0-9}

(34) CR :: \n
Annex IV
Resumen amplio en español

En este anexo se incluye un resumen amplio de la tesis doctoral en español, tal y como requiere la Comisión de Doctorado de la Universidad Politécnica de Madrid.

IV.1 INTRODUCCIÓN

Las **ontologías** se usan ampliamente en las áreas de la Ingeniería de Conocimientos, la Inteligencia Artificial y la Informática, en aplicaciones relacionadas con la gestión de conocimientos, el procesamiento del lenguaje natural, el comercio electrónico, la integración de información inteligente, la recuperación de información, el diseño e integración de bases de datos heterogéneas, la bio-informática, la educación, y en nuevos campos como la Web Semántica y el Grid Semántico.

El término “Ontología” ha sido tomado de la Filosofía, donde se define como una explicación sistemática del ser. Durante las últimas décadas se ha comenzado a utilizar, en las tres áreas apuntadas en el párrafo anterior, como una solución al problema del cuello de botella que supone la adquisición de conocimientos en el desarrollo de sistemas basados en conocimientos. De manera más específica, este término se comenzó a utilizar en 1991, en el contexto de la iniciativa de DARPA “Knowledge Sharing Effort” [Neches et al., 1991]. En este iniciativa se propuso una nueva forma de construir sistemas basados en conocimientos, basada en la recopilación de componentes de conocimientos reutilizables, en lugar de en el desarrollo de bases de conocimientos desde cero. Con este enfoque, los sistemas serían capaces de compartir sus conocimientos declarativos, modelados mediante ontologías, y sus técnicas de resolución de problemas y servicios de razonamiento, y los desarrolladores de sistemas sólo se tendrían que preocupar de crear los conocimientos y razonadores específicos de sus sistemas.
Desde entonces, se han propuesto bastantes definiciones del término “ontología”, muchas de las cuales se recogen y explican en [Guarino and Giaretta, 1995] y [Gómez-Pérez et al., 2003]. Una de las definiciones más citadas es la propuesta en [Studer et al., 1998], que está basada en definiciones previas de Gruber [Gruber, 1993] y Borst [Borst, 1997]: “una ontología es una especificación formal y explícita de una conceptualización compartida”. El término conceptualización se refiere a un modelo abstracto de algún fenómeno del mundo, identificando sus elementos más relevantes. Explícito se refiere a que los tipos de conceptos utilizados y las restricciones sobre su uso se especifican claramente. Formal se refiere al hecho de que la ontología debe ser legible por una máquina. Finalmente, el aspecto más importante es el hecho de que la ontología debe ser compartida. Esto refleja la noción de que una ontología debe capturar conocimientos consensuados, es decir, aceptados por un grupo.


Una situación similar se da con las herramientas de ontologías: existen diversos editores y herramientas de gestión de ontologías que se pueden utilizar para construir ontologías. Entre ellas se pueden citar (en orden alfabético): KAON [Maedche et al., 2003], OilEd [Bechhofer et al., 2001], OntoEdit [Sure et al., 2002], Ontolingua Server [Farquhar et al., 1997], OntoSaurus [Swartout et al., 1997], Protégé-2000 [Noy et al., 2000], WebODE [Arpírez et al., 2003], y WebOnto [Domingue, 1998]. Como en el caso de los lenguajes, los modelos de conocimientos en los que se basan estas herramientas tienen sus propias capacidades expresivas y de razonamiento, dado que están basados en distintos paradigmas de representación, o combinaciones de los mismos. Además, las herramientas de ontologías normalmente permiten exportarlas e importarlas a o desde distintos lenguajes de ontologías.
Los componentes que se pueden utilizar para representar conocimientos en un lenguaje o herramienta normalmente dependen de los paradigmas de representación en que éstos están basados. Por ejemplo, los lenguajes o herramientas basados en combinaciones de marcos y lógica de primer orden permiten construir ontologías con los siguientes componentes [Gruber, 1993]: clases, relaciones, funciones, axiomas formales e instancias. En los basados en lógica descriptiva, los componentes que se pueden utilizar son [Baader et al., 2003]: conceptos, roles e individuos. En redes semánticas, son nodos y arcos entre los nodos.

El problema de la traducción de ontologías [Gruber, 1993] aparece cuando se decide reutilizar una ontología (o parte de ella) con una herramienta o lenguaje diferente a aquéllos en los que la ontología se encuentra disponible. Si los desarrolladores de ontologías deben acometer, de manera individual, la tarea de traducir e incorporar a sus sistemas las ontologías que necesitan, éstos necesitarán una gran cantidad de tiempo y esfuerzo para conseguir sus objetivos [Swartout et al., 1997]. Por esta razón, la amplia reutilización de ontologías en distintos contextos será posible siempre que se proporcionen servicios de traducción de ontologías entre lenguajes y/o herramientas.

En el caso concreto del intercambio de ontologías entre herramientas, este problema se conoce como problema de interoperabilidad. De hecho, la interoperabilidad se define como "la habilidad de dos o más sistemas o componentes para intercambiar información en una red heterogénea y para usar dicha información" [IEEE Std. 1430-1996, 1999]. En el área de las ontologías, esto significa que diferentes herramientas sean capaces de intercambiar sus ontologías.

En la actualidad existen diversos servicios de traducción de ontologías entre lenguajes y/o herramientas. La mayor parte de ellos se utilizan para importar en una herramienta las ontologías implementadas en un determinado lenguaje de ontologías o viceversa, es decir, para exportar ontologías desde las herramientas a lenguajes de ontologías. También existen algunos servicios de traducción entre lenguajes o entre herramientas.

Como se ha comentado anteriormente, las herramientas y lenguajes de ontologías pueden tener distintas capacidades expresivas y de razonamiento. Por esta razón, las traducciones entre ellos no son sencillas ni fácilmente reutilizables. Normalmente, para realizar una traducción entre dos formatos distintos se requiere tomar decisiones en distintos niveles, desde los más básicos (como, por ejemplo, transformar el nombre de un concepto de un formato a otro) hasta otros de
mayor nivel (como, por ejemplo, transformar una relación ternaria entre conceptos a un formato que sólo permite representar relaciones binarias).

