UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES ÉCONOMIQUES

Towards the Democratization of Ontological Modeling through a New Pervasive Means of Representation

PhD thesis submitted to the Faculty of Economics and Business

Institute of Management of Information Systems

University of Neuchâtel

For the PhD degree in Computer Science

by

Selena Baset

Approved by the dissertation committee:

Prof Kilian Stoffel, University of Neuchâtel, thesis director

Prof. Adrian Holzer, University of Neuchâtel, head of jury

Prof. Paul Cotofrei, University of Neuchâtel, expert

Prof. Philippe Cudré-Mauroux, University of Fribourg, external expert

Prof. Thomas Studer, University of Bern, external expert

Defended on 07.06.2019

Faculté des sciences économiques
Avenue du 1er-Mars 26
CH-2000 Neuchâtel
www.unine.ch/seco

LETTRES ET SCIENCES HUMAINES
SCIENCES
DROIT
SCIENCES ECONOMIQUES

UNIVERSITÉ DE NEUCHÂTEL

IMPRIMATUR POUR LA THÈSE

Towards the Democratization of Ontological Modeling through a New Pervasive Means of Representation

Selena BASET

UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES ÉCONOMIQUES

La Faculté des sciences économiques,
sur le rapport des membres du jury

Prof. Kilian Stoffel (directeur de thèse, Université de Neuchâtel)
Prof. Adrian Holzer (président du jury, Université de Neuchâtel)
Prof. Paul Cotofrei (Maître d'enseignement et de recherche, Université de Neuchâtel)
Prof. Philippe Cudré-Mauroux (Université de Fribourg)
Prof. Thomas Studer (Université de Berne)

Autorise l'impression de la présente thèse.

Neuchâtel, le 23 septembre 2019

Annik Dubied

La doyenne
Annik Dubied

Abstract

Despite their proven utility in various areas of knowledge engineering, ontologies are falling short of reaching an equal position as formal domain models in the landscape of enterprise software development. With the ultimate goal of making ontologies more accessible to a wider audience of software development stakeholders, this thesis investigates the reasons that are still standing between ontology integration research efforts as depicted in literature and a successful democratization of ontologies. As our preliminary analysis results suggest, the shy adoption of ontologies in software engineering communities is partially due to the opposing semantics and the different underlying assumptions used in ontologies compared to the other more conventional software modeling paradigms. The technical stacks behind ontological applications and conventional enterprise software also vary greatly; from languages and editors to infrastructure support.

The above reasons constitute a syntactic, semantic and paradigmatic twist the developer has to go through each time trying to use ontologies. In order to minimize this manifold twist, this thesis proposes a loss-less translation function that would facilitate the task of integrating ontologies into conventional code repositories by expressing them directly in the same general-purpose programming language already in use. Such a translation of ontologies from their passive external form into an active executable form does not only enhance the portability of ontologies but also contributes to a substantial reduction in both the steep learning curve and the overhead of the technical stacks usually associated with ontologies. This proposition is further crystallized by the implementation of the translation function using C# as the target language of the translation. Our choice for C# is motivated by a number of factors that we discuss in the corresponding chapter. The implementation also takes into account potential optimizations of the translation process. A substantial reduction in the executable size is achieved by exploiting the dot notation of the target programming language to avoid the redundant representation style inherent in the source XML serialization. Additionally, a compressing technique that detects redundant recurrent anonymous patterns in the source ontology is proposed to reduce the size of the resulting executable by generating a unique corresponding class for each set of recurrent patterns. This can be particularly useful in the bio-medical domain

where the sheer size of ontologies is itself an issue. Furthermore, the proposed technique is not unique to executable ontologies but can also be generalized to other syntactic formats of OWL ontologies.

The thesis further illustrate two potential applications of translating ontologies into an executable representation. Both applications are geared towards addressing the difficulties of fully exploiting ontologies in conventional development environments. We first propose bridging the imperative control gap usually present in the passive form of ontologies by providing executable ontologies with procedural extensions in the form of object-oriented methods. We then move to present a more elaborated prototype of a simplified stack for a knowledge base system that can provide light-weight reasoning services by exploiting the support of the programming environment. An experiment is conducted to assess the utility of the prototype in answering certain type of semantic queries without invoking a DL reasoner. The empirical results obtained provide a supporting evidence of the feasibility of the proposed prototype.

Keywords: Semantics, Knowledge Representation and Reasoning, Executable Ontologies, Description Logics, OWL, Programming Languages

Résumé

Malgré leur utilité prouvée dans de nombreux domaines de l'ingénierie des connaissances, les ontologies ne parviennent pas à égaler cette position comme des modèles de domaine dans le secteur de développement de logiciels d'entreprise. Tout en ayant pour but de rendre les ontologies accessibles à un plus grand nombre d'intervenants dans le cadre du développement de logiciels, cette thèse cherche à étudier les facteurs qui se dressent entre les efforts de recherche sur l'intégration des ontologies comme elle est, d'une part, décrite par la littérature, et, d'autre part, une démocratisation menée à terme. Tout comme notre analyse préliminaire le suggère, la réticence qui caractérise l'adoption des ontologies au sein de la communauté des ingénieurs est en partie due à une sémantique opposée, ainsi qu'aux différentes hypothèses sous-jacentes utilisées dans les ontologies, par rapport à d'autres normes de modélisation d'entreprise plus conventionnels. Les piles techniques derrière l'application des ontologies et les logiciels d'entreprise conventionnels varient considérablement ; des langues aux éditeurs en passant par l'appui aux infrastructures.

Les raisons mentionnées ci-dessus prennent une nouvelle tournure syntactique, sémantique et paradigmatique que le développeur doit prendre en compte à chaque fois qu'il utilise les ontologies. Afin de minimaliser ces diverses tournures, la première étape vers la réalisation de cette thèse a été la proposition d'une fonction de traduction sans perte qui pourrait faciliter la tâche d'intégration des ontologies vers les référentiels de code plus conventionnels en les exprimant directement dans le même langage de programmation en usage. La deuxième étape franchie fut l'implémentation, de manière concrète, de la fonction de traduction en considérant C# comme langage cible. Cette implémentation prend également en compte de potentielles optimisations du processus de conversion. Le choix de C# est motivé par un certain nombre de facteurs discutés dans le chapitre correspondant. Cette implémentation prend également en compte de potentielles optimisations du processus de conversion. On obtient une réduction significative de la taille de l'exécutable en exploitant la notation du langage de programmation qui permit d'éviter la redondance inhérent à la source de la sérialisation XML. De plus, une technique de compression qui détecte les modèles anonymes redondants récurrents dans l'ontologie source est proposée afin de ré-

duire la taille de l'exécutable qui en résulte en générant une classe correspondante unique pour chaque ensemble de concepts récurrents. Ceci peut être particulièrement utile dans le domaine biomédical, où l'étendue des ontologies constitue en elle-même un problème. Par ailleurs, la technique proposée n'est pas restreinte aux ontologies exécutables, mais peut aussi être généralisée à d'autres formats syntactiques d'ontologies OWL.