Los servicios actuales de traducción entre formatos no suelen tener en cuenta esta estructura en niveles de las decisiones de traducción. Asimismo, las decisiones de traducción tomadas normalmente quedan ocultas dentro del código de dichos sistemas. Ambos aspectos hacen difícil entender cómo estos servicios funcionan.

Finalmente, la amplia variedad de capacidades expresivas y de razonamiento de herramientas y lenguajes hace difícil el intercambio de ontologías entre herramientas y/o lenguajes sin perder conocimientos de las ontologías de partida (es decir, preservando la semántica de la ontología), o sin perder o cambiar el significado deseado (es decir, preservando la pragmática de la ontología).

**IV.2 ESTADO DE LA CUESTIÓN**

Hasta el momento, la investigación realizada en el ámbito de la traducción de ontologías se ha enfocado principalmente al desarrollo de servicios de exportación e importación para herramientas de ontologías, aunque también se han realizado algunos esfuerzos en la traducción entre lenguajes o entre herramientas.

En esta sección se presenta el estado actual del área en lo que concierne a la clasificación de los problemas de traducción, enfoques y arquitecturas para la traducción, y tecnología de soporte a algunos de estos enfoques.

**IV.2.1 Caracterización de problemas de traducción**

Los trabajos más importantes sobre la caracterización de los problemas que aparecen en la traducción de ontologías se deben a Hans Chalupsky [Chalupsky, 2000], Michel Klein [Klein, 2001] y Jerome Euzenat [Euzenat, 2001]. Todos ellos se caracterizan por no ser trabajos específicos sobre el problema de la traducción, sino por estar enmarcados en el contexto de otros problemas más generales como la interoperabilidad semántica entre sistemas basados en ontologías, o la mezcla e integración de ontologías.

- [Chalupsky, 2000] define diversos niveles de problemas de traducción, dos de los cuales tienen que ver con las diferencias entre los formatos origen y destino: sintaxis y expresividad. El resto tienen que ver con la influencia del formato de origen en los acuerdos ontológicos tomados al modelar una ontología, y por tanto no son interesantes desde el punto de vista de la traducción de ontologías.
- [Klein, 2001] propone los niveles de sintaxis (equivalente al nivel de sintaxis propuesto en el caso anterior), y los de representación, semántica y expresividad de los formatos origen y destino (que se corresponden con el nivel de expresividad identificado anteriormente).

- Finalmente, [Euzenat, 2001] distingue los siguientes niveles de problemas: codificación, léxico, sintáctico, semántico y semiótico, y considera que las transformaciones se pueden realizar en distintos niveles. Sin embargo, sólo presenta algunas soluciones teóricas para resolver los problemas en el nivel semántico.

Todos estos trabajos se pueden asimismo situar en el contexto del estudio de lenguajes formales y de la teoría de signos ([Morris, 1938]), en los que se definen normalmente tres niveles en los lenguajes formales: sintáctico (que trata sobre cómo los símbolos se estructuran en el lenguaje), semántico (que trata sobre lo que significan o denotan dichos símbolos), y pragmático (que trata sobre cómo se usan e interpretan los símbolos en un determinado contexto).

### IV.2.2 Enfoques y arquitecturas de traducción

Respecto a las arquitecturas utilizadas para la traducción entre lenguajes y/o herramientas, se distinguen tres posibilidades [Euzenat y Stuckenschmidt, 2002]:

- Traducciones dos a dos (enfoque basado en mappings). Este es el enfoque más utilizado, y se basa en determinar (e implementar) mappings ad hoc entre los formatos origen y destino. Tiene el inconveniente de que para realizar traductores entre n lenguajes se necesita especificar $O(n^2)$ traductores.

- Traducciones con un lenguaje común de intercambio, denominado lenguaje pivot. En este caso, las traducciones entre un formato y el lenguaje de intercambio, y viceversa, se realizan del mismo modo que en el caso anterior, pero se consigue reducir la complejidad a $O(n)$ traductores. Los lenguajes KIF, RDF(S) y OWL han sido propuestos como lenguajes comunes de intercambio. Sin embargo, la experiencia ha demostrado que este enfoque no ha resultado exitoso.

- Familias de lenguajes [Euzenat y Stuckenschmidt, 2002]. Este enfoque permite resolver los problemas de traducción del nivel semántico cuando se utilizan formatos basados en lógica descriptiva, que deben ser expresados en el lenguaje DLML. Tiene en cuenta una relación de cobertura entre lenguajes, que es de orden parcial.

### IV.2.3 Tecnología de soporte a la traducción

Los servicios de traducción entre lenguajes y/o herramientas normalmente se crean utilizando lenguajes de programación de propósito general (Java, LISP, etc.), debido a que la tecnología
que da soporte a los enfoques y arquitecturas anteriores es insuficiente. Concretamente, existen dos herramientas específicas para la generación de traductores:

- Transmorpher [Euzenat and Tardiff, 2001]. Esta herramienta recibe un conjunto de ficheros XSLT con las transformaciones que se deben realizar entre lenguajes de lógica descriptiva, y realiza dichas transformaciones. Sólo se puede utilizar si los lenguajes de origen y destino están especificados en XML.

- OntoMorph [Chalupsky, 2000]. Esta herramienta permite especificar los mappings de transformación entre el origen y destino a través de reglas basadas en patrones. Las transformaciones se realizan en dos fases de reescritura, sintáctica y semántica, la última de las cuales requiere que la ontología sea transformada internamente en el lenguaje PowerLoom.

IV.3 PLANTEAMIENTO: PROBLEMAS Y OBJETIVOS

Como resultado del análisis del estado de la cuestión presentado anteriormente, se puede concluir que:

- No existe una propuesta unificada de los niveles que se deben utilizar para caracterizar los problemas de traducción, y en los enfoques propuestos hasta el momento no especifican cómo se deben tratar los problemas en cada uno de los niveles que identifican.

- Desde un punto de vista metodológico, no existen métodos ni técnicas que soporten de manera integrada la compleja tarea de construir servicios de traducción de ontologías. Los sistemas actuales no tienen en cuenta que las decisiones de traducción se pueden tomar en distintos niveles. Por consiguiente, las decisiones de traducción que implementan no son fáciles de entender y los sistemas son difíciles de mantener.

- Desde un punto de vista tecnológico, los sistemas que dan soporte para la generación de servicios de traducción imponen importantes restricciones relativas a los formatos de origen y destino que pueden tratar: OntoMorph sólo se puede aplicar a lenguajes basados en Lisp, mientras que Transmorpher sólo se puede aplicar a lenguajes basados en XML.

- No existen estudios detallados sobre los distintos enfoques utilizados hasta la fecha para la traducción de ontologías, ni sobre su impacto en la preservación de la semántica y de la pragmática de las ontologías transformadas, especialmente en el caso de traducciones cíclicas. Asimismo, los servicios de traducción existentes no suelen incluir documentación de qué conocimientos se pierden y cuáles se mantienen.
Teniendo en cuenta los problemas identificados anteriormente, esta tesis presenta las siguientes contribuciones al estado del arte actual en traducción de ontologías entre lenguajes y/o herramientas:

- Se propone una nueva caracterización de los problemas de traducción, basada en la teoría de signos de Morris y en la clasificación propuesta por Euzenat. Esta caracterización distingue los niveles léxico, sintáctico, semántico y pragmático. Asimismo, se describen los principales problemas que se pueden encontrar en dichos niveles de traducción teniendo en cuenta los lenguajes y herramientas de ontologías utilizados en la actualidad.

- Se propone un nuevo modelo para la construcción y mantenimiento de sistemas de traducción de ontologías. Este modelo se caracteriza, en primer lugar, por ser multicapa: las decisiones de traducción se pueden tomar en cualquiera de los cuatro niveles de traducción anteriores. Asimismo, el modelo propone representar las decisiones de traducción de manera declarativa, utilizando tres lenguajes de transformación: ODELex, ODESyntax y ODESem. Como parte de este modelo se propone un método para la construcción de sistemas de traducción de ontologías, que se compone de cuatro actividades: estudio de viabilidad, análisis de los formatos de origen y destino, diseño del servicio de traducción e implementación del servicio de traducción.

- Se analizan de manera detallada los enfoques de traducción de ontologías existentes desde las perspectivas de la preservación de la semántica y de la pragmática en la traducción. También se describe el ciclo de vida de las ontologías en procesos cíclicos de traducción de ontologías, que se definen como una secuencia de traducciones sucesivas donde el formato inicial y el final coinciden.

En las dos secciones siguientes se presentan el modelo para la construcción de servicios de traducción de ontologías y los resultados del análisis de enfoques de traducción, respectivamente.

**IV.4 MODELO DECLARATIVO MULTICAPA PARA LA CONSTRUCCIÓN DE SERVICIOS DE TRADUCCIÓN DE ONTOLOGÍAS**

**IV.4.1 Niveles de traducción de ontologías**

Como se ha indicado en las secciones previas, el modelo que se propone en esta tesis tiene en cuenta la existencia de cuatro niveles en los que se pueden tomar decisiones de traducción:
léxico, sintáctico, semántico y pragmático. Las fronteras entre todos estos niveles no son estrictas: puede haber decisiones de traducción que afecten a varios niveles.

En el nivel léxico se tratan problemas relacionados con el hecho de que distintos formatos pueden utilizar distintas gramáticas para generar sus símbolos terminales (identificadores de componentes ontológicos, documentación en lenguaje natural y valores de los atributos). En este contexto se pueden identificar tres tipos de formatos: basados en ASCII (con un conjunto de caracteres a utilizar bastante limitado), basados en UNICODE (con un conjunto de caracteres mucho más amplio), y basados en UNICODE y XML (con algunas restricciones importantes relacionadas con la sintaxis XML).

En el nivel sintáctico se tratan problemas relacionados con el hecho de que distintos formatos utilizan distintas gramáticas para generar sus símbolos terminales y no terminales. Desde esta perspectiva, se pueden distinguir cuatro tipos de lenguajes, que no son exclusivos: basados en Lisp (la gramática utilizada para definir componentes de las ontologías está basada en la sintaxis de este lenguaje de programación), basados en XML (normalmente aquí se sitúan los lenguajes de marcado), basados en texto plano ad hoc (que suelen imponer muchas restricciones), y aquéllos que proporcionan interfaces de acceso en lenguajes de programación de propósito general (normalmente Java, que es el caso de las herramientas de ontologías y de varios lenguajes). También en este nivel se tienen que resolver problemas relacionados con los tipos de datos de cada formato, que normalmente se pueden dividir en dos grupos: tipos de datos propios o tipos de datos basados en XML Schema.

En el nivel semántico se tratan problemas relacionados con el hecho de que distintos formatos están basados en distintos formalismos de representación y, por tanto, permiten definir distintos tipos de componentes. También aparecen problemas en este nivel en la traducción entre formatos basados en el mismo formalismo de representación, debido a que no siempre permiten representar los mismos tipos de componentes o de información en dichos componentes.

En el nivel pragmático se tratan problemas relacionados con la interpretación y uso de los conocimientos formalizados en los formatos de origen y destino. El resultado de una transformación de un formato a otro debe ser legible tanto por usuarios como por sistemas que son capaces de interpretar los conocimientos formalizados en un determinado formato. En este nivel se resuelven problemas como el mantenimiento de los identificadores originales utilizados para representar un componente, la ocultación de conocimientos adicionales que han tenido que ser creados para realizar la transformación, la transformación de un conjunto de expresiones en una más sencilla, etc.
IV.4.2 Especificación declarativa de decisiones de traducción

Para la implementación de las decisiones de traducción, se proponen tres lenguajes: ODELex, ODESyntax y ODESem. El último de estos lenguajes permite implementar decisiones de traducción en los niveles semántico y pragmático, dado que los tipos de transformaciones a realizar en ambos niveles son similares.

El lenguaje ODELex permite implementar decisiones de traducción en el nivel léxico. Este lenguaje está basado en el lenguaje lex [Lesk, 1975], que se utiliza para la generación de compiladores de lenguajes. Una especificación en ODELex está organizada en tres partes: la parte que contiene el código definido por el usuario, donde se pueden escribir funciones de carácter general que vayan a ser utilizadas posteriormente; la parte que contiene las declaraciones de componentes de los formatos origen y destino que se van a transformar; y la parte que contiene las transformaciones a realizar por cada uno de los componentes del formato origen.