Cette thèse illustre également deux applications potentielles de conversions d'ontologies en représentation exécutable. Ces deux applications visent à aborder les difficultés relatives à l'exploitation efficace des ontologies dans des environnements de développement conventionnels. Dans un premier temps, nous proposons un procédé de pontage de l'interstice de commande impératif généralement présent dans les formes d'ontologie passives en fournissant des ontologies exécutables avec des extensions procédurales dans la forme de procédés orientés objets. Dans un deuxième temps, nous nous axons sur la présentation d'un prototype plus élaboré de pile simplifiée pour un système de base de connaissances pouvant fournir des services de raisonnement légers en exploitant le support de l'environnement de programmation. Nous conduisons une expérience afin d'évaluer l'utilité du prototype en répondant à certains types de requêtes sémantiques sans avoir recours à un raisonneur de Description Logique (DL). Enfin, les résultats empiriques obtenus fournissent des éléments de preuve démontrant la faisabilité du prototype proposé.

Mots clés : Semantiques, Représentation des connaissances et raisonnement, Desciption Logics, OWL, Ontologies exécutables, Langages de programmation

Acknowledgements

It has been a long way for me to get to this moment but I feel very fortunate for being surrounded with great colleagues, friends and family who have all contributed in one way or another to the realization of this thesis.

My sincerest gratitude to Professor Kilian Stoffel, my thesis director. I am very grateful for having had the opportunity to do this PhD. I truly appreciate the trust you put in me early on. Thank you for all the enlightening discussions we had.

Many thanks to the members of my thesis committee: Professor Paul Cotofrei, Professor Philippe Cudré-Mauroux, Professor Adrian Holzer and Professor Thomas Studer for setting aside time to review my work and for their insightful comments.

A special thank you to Eugenia for being such a great organizer and for all the administrative support she provided over the last few years.

On this way, I met many people, several of whom became dear friends. My sincere thanks to my colleagues and fellows: Simin, Martina, Pierluigi, Eliane, Alessio, Arielle, Aditya and Kristoffer for having been such a great company, and for all the interesting and insightful discussions we have had over coffee or during our lunch breaks.

Then of course, a warm-hearted thank you goes to my family overseas. Thanks mom and dad for your lasting encouragement regardless of the choices I make. Ramie, Yazan and Omar, I cannot be more proud to be your sister.

Finally, to you Munib, thank you my dear for walking the whole way with me, for your love and continuous support.

Dedication

To my beloved Lara

For bringing so much meaning to my life.

"Begin at the beginning", the King said gravely, "and go on till you come to the end: then stop."

Lewis Carroll, Alice in Wonderland

List of Acronyms

AGI	Artificial General Intelligence.
AI	Artificial Intelligence.
AST	Abstract Syntax Tree.
CLOS	Common Lisp Object System.
CLR	Common Language Runtime.
CWA	Closed World Assumption.
DFS	Depth First Search.
DL	Description Logics.
EDM	Enterprise Data Model.
FOL	First Order Logic.
GO	Gene Ontology.
ICD10	The International Classification of Diseases Version 10.
JIT	Just In Time.
KR	Knowledge Representation.

KRR	Knowledge Representation and Reasoning.
LINQ	Language Integrated Query.
ODM	Ontology Definition Metamodel.
ODSD	Ontology Driven Software Development.
OMG	Object Modeling Group.
OntoJIT	Ontologies Just In Time.
OOP	Object Oriented Programming.
OWA	Open World Assumption.
OWL	The Web Ontology Language.
RDF	Resource Description Framework.
RDFS	Resource Description Framework Schema.
SNOMED CT	The Systematized Nomenclature of Medicine – Clinical Terms.
SPARQL	Sparql Protocol and Query Language.
UML	Unified Modeling Language.
UNA	Unique Name Assumption.
URI	Uniform Resource Indicator.
W3C	The World Wide Web Consortium.
XML	Extensible Markup Language.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"I am reminded of the story of the English linguist who studied many different languages and finally concluded that English was the best of all languages because in English, the words come in the same order that you think them."

anecdote, Eric Lippert

1.1 Motivation and scope

7097 is the number of natural languages spoken by humans all over the world [1]. A comparable number of 8945 languages is "spoken" by machines [2]. But while more and more natural languages are endangered, computer languages are certainly on the rise [3]. The

1. According to an article published in the Ethnologue website https://www.ethnologue.com/guides/how-many-languages accessed on 20^{th} January 2019.
2. According to HOPL: Online Historical Encyclopaedia of Programming Languages. Website accessed on 20^{th} January 2019.
3. According to same two sources.

modest history of machines existence compared to the antiquity of natural languages simply means that computer languages are increasing at a rate that is an order of magnitude faster than any growth rate that human languages have ever witnessed. What is even more puzzling is that machines, at least up to our date, are supposed to be agnostic to ethnics or geopolitical differences, factors that can foster the growth and diversity of human languages. For computer languages, apparently, these factors are linked to the proliferation of the different paradigmatic schools behind these languages[1]. At this point of time we have the luxury of adopting philosophically varying paradigms that can have a say on our choice of programming languages. We also have all the technology it takes for the creation of new programming languages to become a commodity. Seen from that perspective, all sorts of generic as well as domain specific languages will keep popping-up. Nonetheless, with the advances of natural language understanding and the maturity of certain multi-paradigm programming techniques, it might very well be that a few of the dominant programming languages would be able to absorb the underlying variations in the way we communicate with computers. Ultimately, the question of whether we are heading towards a universal "Lingua-franca" where humans and computers can truly communicate is quite an interesting one. Trying to get the answer to such a big question in the time frame of a PhD thesis is rather a mere naive optimism. Yet, and on a more modest scale, the unnecessarily diverging directions which are taken by some computer languages is a concern that motivated the work presented in this thesis.

Targeting a fine-grained area of the computer languages eco-system, this thesis is situated on the cross roads between languages used in the field of knowledge representation and those used in conventional software development. With an admittedly "convergenist" mindset, this dissertation will be looking into means of enforcing the interoperability between these languages and, where necessary, workarounds for overcoming their inherent differences both at a the conceptual and implementation levels. The ultimate goal here being the democratization of techniques used in the knowledge representation field into a wider audience of software development stakeholders.