El lenguaje ODESyntax permite implementar decisiones de traducción en el nivel sintáctico. Este lenguaje está basado en el lenguaje yacc [Johnson, 1975], que se utiliza para la generación de compiladores de lenguajes. Una especificación en ODESyntax está organizada en cinco partes: la parte que contiene el código definido por el usuario; la parte que contiene las declaraciones de componentes de los formatos origen y destino, que no tiene por qué coincidir con la especificada en el nivel léxico; la parte que contiene las declaraciones de métodos de acceso a los componentes de los formatos de origen y destino; la parte que contiene las declaraciones de métodos de creación y modificación de componentes del formato destino; y la parte que contiene las transformaciones de tipos de datos.

Finalmente, el lenguaje ODESem permite implementar decisiones de traducción en los niveles semántico y pragmático. Este lenguaje está basado en los lenguajes que permiten implementar sistemas de producción, aunque tiene algunas características propias adaptadas para resolver problemas en la traducción de ontologías. Una especificación en ODESem está organizada en tres partes: la primera parte contiene el código de usuario, como en los casos anteriores; la segunda parte declara el orden en el que se realizarán las transformaciones de los distintos componentes de la ontología origen, o el post-proceso de los componentes obtenidos en el formato destino; la tercera parte especifica las reglas de traducción de componentes y de post-proceso.
IV.4.3 Método declarativo multicapa para la creación de servicios de traducción

Como parte del modelo de creación de servicios de traducción se propone un método que consta de cuatro actividades (estudio de viabilidad, análisis de los formatos de origen y destino, diseño del servicio de traducción, e implementación del servicio de traducción).

Este método recomienda seguir un ciclo de vida iterativo para el desarrollo del servicio de traducción: en primer lugar se identifica un primer conjunto de componentes y expresiones que pueden ser fácilmente transformados de un formato a otro; a continuación se continúa con componentes y expresiones más complejas, etc. Se recomienda este ciclo de vida porque la tarea de traducir entre formatos es compleja: se deben tener en cuenta muchos aspectos y tomar muchas decisiones. En este sentido, un ciclo de vida iterativo asegura que las decisiones más complejas se toman una vez que se tiene un conocimiento más extenso de los formatos implicados.

En cada ciclo de desarrollo se propone realizar las cuatro actividades propuestas de manera secuencial con retroalimentación. Por ejemplo, si durante la fase de diseño se descubre un problema que procede del análisis entonces se debe volver a la fase de análisis para resolverlo.

A continuación se describe en detalle cada una de las actividades del método:

- **Estudio de viabilidad.** Su objetivo es decidir si el servicio de traducción se puede realizar teniendo en cuenta las restricciones impuestas por el método (por ejemplo, si se pueden dividir los problemas de traducción en capas), y algunas otras restricciones de carácter tecnológico (por ejemplo, si los formatos de origen y destino proporcionan interfaces de acceso a los conocimientos). Recibe como entrada la información correspondiente a los formatos de origen y destino y proporciona como resultado una recomendación sobre la viabilidad del sistema, para lo que se utiliza un formulario de preguntas. Esta actividad es realizada por un ingeniero de conocimientos.

- **Análisis de los formatos de origen y destino.** Su objetivo es obtener una descripción detallada de los modelos de conocimientos de ambos formatos, especificando claramente qué conocimientos se pueden transformar y qué conocimientos se pueden preservar en el proceso de transformación. En esta actividad también se decide qué enfoque se utilizará para transformar y preservar conocimientos, tal y como se describe en la siguiente sección. La entrada a esta actividad es el documento de recomendación obtenido en la actividad anterior, así como la información correspondiente a los formatos de origen y destino. Se obtiene como resultado un documento de análisis de los formatos, donde se comparan sus
modelos de conocimientos de manera detallada, se determina el ámbito del servicio de traducción, y se especifican las pruebas de sistema que dicho servicio deberá pasar tras su implementación. La actividad está compuesta de seis tareas, que son realizadas por un ingeniero de conocimientos. Para la descripción de los modelos de conocimientos se pueden utilizar técnicas informales y formales de comparación de modelos de conocimientos.

- **Diseño del servicio de traducción.** Su objetivo es obtener una descripción detallada de la arquitectura del servicio y de las transformaciones a realizar en cada uno de los niveles de traducción identificados (léxico, sintáctico, semántico y semiótico). Recibe como entrada el documento de análisis, y proporciona como salida un documento de diseño que describe los mappings que se deben realizar para transformar las ontologías del formato origen al formato destino. Esta actividad está compuesta de seis tareas, la mayor parte de las cuales se realizan en paralelo. Entre estas tareas destacan el diseño de transformaciones en el nivel léxico, en el nivel sintáctico, en el nivel semántico y en el nivel semiótico. Para cada tarea se proponen distintas técnicas de descripción de las transformaciones. Las tareas relacionadas con problemas léxicos y sintácticos pueden ser realizadas por un arquitecto software, mientras que las demás deben ser realizadas por un ingeniero de conocimientos.

- **Implementación del servicio de traducción.** Su objetivo es implementar las transformaciones identificadas en el diseño, mediante un lenguaje declarativo, de tal modo que los servicios de traducción generados sean fáciles de crear y modificar. Recibe como entrada el documento de diseño en el que se especifican las transformaciones a realizar en cada uno de los niveles de traducción identificados, y los requisitos de integración del servicio de traducción en una herramienta o plataforma, si es necesario. Como resultado se obtiene el servicio de traducción implementado en un lenguaje declarativo, convenientemente validado con las pruebas de sistema definidas durante el análisis, así como aquellos subsistemas e interfaces necesarios para la integración en una herramienta o plataforma, si es necesario. Esta actividad se divide en ocho tareas, cuatro de las cuales están relacionadas con los cuatro niveles de traducción identificados, junto con otras relacionadas con la reutilización de implementaciones similares, integración de los resultados y ejecución de tests. En las tareas de implementación se utilizan los lenguajes ODELex, ODESyntax y ODESem, descritos en la sección anterior. Estas tareas son llevadas a cabo por programadores e ingenieros de conocimiento.
IV.5 PRESERVACIÓN DE SEMÁNTICA Y PRAGMÁTICA EN TRADUCCIONES

El modelo propuesto en la sección anterior considera la posibilidad de tomar e implementar decisiones de traducción en cuatro niveles: léxico, sintáctico, semántico y pragmático. A pesar de que las decisiones que se deben tomar en todos estos niveles son igualmente importantes para conseguir construir sistemas de traducción entre dos formatos, normalmente aquéllas que se toman en los niveles semántico y pragmático son las que determinan las características principales de los servicios de traducción creados.

En esta sección se resumen los resultados de analizar cuatro alternativas para la traducción de ontologías entre lenguajes y/o herramientas, que han sido utilizadas habitualmente en el desarrollo de servicios de traducción. Para cada una de estas alternativas se ha hecho especial énfasis en cómo conservan los conocimientos relacionados con los niveles semántico y pragmático, tanto en la traducción directa entre el formato origen y destino, como en transformaciones cíclicas.

IV.5.1 Traducción indirecta, mediante la utilización de formatos comunes de intercambio.

Como ya se describió en la sección de arquitecturas de traducción, esta alternativa consiste en utilizar un formato común (pívot) para el intercambio de ontologías. Este formato puede haber sido diseñado específicamente para el intercambio de ontologías (por ejemplo, KIF) o tratarse de un formato de amplio uso para el que ya existen servicios de traducción disponibles (por ejemplo, RDF(S) y OWL).

La principal ventaja de esta alternativa es que se reduce el número de servicios de traducción que se deben crear para realizar traducciones entre \( n \) formatos (con orden de complejidad \( n \)). Asimismo, en el caso de formatos de amplio uso no se necesita ningún esfuerzo adicional para crear los servicios de traducción correspondientes, dado que normalmente se encuentran ya implementados.

Desde el punto de vista semántico, el principal problema que se afronta es la pérdida de semántica en la traducción, dado que los formatos de intercambio no son siempre tan expresivos como el modelo de conocimientos de partida.

Desde el punto de vista pragmático, en el caso de que el formato de intercambio sea suficientemente expresivo como para representar cualquier modelo de conocimientos, los mismos conocimientos se podrán expresar de muy diversas maneras. Esto hace difícil tener en
cuenta todas las posibilidades a la hora de importar desde el formato de intercambio, dificultando la legibilidad de la ontología y la tarea de importación desde el mismo.

**IV.5.2 Traducción directa mediante transformaciones entre los formatos de origen y destino, sin utilizar componentes adicionales**

Esta alternativa propone transformar al formato destino todo aquello que se pueda traducir del formato origen, tomando todas las decisiones de traducción que sean necesarias. Este enfoque es el que se ha utilizado en la mayoría de los servicios de traducción disponibles en la actualidad.

Desde el punto de vista semántico, esta alternativa presenta el problema de la *pérdida de conocimientos* en la traducción en la mayor parte de los casos. Esto se debe a que normalmente los formatos origen y destino no permiten representar los mismos conocimientos, y por tanto hay partes de la ontología que no se pueden transformar.

Sin embargo, desde el punto de vista pragmático no aparecen muchos inconvenientes, dado que la *ontología normalmente es legible en el formato destino* y sólo en algunos casos aislados el resultado de las transformaciones es difícil de comprender.

**IV.5.3 Traducción directa mediante la instanciación de la ontología de representación de conocimientos del formato original en el formato destino**

En esta alternativa el proceso de traducción se realiza en dos pasos: en primer lugar, se crea en el formato destino la ontología de representación de conocimientos del formato origen, como si se tratase de una ontología de dominio; en segundo lugar, se transforma la ontología del formato origen al formato destino mediante la instanciación de la ontología creada previamente.

La ventaja principal de esta alternativa se encuentra en el nivel semántico: *los conocimientos se preservan en la transformación*. Asimismo, *las decisiones de traducción son sencillas*, dado que no se está transformando el modelo de conocimientos a otro distinto, sino al mismo expresado como instancias de una ontología de dominio.

Sin embargo, desde el punto de vista pragmático no se obtienen buenos resultados, dado que la ontología en el formato destino no está expresada en términos del modelo de conocimientos destino, sino del origen. En este caso, *los usuarios de la ontología* (ya sean humanos o sistemas) *no serán capaces de comprender y/o utilizar con facilidad la misma*. 
IV.5.4 Traducción directa mediante transformaciones entre los formatos de origen y destino, utilizando componentes adicionales para preservar la semántica en la traducción

Esta alternativa goza de las ventajas de las dos anteriores: por un lado, después de la transformación la ontología es legible en el formato destino, dado que se utilizan los componentes estándar del formato destino; por otro lado, se mantienen los conocimientos que se podrían perder debido a las diferencias entre los modelos de conocimientos de los formatos de origen y destino, normalmente instanciando la ontología de representación de conocimientos del formato de origen en el destino.

El principal inconveniente de esta solución es que las decisiones tomadas para la transformación entre formatos son en general más complejas que en el resto de alternativas (deben proponerse transformaciones directas y otras transformaciones para preservar conocimientos). Asimismo, esta solución no es válida en general para su aplicación a la generación de ontologías en lenguajes comunes de intercambio, dado que existen partes de la ontología que se transforman de acuerdo con el formato de origen.

Desde la perspectiva pragmática, ya se ha comentado que la ontología sigue siendo legible, a pesar de que aparezcan elementos correspondientes al formato de origen. Esta dimensión se ve ampliamente mejorada en el caso de que el formato destino permita ocultar esta información a los usuarios (por ejemplo, esto es posible en la herramienta Protégé-2000 mediante la ocultación de clases y metaclasses).

IV.6 EXPERIMENTACIÓN

Los resultados de esta tesis han sido ampliamente experimentados en el desarrollo de servicios de traducción (importación y exportación) para la plataforma de desarrollo de ontologías WebODE.

En primer lugar, el modelo declarativo multicapa propuesto se ha obtenido de una manera experimental a partir de la creación de los servicios de exportación de WebODE a los lenguajes XML, XCARIN, OIL, DAML+OIL, FLogic, Prolog, Java y Jess, y de los servicios de importación a WebODE desde los lenguajes XML, XCARIN y DAML+OIL.