1.2 Research questions and contributions

By further narrowing down the scope defined earlier into two flagship languages: The Web Ontology Language (OWL) as an annotation language used in Knowledge Representation (KR) systems and C# as a multi-paradigm language used in conventional software development, we can now introduce the research questions addressed in our work as follows:

— Given the inherent syntactic, semantic and paradigmatic differences of OWL as a knowledge representation language, to which extent can a mainstream multi-paradigm programming language be used to represent ontologies originally written in OWL?

— How representing ontologies in a mainstream programming language would help in bringing ontologies closer to the communities of conventional software developers? What are some of the potential applications of ontologies in their new representation?

Given the methodological nature of the raised questions, it was difficult, and not relevant, to test the corresponding hypotheses against some pre-established quantitative criterion. For that reason, a proof by construction approach was adopted to validate the proposed answers to the questions above.

Beside systematically surveying the literature to provide a reference framework on existing ontology integration approaches, our work contributes to the the-state-of-the-art by introducing the definition and the implementation of a module to translate OWL ontologies into an executable form. To put the translation results into test, a prototype for a light-weight knowledge base system is also introduced. By exploiting the support of the programming environment, such a light-weight system can drastically reduce the technical stack complexity and the steep learning curve usually associated with OWL ontologies.

1.3 Thesis Structure

The rest of this thesis is organized as follows:

— **Chapter 2** starts by setting the background for the subsequent chapters. It provides a preliminary overview on Knowledge Representation (KR) as the sub-field of AI research where ontologies emerged. With a gradual narrowing scope, it passes through early KR systems including frames and semantic networks to logic-based formalisms, fragments of first-order logics and the Description Logics (DL) fragment in particular. The chapter then moves on to covering the most notable languages and standards used to build and share ontologies.

— **Chapter 3** continues on covering the background materials for the areas related to programming languages and paradigms. While by no means an exhaustive review, this chapter addresses some of the important questions to be raised when given the problem of using a general-purpose programming language to represent ontologies. The motivation behind choosing C# to answer most of these questions is included in this chapter.

— **Chapter 4** is dedicated for establishing the state-of-the-art. With the modest adoption of ontologies in software development communities as a research problem, this chapter takes the form of a systematic literature survey of existing integration approaches between ontologies and software modeling languages and practices. The analysis of the surveyed resources resulted in a classification framework that we also present in this chapter. Finally the chapter concludes with some of the insight we gained about the further implicit reasons contributing to the overall reluctance of software engineering communities to embrace ontologies in the realm of conventional software development.

— **Chapter 5** makes use of the knowledge collected from the previous chapter to present a novel approach for integrating ontologies into the native programming environment of software developers. The proposed approach is based on translating

ontologies into a general-purpose programming language. An abstraction of the translation problem is first established in this chapter.

— **Chapter 6** continues on the work started in chapter 5 but with a more concrete description of the implementation of the translation function. The chapter starts by explaining the motivation behind the chosen target programming language then moves to presenting the translation module along with an extensive discussion on the particularities of the translation process and the potential optimization techniques. The chapter concludes by presenting the translation results for some selected ontologies from various domains.

— **Chapter 7** presents two potential applications of the new executable representation of ontologies. The work proposed in this chapter adopts an empirical approach and is based on realistic examples drawn from the domain of software engineering where ontologies in their executable form could be exploited in practical applied settings.

— **Chapter 8** finally concludes this thesis with a summary of thesis achievement and an outlook for future work.

1.4 Publications

This following conference and journal publications are related to the PhD thesis in hand and were realized during the course of working on it:

— Selena Baset and Kilian Stoffel. OntoJIT: Parsing native DL into executable ontologies in an object-oriented paradigm. In OWL: Experiences and Directions - Reasoner Evaluation, pages 1-14. Springer, 2016.

— S Baset and K Stoffel. OntoJIT: Exploiting CLR compiler support for performing entailment reasoning over executable ontologies. International Journal of Knowledge Engineering, 4(1):10-16, 2018.

— Selena Baset and Kilian Stoffel. Object-oriented software modeling with ontologies around. In Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering SEKE 2018, pages 29-33.

— Selena S Baset and Kilian Stoffel. Procedural extensions for executable ontologies in conventional software development. Data Science and Knowledge Engineering for Sensing Decision Support. October 2018, World Scientific, pages 870-877.

— Selena Baset and Kilian Stoffel. Object-Oriented Modeling with Ontologies Around: A Survey of Existing Approaches. International Journal of Software Engineering and Knowledge Engineering IJSEKE. 28-11n12, November 2018. pages 1775-1794.

Chapter 2

In the Land of Ontologies

$(B \vee \neg B) \dashv Question$

$Question \dashv (B \vee \neg B)$

Shakespeare, adapted

2.1 Introduction

Some of the most important manifestations of adopting a declarative paradigm are systems involving ontologies or Knowledge Representation (KR) systems as they are commonly known. In a KR system, ontologies are used as declarative specifications of terms in a given application domain. These are then operated on by some "standard" cross-domain engines that are orthogonal to the particular problem in hand. The fact that the supporting infrastructure surrounding ontologies does not need to be engineered for each particular domain, allows the focus of development to be shifted towards the specifications rather than the details of the implementation. In other words, from the perspective of programming languages paradigms, it is quite reasonable to say that ontologies are by design declarative. It is actually what makes them well suited for knowledge representation

tasks; but when other software engineering tasks are concerned, it is often where we hit the limits of what we can achieve with them.

This very last sentence is admittedly a strong affirmative one and we will be working on backing it with some support throughout the subsequent chapters by elaborating at length on the strengths and limitations of ontologies once taken into the realm of mainstream programming. Before going any further, though, we ought to pay tribute to ontologies in their native surrounding, KR systems.

We will therefore start this chapter with an overview of KR as the sub-field of AI research where ontologies emerged. We will then briefly cover other alternatives that were used before ontologies in their current format were adopted. Finally, we cover the literature of the most important languages and standards used to represent, share and "consume" ontologies.

2.2 KR as a sub-field of AI

Even though considered a contemporary field of research, the study of knowledge representation is as old as science itself. Its early ground thoughts root back in the ancient history when Greek philosophers were trying to answer metaphysics questions about the fundamental representational categories of existence, how existent objects relate to each other and the differences between reality and our perception of what is existent [2][3].

In our modern days the same term, or its more complete version Knowledge Representation and Reasoning (KRR), has emerged as a concrete field of research in artificial intelligence and computer science. It comprises the sub-field of research dedicated to studying formal high level descriptions of the world and the ability to find implicit knowledge out of explicitly represented facts via automated reasoning [4].