Una vez depurado el método con los experimentos anteriores, éste ha sido utilizado con éxito en la construcción de los servicios de importación y exportación de WebODE desde/a los lenguajes de ontologías RDF(S) y OWL, y desde/a la herramienta de ontologías Protégé-2000.
Asimismo, en relación con el análisis de las distintas alternativas de traducción mostradas en la sección anterior, todas ellas han sido probadas en alguno de los servicios de traducción creados, del siguiente modo:

- La primera alternativa (transformación indirecta, utilizando formatos de intercambio) ha sido probada en el contexto del Special Interest Group on Enterprise-Standard Ontology Environments de la red temática europea OntoWeb, donde se ha demostrado que la utilización de RDF(S) y OWL como lenguajes de intercambio entre las plataformas WebODE y Protégé-2000 no permite conservar conocimientos, además de producir problemas en los niveles léxico, sintáctico y semiótico.

- La segunda alternativa (transformación directa sin componentes adicionales) ha sido probada con éxito en los exportadores e importadores de WebODE a XCARIN, RDF(S), DAML+OIL y OWL, y en el exportador a FLogic.

- La tercera alternativa (transformación directa mediante instanciación de una ontología de representación en el formato destino) ha sido probada con éxito en los exportadores e importadores a XML, y en los exportadores a Prolog, Java y Jess. En todos estos casos, se ha representado el modelo de conocimientos de WebODE en estos lenguajes genéricos, sin pérdidas de conocimientos.

- La cuarta alternativa (transformación directa con componentes adicionales) ha sido utilizada en el exportador e importador a la herramienta de ontologías Protégé-2000.

Con el objetivo de poder comparar todos estos enfoques, se han creado asimismo servicios de traducción de ontologías entre WebODE y Protégé-2000 utilizando las cuatro alternativas (aunque la que actualmente se utiliza es la cuarta). Los mejores resultados desde las perspectivas semántica y pragmática son los proporcionados por la cuarta alternativa. Esto puede justificar el esfuerzo adicional necesario para la generación del traductor ad hoc entre ambas plataformas.

**IV.7 CONCLUSIONES**

En esta tesis se han abordado algunos de los problemas existentes en el estado del arte actual sobre traducción de ontologías. Las soluciones planteadas no pretenden resolver el problema de la traducción, sino ayudar en la creación de sistemas de traducción, facilitando la tarea de construcción y mantenimiento de los mismos.
En primer lugar, se ha tratado la falta de consenso en la clasificación de niveles de traducción en los que se deben abordar problemas, proponiendo un conjunto de niveles basado en la teoría de signos.

En segundo lugar, se ha propuesto un modelo integrado para la construcción de servicios de traducción de ontologías, basado en los niveles de traducción anteriores, y con el cual las decisiones de traducción se implementan de manera declarativa, lo que hace más fácil la construcción y mantenimiento de servicios de traducción. Asimismo, este modelo propone un ciclo de vida iterativo en la construcción de servicios de traducción y se identifican las técnicas que se pueden utilizar en cada una de las tareas.

Finalmente, se ha realizado un análisis detallado de distintas alternativas de traducción que se pueden encontrar en el estado del arte actual, poniendo especial énfasis en cómo conservan la semántica y pragmática de las ontologías transformadas.

**IV.8 FUTURAS LÍNEAS**

A pesar de las contribuciones al estado del arte realizadas en esta tesis, existen algunos problemas de investigación que no se han tratado en la misma y que pueden resultar interesantes para ser abordados en el futuro:

En primer lugar, el modelo propuesto está orientado a la transformación de ontologías de dominio entre dos lenguajes y/o herramientas de ontologías. Este modelo no está diseñado específicamente para otros formatos que permitan representar otros tipos de conocimientos, como servicios Web semánticos, ontologías de alto nivel, ontologías de representación de conocimientos, conocimientos sobre planificación, etc.

En segundo lugar, el modelo propuesto no considera la optimización del proceso de traducción del sistema construido con él, tanto en temas de espacio como de tiempo necesitado durante la traducción.

Otro problema abierto está relacionado con la transformación de partes de la ontología, en lugar de toda la ontología completamente. En este sentido, se podría considerar en el futuro la integración de los sistemas de traducción generados con sistemas de gestión de la evolución de ontologías, para realizar la traducción sólo de las partes de la ontología que han evolucionado desde una traducción previa.

Los lenguajes declarativos propuestos en esta tesis (ODELex, ODESyntax y ODESem) no son suficientemente expresivos como para representar cualquier tipo de transformación a realizar en
estos niveles. En el caso de transformaciones complejas, se requiere el uso de funciones de transformación implementadas de manera ad hoc en lenguajes de programación de propósito general.

El análisis de la conservación de la semántica y la pragmática en la traducción de ontologías sólo trata de los aspectos de representación de las ontologías, y no de los aspectos relacionados con el razonamiento.
ABox, 12, 14
Ad hoc text formats, 52
ADD, 76, 84
Algernon, 244
ALL, 74
Amaya, 220
Analysis of source and target formats, 99
Anonymous class, 57, 61
ASCII-based formats, 49
Attributes:
   class attributes, 21, 24, 204, 216, 224
   concept attributes, 12, 224
   instance attributes, 21, 24, 215, 235
Automatic classification, 62, 229, 240
BNF grammar, 121
bottom, 61
Cardinality restriction, 56
CARIN, 29