What makes the field of knowledge representation interesting is the perspective from which

it views human intelligence. Unlike other science branches such as philosophy, psychology or neuroscience that relate intelligent behavior to the biological and psychological aspects of the nervous system, knowledge representation argues that it is what humans *know* that allow them to behave intelligently and for this reason the study focus should be on the knowledge rather than the knower [5].

Putting this accent on *knowledge* rather than the *knower* is also what differentiates knowledge representation from machine learning [5]. Both fields are siblings in AI research and both aspire to reproduce intelligent behavior in machines; but while machine learning tries to imitate the perceiving and learning aspects of human intelligence, KR resorts to logic as a solid foundation to represent facts about the world and feed them to the machine that can, by means of deductive reasoning, deduce new ones that were otherwise implicit. The future AI, however, is an Artificial General Intelligence (AGI) whose models are equally good in perceiving and learning as well as in abstracting and reasoning. Conscious about these requirements, the AI scientific community has lately arrived to the revelation that AGI need not lie exclusively in one of these approaches but in the successful integration between the two [6]. That does not necessarily mean that the two camps have always enjoyed a harmonious history of coexistence. In the last decades both siblings have witnessed some ups and downs and the utility of applying one over the other has always been the subject of lively debates. Since their start in the 1950s up to the early 1970s, machine learning and other AI computation models inspired by neural connections have enjoyed a high season. Soon after, these models have gone into a winter period [7] [8] leaving the stage empty for expert systems involving ontologies and other KRR systems to take over. A few decades later, machine learning methods had their great comeback, not only due to theoretical advances in the field [9] but also thanks to other advances not directly related to AI such as the increasing computer memory and processing power and the large training data sets becoming readily available. Could the KR field also get a similar kind of push-forward triggers by some neighboring field of research? A new pervasive means of representation, perhaps.

2.3 Early KRR systems

By definition, KRR is "the field of study concerned with using formal symbols to represent a collection of prepositions believed by some putative agent" [5]. These symbols represent a finite set of prepositions out of the possibly infinite prepositions the agent believes. The field is also concerned with the computational mechanisms that help making implicit knowledge available to the agent when required. This is where the reasoning aspects of a KRR system come on the scene, according to Brachman and Levesque [5]: "Reasoning is the formal manipulations of symbols representing a collection of believed proposition to produce representation of new ones".

The literature on KRR started with some innovative but rather uncertain steps that mainly tried to copy the intuitions of human thinking. After some ups and downs and faced with some serious difficulties, these attempts were slowly abandoned leaving place to a more formal approach that was based on logic. Logic-based formalisms took over and this marked a disruptive start of a monotonic phase with steady advances that lasted for a couple of decades. Observing the KR research community nowadays, one encounters less and less papers on theoretical foundations but a wealth of application papers. With this diminishing slope of theoretical advances, one might assume that KRR as a field of research has reached a saturation point, but it could also mean another disruptive leap is ahead KRR systems. At this point, we might only speculate the future of KRR systems, but we can certainly look more in details at their history.

2.3.1 Frames and semantic networks

Semantic networks [10][11] and frames [12] were originally opted for rather specific representational tasks of human cognitive behavior but the formalisms behind these structures have evolved to serve as general-purpose tools for modeling problems in different domains. Nowadays, the general term network-based structures is loosely used to denote both se-

mantic networks and frame systems regardless of their different features [13][14][15].

Semantic networks borrow an important feature of human memory in which concepts are arranged in an associative network pattern. This feature was described by Sigmund Freud as free association [16]. He used it with his patients as a technique to stimulate memory retrieval. Assuming this associative network arrangement of concepts and given a seed concept, a patient can continually relate one concept to another until eventually stumbling across an important memory. Semantic networks build on this idea to represent concepts in a given domain as nodes in a directed network where connections between the nodes represent the relations between the concepts. Typical relations between concepts in semantic networks are the "AKO" (A kind of) and "IS_A" (Is a) [17]. Using these basic notions, semantic networks can easily represent a large number of relations between any number of objects in a given application domain.

Frames, in turn, stand for another representation technique that evolved from the cognitive intuition of semantic networks. In fact, they can be regarded as an implementation of semantic networks [12]. Frames follow a kind of object-oriented approach and are therefore more structured than their semantic network ancestors. Knowledge in a frame-based system is represented as a collection of frames in which each frame has a number of slots that can be filled with links to other frames. We can differentiate between generic frames denoting a class of objects and instance frames denoting individuals. Inheritance mechanism is supported between generic frames, as well as the possibility of an instance frame to override a default slot value provided by its generic frame. One substantial difference that frames have in comparison with semantic network is their inferential capabilities powered by a procedural extension to the declarative nature of the slots. This extension takes the form of triggers (Demons) attached to the slot as a means of controlling slot-level manipulations; they serve as slot self-checking mechanism to ensure frame correctness and consistency. Scripts are another procedural extension that frames have. A script is a type of frames that is used to indicate the steps needed to complete a task in a defined order.

Over the course of time, frames and semantic networks have failed to sustain their first appeal and they are less present in modern KRR applications. Yet, their basic intuitive ideas are what influenced the subsequent KR languages that we are reviewing in the following sections.

2.3.2 Logic-based formalisms

The proximity of semantic networks to human cognitive nature attracted practitioners to network-based systems but confronted with the lack of precise semantics, the pursuit of another means of providing semantics to representational structures grew larger. That was the moment when the role of logic in KR systems got into the spot light. One of the most notable examples on systems that first introduced logic into frame-based languages is the KL-One system. KL-One integrated the automated deductive reasoning of logic-based languages into hierarchical semantic networks [18]. Other frame-based languages have also seen some success in that area [19][20][21].

Logic offered a rich palette of declarative languages that are well suited for knowledge representation tasks. Owing to its simplicity, propositional logic was one of the first candidates for the job but at the same time, this simplicity also imposed many constraints on the kind of knowledge propositional logic can express. This limitation could be overcome by First Order Logic (FOL) with its quantification property. With FOL, one can construct abstract sentences that describe a set of objects instead of a specified instance. Given some variable x, this is achieved with the help of two kinds of quantifiers: 1) The universal quantifier $(\forall x)$ used to indicate that the sentence should hold true for any value of the variable x. 2) The existential quantifier $(\exists x)$ used to indicate that the sentence should hold true for at least one value of x. FOL offered the right balance between simplicity and expressiveness. For that reason, most knowledgebase systems capture domain knowledge using a fragment of first-order predicate calculus equipped with certain reasoning capabilities to entail implicit knowledge by means of logical consequences [4]. What is worth

highlighting here, however, is that the more expressive the language is the more complex reasoning becomes. This is because with more expressiveness the overhead of proving the soundness of reasoning, i.e. entailing only true facts, and proving completeness, i.e. entailing all true facts, increases drastically. This trade off between expressiveness and reasoning complexity is the reason behind the many logical languages derived from FOL (subsets or supersets of FOL) and the countless number of research papers analyzing these languages. Beside FOL, many other non-monotonic logics have also emerged such as default logic, auto-epistemic logic, some modal logics and many others [22][23]. For the sake of our research topic, however, we will limit this review to logics derived from First Order Logic.