Chimaera, 28
CKRL, 139
Classes, 12, 203, 214, 222, 231
class taxonomy, 12
Classic, 12
Common interchange format, 35
CommonKADS, 96; worksheet, 96
Complete, 11
Complete class definition, 55, 61
Concepts, 13, 203, 214, 222, 231
   concept taxonomies, 21, 24, 206, 216,
   225, 236, 254
   defined concept, 14
   primitive concept, 14
Consequence preservation, 136
Constraint checking, 22, 212, 229, 240
ContentWeb, 42
CREATE, 76, 83
Cyc:
  CycL language, 15
Cyclic transformation, 138
DAML:
  DAML ontology library, 249
DAML+OIL, 17, 29, 30, 169, 220, 230, 251, 259, 260
DAML-ONT, 220
Datatypes:
  internal datatypes, 53
  XML Schema datatypes, 53
Description logic, 2
Design of the ontology translation system, 111
Direct ontology translation, 150
Disjoint decomposition, 56
DLML, 37
DQL, 229
DUET, 229
e-cl@ss, 10
EON2002, 172
EON2003, 145
Esperonto, 42
Euler, 240
EXEC, 84
Exhaustive decomposition, 56
EXPRESS, 139
Expressive power, 20
Expressiveness, 54
Facets, 24
FaCT, 29, 228, 240, 244
Family of ontology languages, 35
Feasibility:
  business feasibility, 89
  feasibility study, 89
  project feasibility, 89
  technical feasibility, 89
First class citizen, 67
First order logic, 2, 11, 244
FLogic, 16, 29, 30, 169, 244, 251, 260
FLORA, 249
forEach ... IN ..., 84
Formal axioms, 12, 14, 22, 211, 217, 228, 239
Frame Ontology, 51, 202, 205
Frames KR paradigm, 2, 11, 12, 244
Functions, 12, 21, 25, 210, 228, 239
GET, 68
GETALLCOMPONENTS, 84
GETCOMPONENT, 84
ICS-FORTH RDFSuite, 220
Implementation of the ontology translation system, 123
Indirect ontology translation, 139
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDIVIDUAL</td>
<td>74</td>
</tr>
<tr>
<td>Individuals</td>
<td>13, 14, 211</td>
</tr>
<tr>
<td>INFORMATION</td>
<td>74</td>
</tr>
<tr>
<td>Inheritance</td>
<td>212</td>
</tr>
<tr>
<td>INIT</td>
<td>68</td>
</tr>
<tr>
<td>In-processing transformations</td>
<td>80</td>
</tr>
<tr>
<td>Instances</td>
<td>12, 22, 25, 211, 217, 228, 239</td>
</tr>
<tr>
<td>Intermediate representations</td>
<td>252</td>
</tr>
<tr>
<td>Interoperability problem</td>
<td>3</td>
</tr>
<tr>
<td>Interpretation preservation</td>
<td>136</td>
</tr>
<tr>
<td>Iterative life cycle</td>
<td>87</td>
</tr>
<tr>
<td>JCup</td>
<td>72</td>
</tr>
<tr>
<td>Jena toolkit</td>
<td>220</td>
</tr>
<tr>
<td>Jess</td>
<td>29, 229, 244</td>
</tr>
<tr>
<td>JLex</td>
<td>66</td>
</tr>
<tr>
<td>JTP</td>
<td>212, 229</td>
</tr>
<tr>
<td>KACTUS project</td>
<td>9</td>
</tr>
<tr>
<td>KAON</td>
<td>30</td>
</tr>
<tr>
<td>KIF</td>
<td>16, 35, 139, 140, 201, 245</td>
</tr>
<tr>
<td>KL-ONE</td>
<td>12</td>
</tr>
<tr>
<td>Knowledge level</td>
<td>33</td>
</tr>
<tr>
<td>Knowledge Representation:</td>
<td></td>
</tr>
<tr>
<td>KR languages</td>
<td>15</td>
</tr>
<tr>
<td>KR ontologies</td>
<td>102</td>
</tr>
<tr>
<td>KR paradigms</td>
<td>19, 53</td>
</tr>
<tr>
<td>KR paradigms</td>
<td>15</td>
</tr>
<tr>
<td>Knowledge Sharing Effort</td>
<td>1, 201</td>
</tr>
<tr>
<td>Kris</td>
<td>12</td>
</tr>
<tr>
<td>KRSL</td>
<td>139</td>
</tr>
<tr>
<td>Krypton</td>
<td>12</td>
</tr>
<tr>
<td>lex</td>
<td>65, 66</td>
</tr>
<tr>
<td>Lexical layer</td>
<td>48, 65, 121, 130, 136</td>
</tr>
<tr>
<td>LHS (Left Hand Side)</td>
<td>83</td>
</tr>
<tr>
<td>Lisp-based formats</td>
<td>52</td>
</tr>
<tr>
<td>LOOM</td>
<td>12, 16; LOOM ontologies, 28</td>
</tr>
<tr>
<td>Mapping translation approach</td>
<td>35</td>
</tr>
<tr>
<td>Metaclasses</td>
<td>12, 55, 160, 245</td>
</tr>
<tr>
<td>METHONTOLOGY</td>
<td>40, 250</td>
</tr>
<tr>
<td>METRICA</td>
<td>90</td>
</tr>
<tr>
<td>Minimal encoding bias</td>
<td>33</td>
</tr>
<tr>
<td>MKBEEM</td>
<td>42</td>
</tr>
<tr>
<td>Model theory</td>
<td>20, 104</td>
</tr>
<tr>
<td>Model-theoretic semantics</td>
<td>104</td>
</tr>
<tr>
<td>Multiple inheritance</td>
<td>22, 229, 240</td>
</tr>
<tr>
<td>Neutral Representation project</td>
<td>142</td>
</tr>
<tr>
<td>Notational conciseness</td>
<td>42</td>
</tr>
<tr>
<td>Notational efficacy</td>
<td>42</td>
</tr>
<tr>
<td>OCML</td>
<td>16, 28</td>
</tr>
<tr>
<td>ODEAnnotate</td>
<td>250</td>
</tr>
<tr>
<td>ODEClean</td>
<td>165, 252, 256, 258</td>
</tr>
<tr>
<td>ODEDialect</td>
<td>41</td>
</tr>
<tr>
<td>ODEKM</td>
<td>250</td>
</tr>
<tr>
<td>ODELex</td>
<td>41, 65, 130</td>
</tr>
<tr>
<td>ODEMerge</td>
<td>252, 256, 257</td>
</tr>
</tbody>
</table>
A declarative approach to ontology translation with knowledge preservation