Prolog

Originated from research on automated theorem proving, Prolog or PROgramming in LOGic is a class of programming languages based on a Turing-complete subset of FOL called Horn Clauses. Taking a subset of FOL allowed for simpler and faster inference through resolution [24]. A program in Prolog is a collection of predicates representing facts and rules. The user initiates program execution by posing a query. Given the user query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, this means that the union of clauses of the negated query is false. It follows that the original query, with the appropriate variable bindings in place, is a logical consequence of the program [25].

One particular implementation of Prolog is AnsProlog* or or A-Prolog [26]. It shares with pure Prolog its syntactic form but differs in the semantics behind. AnsProlog* is based on answer sets semantics. While queries in both AnsProlog and pure Prolog are executed in a top down approach, AnsProlog* literature argues that logic programming with answer set semantics is more suited for knowledge representation tasks since pure Prolog is not a declarative language; pure Prolog is sensitive to the order of rules and the

order of literals in the body of rules, the thing that can introduce infinite loops in the case of inappropriate order. Pure Prolog is also less expressive in the propositional case since it does not allow for disjunction in the head of the rule [27].

Nowadays and regardless of the implementation details, Prolog is considered as a general-purpose declarative programming language. Many extensions to the original Prolog have also emerged; for example SWI-Prolog [28] conforms to the HTML and XML web protocols to add support for server-side web programming. Other semantic web extensions have also been added to Prolog to support OWL and RDF [29][30]. Flora-2 is an object-oriented rule-based system for knowledge representation and reasoning that is derived from F-Logic [31][32].

2.3.3 Description Logics for knowledge representation

Description Logics (DL) is the most recent term [1] denoting a family of languages derived from FOL [4]. In term of expressiveness, DL languages are positioned between propositional logic and FOL and reasoning is decidable for the vast majority of problems in DL. In knowledge representation systems, DL has substantially helped overcoming the deficiency of semantic networks by providing them with formal and declarative semantics [33]. The main modeling blocks in DL are logical statements, i.e. axioms, describing concepts, roles, individuals and their relations. A finite set of axioms is then used to encapsulate the domain knowledge in a knowledgebase (KB) compromising two main components: The terminological box $TBox$ and the assertional box $ABox$. The $TBox$ defines the relevant concepts of the target domain while the $ABox$ uses the vocabulary from $TBox$ to describe properties of individuals occurring in that domain. The vocabulary used in the $TBox$ incorporates concepts that denote sets of individuals and roles that denote binary relations between individuals. Some sources in the literature prefer to further split the terminological axioms in two groups rather than one: The $TBox$ for concepts and an-

1. Other older terms were: terminological systems and concept languages.

other *RBox* for role terminological axioms [4]. The KB users can then use combinations of atomic concepts ("elementary descriptions from which we inductively build complex descriptions by means of constructors" [4]) and roles to construct more complex descriptions. The set of possible concept combinations in a DL language is determined by the constructors the language allows, and that in turn characterizes the expressiveness and complexity of reasoning for that language. The minimal DL language that is of interest in practical application is the attributive language $\mathcal{AL}$ [34]. The following set of syntax rules is used to form concept descriptions in $\mathcal{AL}$:

$$
\begin{array}{rcll}
C, D & \rightarrow & A \mid & \text{atomic concept} \\
 & & \top \mid & \text{universal concept} \\
 & & \bot \mid & \text{bottom concept} \\
 & & \neg A \mid & \text{atomic negation} \\
 & & C \sqcap D \mid & \text{intersection} \\
 & & \forall R.C \mid & \text{value restriction} \\
 & & \exists R.\top \mid & \text{limited existential quantification}
\end{array}
$$

Other DL languages are either subsets of, or extensions to, the $\mathcal{AL}$ language. They can be obtained by restricting or allowing constructors or concept/role operators. For example, if, in addition to the syntax rules above, the full existential quantification ($\mathcal{E}$ constructor in DL) and number restriction ($\mathcal{N}$ constructor) were also allowed, the resulting extension of $\mathcal{AL}$ is called $\mathcal{ALEN}$. Table 2.1 provides a list of DL constructors along with some of the most notable basic DL fragments. The full list of possible DL fragments/extensions goes long and covering them in the context of this chapter can quickly go out of scope. We refer interested readers to the work in [36] that provides a comprehensive survey for operators of DL languages and other less known fragments of FOL.

As for the practical use of DL languages, different application domains contributed to the success of DL languages in KR systems. In information systems, the CLASSIC system,

Table 2.1 – List of DL constructors

Constructor	Function
$\mathcal{AL}$	Attributive language (atomic negation, concept intersection, universal restrictions, limited existential quantification).
$\mathcal{FL}$	Frame-based description language (concept intersection, universal restrictions, limited existential quantification and role restriction).
$\mathcal{FL}^-$	A sub-language of $\mathcal{FL}$ obtained by disallowing role restriction (equivalent to $\mathcal{AL}$ without atomic negation).
$\mathcal{FL}_0$	A sub-language of $\mathcal{FL}^-$, obtained by disallowing limited existential quantification.
$\mathcal{S}$	An abbreviation for $\mathcal{AL}$ with transitive properties.
$\mathcal{C}$	Complex concept negation.
$\mathcal{U}$	Concept union.
$\mathcal{E}$	Full existential qualification.
$\mathcal{H}$	Role hierarchy.
$\mathcal{R}$	Limited complex role inclusion axioms.
$\mathcal{O}$	Nominals.
$\mathcal{I}$	Inverse properties.
$\mathcal{F}$	Functional properties.
$\mathcal{N}$	Cardinality restrictions.
$\mathcal{Q}$	Qualified cardinality restrictions.
(D)	Use of datatype properties, data values or data types.

Reproduced from multiple sources including the Description Logics handbook [4] and Protégé documentation [35].

developed at AT&T, was the first implementation of a DL system [37]. Rule extensions and the ability to provide explanations for the reasoning results obtained was later added to KR systems in the domain of computer configurations [38]. The bio-medical domain has also contributed to the construction and maintenance of very large ontologies to organize medical knowledge [39]. More recently, the Semantic Web, or Web 3.0 as it is sometimes referred to, is the domain where more efforts are put to extend the web with formal semantics in order to allow automated agents to understand and link between the information content of web resources [15].

2.3.4 Ontologies

The previous sections briefly covered KR systems and languages without making explicit reference to the exact term ontologies. The simple reason for this is that ontologies are nothing but the manifestation of the long efforts put on developing mature KR languages. The key aspects of an ontology are put together by Gruber et al. in 1993: "An ontology is a formal and explicit specification of a shared conceptualization" [40][41]. A bit less concise definition would interpret ontologies as models used by computer systems to describe and share knowledge about the real world. This is achieved by explicitly defining the concepts relevant to the domain, and the relationships between these concepts, using a formal computer language to avoid ambiguity or incomplete specifications. The formal computer language here, after a few predecessors, is the Web Ontology Language (OWL); a language directly based on one of the DL languages discussed in the previous section. An anatomical cut of a typical ontology reflects a hierarchical structure of $TBox$ concepts along with the $ABox$ individuals, the relationships among concepts and/or individuals and finally the attributes describing concepts and individuals and their values.

With the questions on the theoretical basis for ontologies largely answered, current research is concerned with finding solutions for issues related to their use in practice. For example, one of the main principles to follow when building ontologies is ontology sharing and reusibility. Ontologies can have different levels of granularity where upper or top-level ontologies describe very general and domain-independent concepts which in turn can be used in building other domain-specific ontologies with somehow finer-grained concepts. This practice has created some redundancy in major ontologies across application domains. Consequently, the research on ontologies in recent years continues to focus on properly defining design patterns for building ontologies as well as finding better solutions to the problem of ontology refactoring and ontology alignment [42].

2.4 On the technical side, languages and standards

2.4.1 Overview

What differentiates ontological modeling from other modeling paradigms such as the Unified Modeling Language (UML) or Entity Relationship diagrams is that ontologies are intended a priori to be shared and they need therefore to have formal precise semantics. This calls for a standard ontological language to be used when building and sharing ontologies. In response to this call, many ontology languages were proposed during the last two decades. Most notably we have the frame-based FLogic by Kifer et al. [19] and DAML+OIL By Horrocks and McGuinness et al. [43] [44]. Nowadays, ontology modeling is largely dominated by the The Web Ontology Language (OWL), the W3C standard language for the semantic web. Another important standard for building and sharing ontologies is the Resource Description Framework (RDF) [45], the specification adopted by W3C for describing and interchanging metadata on the Web [46]. Along with OWL and RDF we find SPARQL; another important standard that is used to query ontologies. We will provide each of these three standards with a brief description in the following sections.

2.4.2 The Web Ontology Language (OWL)

The Web Ontology Language (OWL) has made long strides to allocate itself a distinctive spot as the The World Wide Web Consortium (W3C) language of choice for building and maintaining ontologies in the Semantic Web. OWL has two versions: OWL and OWL 2; and both versions have many sub-languages that are varying in expressiveness at an increasing complexity overhead [47][48]. These sub-languages can be seen as different flavors tailored to fulfill the different needs of ontology stakeholders.

The most restricted sub-language is OWL Lite and the most expressive one is OWL Full

which has a very expressive vocabulary but is not anymore decidable. In between OWL Lite and OWL Full, we find OWL DL, a language based on Description Logics that offers the right balance between expressiveness and decidability for most KR applications [49]. In the first version of OWL, OWL DL is based on the $\mathcal{SHOIN}(\mathcal{D})$ extension of $\mathcal{AL}$ whereas OWL 2 DL is based on $\mathcal{SROIQ}(\mathcal{D})$ [50] and is further segmented into the following profiles: OWL EL for ontologies containing very large numbers of properties and/or classes, OWL RL intended for large volumes of instance data and finally OWL QL usually used when scalable reasoning is desired without sacrificing too much expressive power [47][48].

Ontologies defined in OWL consists of classes, properties and individuals all of which are designated by axioms of class, data ranges, datatypes and expressions for object properties [47]. An OWL ontology $\mathcal{O}$ is a finite set of axioms that describe intentional knowledge (*TBox* and *RBox* axioms in DL terms) or existential knowledge (*ABox*) assertions [4].

2.4.3 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a specification adopted by the W3C for describing and interchanging metadata on the Web [46]. Throughout its history, RDF has undergone many transformations that resulted in RDF possessing a variety of syntax notations and data serialization formats but as an abstract model, it had always maintained the original purpose it was created for which is supporting the semantic web in "bringing structure to meaningful content of web pages". One can relate the role of RDF in describing a resource information to the role of HTML in visually rendering the content of a web page. The main difference is that HTML renders content intended for human receptors whereas RDF is concerned with annotating web content to become accessible for machines. Basically, all RDF notations rely on triplets, a very simple but yet efficient format, to capture and exchange knowledge. A triplet takes the form of `(s,p,o)` tuples or (subject, predicate, object) tuples where the subject `(s)` -a Uniform Resource Indicator

(URI) - denotes the resource and the predicate `(p)` denotes a relationship between the subject and the object `(o)` which in turn can be denoted either via a URI or some literal of a primitive data type. An RDF document is then simply a collection of triplets encoding statements and/or assertions about the resource being described. These collections can be easily fitted into a labeled, directed multi-graph making RDF a suitable data model for certain kinds of knowledge representation tasks.

Ontological extensions to the RDF model, such as RDF Schema (RDFS) [51], are also available. RDFS is built on top of RDF but it provides a more expressive set of vocabulary (e.g. `rdfs : domain`, `rdfs : range` or `rdfs : subclassof`) to add ontological structure and enrich the semantics of RDF documents.

2.4.4 SPARQL

The name SPARQL is a recursive acronym for Sparql Protocol and Query Language. It was made a standard by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium. SPARQL 1.0 became an official W3C recommendation in 2008 followed by the current version 1.1 in 2013. Just like SQL is the standard query language for relational databases, SPRAQL is the query language for RDF and RDFS data sources [52]. It can also be used to perform some queries against OWL ontologies. SPARQL uses a syntax that is similar to the syntax of SQL. It also provides some keywords to support analytical operations such as JOIN, SORT and AGGREGATE. The difference SPARQL has to the more traditional SQL is that since SPARQL queries are executed against RDF databases (RDF graphs), it allows for greater flexibility; like for example the support for querying multiple graphs and semi-structured data or entailing implicit relations. Apart from the querying specifications, SPARQL also defines a protocol which dictates the manner in which queries are executed over the web and the set of supported formats for the query results. SPARQL queries are encapsulated in HTTP endpoints that return query results in a variety of formats including XML, JSON and Others [53][54].