ODESem, 41, 80, 128, 129
ODESeW, 250
ODESWS, 250
ODESyntax, 41, 70, 129
OIL, 17, 29, 169, 260
OilEd, 29
OILEd, 229
OKBC, 16, 29, 244, 255
OKBC Ontology, 51, 202
OKBC-based Prolog, 256
OntoDesigner, 253
OntoEdit, 30, 229
On-To-Knowledge, 40
Ontolingua, 16, 139, 201
Ontolingua Server, 28, 140, 202
Ontolingua Server ontology library, 249
Ontologies:
  heavyweight ontologies, 10, 11, 12, 24
  lightweight ontologies, 10, 24
  ontology, 1, 7
  Ontology, 7
Ontology development tools, 27
Ontology evaluation, 259; Ontology evaluation tools, 27
Ontology evolution, 30
Ontology languages, 2, 15
  ontology markup languages, 17
  web-based ontology languages, 17
  Ontology learning tools, 27
  Ontology management API, 43, 52, 72
  Ontology mapping, 30
  Ontology merge and alignment tools, 27
  Ontology querying tools and inference engines, 27
  Ontology tools, 2
  Ontology translation problem, 2, 33
  Ontology-based annotation tools, 27
  OntoMorph, 37, 41
  OntoSaurus, 28, 140
  OntoWeb SIG on Enterprise-Standard Ontology Environments, 144
  Order preservation, 136
OVERLAP, 69
OWL, 19, 29, 35, 102, 139, 148, 230, 251, 259, 260
  annotation properties, 150, 161
  OWL DL, 102, 149, 170
  OWL Full, 102, 149
  OWL KR ontology, 102
  OWL Lite, 102, 149
PAL, 29, 53, 244, 245, 249
  Partial class definition, 55, 61
  Partition, 56, 206
  Peer-to-peer transformation, 35
PIF, 139
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivot translation approach</td>
<td>35</td>
</tr>
<tr>
<td>Pivot translation architecture</td>
<td>139</td>
</tr>
<tr>
<td>Post-processing transformations</td>
<td>79</td>
</tr>
<tr>
<td>Pragmatic layer</td>
<td>58, 115, 128, 136</td>
</tr>
<tr>
<td>Pragmatics</td>
<td>19</td>
</tr>
<tr>
<td>Pre-processing transformations</td>
<td>79</td>
</tr>
<tr>
<td>Primitive classes</td>
<td>55</td>
</tr>
<tr>
<td>Prolog</td>
<td>29, 244, 255, 257, 260</td>
</tr>
<tr>
<td>PROMPT</td>
<td>29</td>
</tr>
<tr>
<td>Properties</td>
<td>21, 215</td>
</tr>
<tr>
<td>Property restriction</td>
<td>56</td>
</tr>
<tr>
<td>Protégé-2000</td>
<td>29, 171, 243</td>
</tr>
<tr>
<td>QName</td>
<td>49</td>
</tr>
<tr>
<td>RACER</td>
<td>29, 228, 240</td>
</tr>
<tr>
<td>RDF</td>
<td>17, 139, 144, 212</td>
</tr>
<tr>
<td>RDF data model</td>
<td>212</td>
</tr>
<tr>
<td>RDF Schema</td>
<td>17, 139, 144, 212, 244</td>
</tr>
<tr>
<td>RDF(S)</td>
<td>17, 29, 30, 35, 144, 170, 213, 221, 230, 250, 251, 259, 260</td>
</tr>
<tr>
<td>RDFQL</td>
<td>219</td>
</tr>
<tr>
<td>RDQL</td>
<td>219</td>
</tr>
<tr>
<td>Reasoning</td>
<td>14, 54</td>
</tr>
<tr>
<td>Reification</td>
<td>218</td>
</tr>
<tr>
<td>Relations</td>
<td>12, 21</td>
</tr>
<tr>
<td>ad-hoc relations</td>
<td>253</td>
</tr>
<tr>
<td>binary relations</td>
<td>25, 208, 216, 226, 237</td>
</tr>
<tr>
<td>REMOVE</td>
<td>76, 84</td>
</tr>
<tr>
<td>REMOVEALL</td>
<td>76, 84</td>
</tr>
<tr>
<td>REMOVEINDIVIDUAL</td>
<td>76</td>
</tr>
<tr>
<td>REPEATED</td>
<td>68</td>
</tr>
<tr>
<td>RHS (Right Hand Side)</td>
<td>83</td>
</tr>
<tr>
<td>RIL</td>
<td>219</td>
</tr>
<tr>
<td>Roles</td>
<td>13, 14, 21</td>
</tr>
<tr>
<td>RosettaNet</td>
<td>10</td>
</tr>
<tr>
<td>RQL</td>
<td>219</td>
</tr>
<tr>
<td>Rules</td>
<td>26</td>
</tr>
<tr>
<td>Second class citizen</td>
<td>56</td>
</tr>
<tr>
<td>Semantic layer</td>
<td>53, 118, 129, 136</td>
</tr>
<tr>
<td>Semantic network</td>
<td>2</td>
</tr>
<tr>
<td>Semantics</td>
<td>19</td>
</tr>
<tr>
<td>axiomatic semantics</td>
<td>20</td>
</tr>
<tr>
<td>Semiotics</td>
<td>19</td>
</tr>
<tr>
<td>SENSUS</td>
<td>9</td>
</tr>
<tr>
<td>SHIQ</td>
<td>13, 222</td>
</tr>
<tr>
<td>SHOE</td>
<td>17</td>
</tr>
<tr>
<td>SilRI</td>
<td>219</td>
</tr>
<tr>
<td>Slots</td>
<td>12, 21</td>
</tr>
<tr>
<td>Sound</td>
<td>11</td>
</tr>
<tr>
<td>SUMM</td>
<td>139</td>
</tr>
<tr>
<td>Synset preservation</td>
<td>136</td>
</tr>
<tr>
<td>Syntax</td>
<td>19</td>
</tr>
<tr>
<td>Syntax layer</td>
<td>51, 70, 119, 129, 136</td>
</tr>
<tr>
<td>TABLE</td>
<td>68</td>
</tr>
<tr>
<td>TBox</td>
<td>12, 14</td>
</tr>
</tbody>
</table>
A declarative approach to ontology translation with knowledge preservation

| Theory of signs, 19                  | WebOnto, 28, 140                   |
| Transmorpher, 37, 41                | WebPicker, 252                     |
| TRIPLE, 228, 240                    | WordNet, 244                       |
| UML, 250                            | XCARIN, 169, 251, 260              |
| UMLS, 244                           | XMI, 250                           |
| UNICODE&XML-based formats, 49       | XML, 29, 30, 169, 244, 250, 251    |
| UNICODE-based formats, 49           | XML Schema, 244, 250               |
| UNSPSC, 10                          | XML-based formats, 52              |
| WAB, 29, 253, 258                   | XOL, 17                            |
| WebODE, 29, 229, 250                | yacc, 70, 72                       |
| Documentation service, 256          | Yahoo! Directory, 10               |
| WebOnt, 230                         |                                  |