2.5 Ontologies from a software engineering perspective

The folks in the neighboring software engineering community have never turned a blind eye into the evolution of ontologies, even if these were the product of a different field of research. This is because modeling, ontological or not, is a core activity in the life cycle of any software project. The intersection of interest between the two communities resulted in a whole line of literature on the relationship between ontologies and software modeling and the confusion caused by the variety of their different associated interpretations [55][56][57]. Chapter 4 is completely dedicated to providing a thorough scan of this particular area of research.

Language wise, in parallel to the fast-paced development of languages and standards in knowledge representation and semantic web communities, the software engineering community was also putting similar efforts into crystallizing a standard language for ontology modeling. Already in 2001, Cranfield [58] proposed using UML for representing ontologies, a proposition backed by the wide-acceptance of UML among software engineers. In 2003, the Object Modeling Group (OMG) released its first version of a dedicated Ontology Definition Metamodel (ODM) specification [2]. The ODM specification defines the mapping between the meta-models of OWL and RDFS and the Common Logics Meta-model [59]. ODM is now in its 1.1 version which was released in September 2014. A use case [3] from IBM provided a prove-of-concept implementation of the ODM specification. This use case is probably the most notable example of UML ontologies. Apart from the IBM use case, comparing the sparse results one gets digging the literature for UML ontologies to the number of readily available OWL ontologies, one can safely assume a limited usage of UML profiles for constructing ontologies compared to other options such as OWL or RDFS.

2. OMG ODM: https://www.omg.org/spec/ODM/
3. IBM Semantics Toolkit: http://freshmeat.sourceforge.net/projects/ibmsemanticstoolkit

Chapter 3

On Programming Languages and Paradigms

> Computers are good at following instructions, but not at reading your mind.
>
> Donald Knuth, The TeXbook, 1984

3.1 Introduction

The last chapter provided a brief review on languages used in the realm of knowledge representation and reasoning. Logically, we should equally cover languages used in the realm of software engineering, except this latter seems just too vast and the aspects surrounding programming languages are way too many to be condensed in one single chapter. Instead, what we try to do in this chapter is to shed some light into certain areas related to programming languages, their design and the paradigms associated with them. The review provided here does not attempt at being inclusive nor systematic yet

we justify it by our wish to provide the reader with a perspective from which we can share a train of thoughts when reading through the following chapters.

3.2 On programming languages

With each one of the three generations of programming languages [1], we are climbing up the abstraction ladder a bit further. This means that with each generation we need to worry less about lower level aspects of computer languages and can, hopefully, gear our focus towards more effective and expressive human-computer interaction. Yet, our aspiration to program, or rather to converse with, the computer in a "lingua-franca" might span any time from a few years to a few decades. For the time being, with the current generation of modern programming languages, many design aspects are still not to be taken for granted when making our choice for a programming language.

3.2.1 Typing aspects

Among the most perceptible forms of abstraction we can see in modern programming languages compared to earlier generations of binary and assembly languages is type abstraction. The support for modular typing allows the programmer to think of data structures at a higher level than the assembly registers or the binary 1s and 0s. This applies to all kinds of constructs in a computer program from a single variable type declaration to expressions and functions return types to modules of different sizes. The type system of a programming language is the set of rules that govern assigning the type property to these various constructs [60]. Beside enabling certain compiler optimizations, the main purpose of a type system is to reduce execution errors due to type mismatch by delegating type consistency checking to the compiler (or sometimes the interpreter) [61].

1. Five according to some less formal classifications.

A type error is an unintended discrepancy in the type assignment that might manifest in multiple stages of a program life cycle from development to execution. The timing and extent of mechanisms the type system uses to detect, or prevent, type errors constitute a criterion based on which programming languages are usually categorized. Programming languages are categorized into statically- or dynamically-typed depending on when type checking happens.

In **statically-typed languages**, type checking happens at compile time. This can help in detecting errors earlier before the program execution, which should increase the reliability of the delivered program [61]. Static typing also results in compiled code that executes faster. This is because the compiler can reuse the the exact type information that was first obtained during static type verification to produce optimized machine code. Java, Haskell, C and C++ are among the most notable statically-typed languages.

In contrast, by deferring type checking to runtime, the compilers (or interpreters) of **dynamically-typed languages** have less code to revisit and can run faster contributing to a shorter edit-compile cycles [62]. Dynamic typing also allows constructs that some static type checking would reject as illegal (as for example in allowing mock objects to be transparently used in place of full data structures); this can come very handy when prototyping or when writing transitional code. Perl, Lisp, Python, PHP and JavaScript are all examples of programming languages that are dynamically-typed. On the other hand, dynamic typing provides no guarantees to avoid type errors that can manifest at runtime and may lead to runtime failures. As a middle solution, many languages, such as C# and Java now support, to a certain extent, a combination of static and dynamic type checking [63].

Programming languages are also categorized into **strongly-typed** or **weakly(loosely)-typed** languages depending on the extent of the constraints that are imposed by the compiler when assigning or converting types of a programming construct [64]. This last distinction is used in a less strict manner since no universally accepted definition exists to

precisely separate between the two categories. In fact, this categorization is not completely disjoint from the previous one as the choice of static or dynamic type checking is among the deciding factors on how "strong" typing in a programming language is. Coining the extra checks with type correctness, i.e. type safety, the compiler of strongly-typed languages, such as C++ or Java, requires the programmer to explicitly declare variables and method return types; it also limits type conversion operations to explicit casting with no information loss. On the other end of the spectrum, loosely-typed languages, such as Perl, allow for greater flexibility in type declaration and assignment. In this kind of languages, the programmer need not declare the type of a variable or the return type of a function; these can be inferred by the compiler. Loosely-typed languages also support implicit type conversion which allows for using a value of one type as if it was a value of another type.

Like the choice between static and dynamic typing, deciding on how strongly-typed a programming language should be involves certain trade-offs. Strong typing advocates believe that in addition to the increased type safety, the code of programs written in a strongly-typed language enjoys greater readability thanks to the more verbose type declaration imposed by the compiler. Weak typing advocates argue that the enhanced code readability comes most of the times at the price of rigid and less efficient developing experience. Many different shades exist to honor the views of both camps and languages like Common Lisp, F# or Haskell permit implicit type conversions in a type-safe manner. Ultimately, type errors constitute one family of programming errors among many others, and the impact of type checking policy depends on the percentage of type errors with respect to the overall possible errors.

3.2.2 Support for dynamic compilation

Compilers of modern programming languages support, to a greatly-varying extent, a sort of compilation called dynamic compilation. Dynamic compilation is an implementation

technique deployed by some programming languages to improve program execution performance by combining the two traditional approaches to translation to machine code: Interpretation and Ahead-of-Time compilation (AOT) [65]. AOT compilers translate, possibly over some intermediate steps, the program code written in a high-level language such as C or C++, or an intermediate language such as .NET Common Intermediate Language (CIL) or Java bytecode into an optimized native machine code. Interpreters, on the other hand, translate the code line by line and perform execution immediately eliminating the need for the intermediate compilation steps. Generally speaking, compiled programs run faster since the compilation process devotes an arbitrary amount of time to code analysis and optimizations. This performance gain in compiled programs comes at the price of less flexibility compared to interpreted programs that tend to be smaller and more portable; in addition, interpreters have access to run-time information that are unobtainable when performing static analysis prior to execution.

Dynamic compilation aims at getting the advantages of both approaches, i.e. the speed of compiled code with the flexibility of interpretation. A dynamic compiler continues translating high level code after the program execution has started so that the compiler would have access to the runtime environment information that were unavailable to ahead-of-time compilers and can therefore enjoy the flexibility of interpretation while maintaining the performance optimization of compilation. The principle here is to selectively compile parts of the program code "hot spots" and delay, or sometimes even avoid, evaluating other infrequent parts. This can sometimes result in substantial performance gain. Unfortunately, deploying dynamic compilation is no free lunch either [66]. Runtime environments using dynamic compilation typically suffer from an initial performance lag; they have programs run slowly for the first few minutes, and then after that, when most of the compilation and recompilation is done, it runs quickly afterwards. That is why dynamic compilation is undesirable in certain cases.

A closely related technique is incremental compilation. To support incremental compila-

tion, the compiler needs to be part of the runtime system. In consequence, source code can be read in at any time and from multiple sources including data structures constructed by the running program and translated into a machine code which is then immediately available for program use. Incremental compilation also provides a mixture of the benefits of interpreted and compiled languages. Unlike dynamic compilation, however, incremental compilation does not involve further optimizations after the program is first run.

3.2.3 Other aspects of a programming language

Technically speaking and as it will become more apparent in the following chapters, the flexibility of the language typing system and a strong support for dynamic compilation are key aspects when considering the use of a programming language to implement the proposed approach. On a general more strategic level however, democratizing ontologies by representing them using a programming language would only be fruitful if the programming language in use is itself easily accessible; a mainstream language in other words.

Many factors can contribute to a programming language becoming mainstream: clean design choices, efficient memory management, cross platform portability, runtime performance, a modest learning curve, expanded use in production, community support, good strong libraries and last but not least the programming paradigms supported by the language. Getting a hang of all these factors when evaluating a programming language is certainly not a walk in the park. Typically, the nature of the problem in hand will have a say on our language of choice.

3.3 On programming paradigms

3.3.1 The early conception

Research in software engineering has followed a different evolution path compared to the other fields of computer science where the theoretical foundations to answer research questions were mature enough before they can be consumed in industry. The questions arising in software engineering communities were maybe less ambitious but certainly more imminent in the sense that they were related to the operational aspects of software systems already in use across enterprises of large scale.

With the expanding use of database systems and the growing diversity of enterprise software technical stacks, many problems have risen when trying to connect or move around data models between different software systems. The name impedance mismatch was commonly used to denote this kind of difficulties concealing the differences between the database model and the programming language. The urgent need to find solutions to problems such as the impedance mismatch resulted in new technical trends being implemented in parallel, if not one step ahead, of their theoretical foundations.

Over the years, the accumulating collective wisdom gained while dealing with such kind of questions have clustered around a few schools of thinking that we now call programming paradigms. Like in other scientific disciplines, programming paradigms adhere to Kuhn's definition of a scientific paradigm being the set of "universally recognized scientific achievements that, for a time, provide model problems and solutions for a community of practitioners" [67]. Where programming paradigms might diverge from the Kuhnian view of scientific paradigms is their evolution cycle. While the emergence of a new programming paradigm might "shift" around the standards in software development practice, it is certainly not meant to take over or replace existing paradigms as is usually the case with paradigm shifts in other scientific disciplines. This can explain the coexistence of

numerous programming paradigms. To that end, the complete list of paradigms currently in practice grows long and is certainly out of the current scope. That is why, in the following paragraphs will only briefly highlight a handful of them, those we believe are essential to keep in mind while reading through the rest of this thesis.

3.3.2 Some notable paradigms

Among the first identified programming paradigms is the structural programming paradigm that came into place with the rise of the third generation of programming languages allowing the organization of the program code in blocks. Procedural programming is another paradigm directly derived from structural programming. In that paradigm, procedures, also known as subroutines are used to dictate to the machine a series of computational steps to be carried out in order to achieve the program goal. Both structural and procedural programming are examples of a family of paradigms called imperative programming paradigms. In imperative programming, the programmer decides on how the control flow is used to manage a program's state and its output; this includes the sequence and order of code statements to be executed. On a higher level of abstraction, we have the family of declarative programming languages where the programmer delegates the details of managing the program state to the language and focuses instead on stating what needs to be done rather than how. Some of the most known declarative paradigms are the functional and logical paradigms. The logical paradigm views computation as automated reasoning over a body of knowledge. The last chapter explained at length the kind of languages and tools that can help in realizing this view. Functional programming, on the other hand, is a paradigm that favors stateless programming by avoiding the side effects of variable assignments. It views computation as a composition of mathematical functions where the value of the passed variable remains intact after function execution. For that reason, functional programming makes extensive use of recursion and anonymous functions (lambda expressions).

So far we have deliberately left out the Object Oriented Paradigm (OOP). A paradigm that is omnipresent in almost all kinds of software projects. In this paradigm, the domain data model is reflected into a set of objects pertaining into a defined set of classes. Classes, from the most abstract to most concrete, are organized through inheritance relationships in a tree hierarchy. Objects are considered first class citizens; their internal state is hidden and only properly accessible through the corresponding class interface. In OOP terms this practice is known as encapsulation. Another OOP concept is polymorphism; a design principle that enables a single interface to stand for different named type implementations as long as these implementations adhere to the declared public interface. Together, inheritance, encapsulation and polymorphism form the core principles of OOP design. The object oriented paradigm is arguably listed under the imperative paradigm but the number of OOP projects written in mostly-declarative languages suggests that the OOP paradigm has less to do with being imperative or declarative and more with certain data modeling aspects as we shall see in the next chapter.

3.3.3 Multi-paradigm programming

As described earlier in this section, programming paradigms had their origin in industry. Eventually, they also enjoyed some presence in academia [68][69]; but to say that programming paradigms were always received with enthusiasm among scholars in the domain is certainly an exaggeration. According to Hooper [70]: "The trouble with programming paradigms is that it is rather difficult to say what one is, or how we know we have a new one. Without precise definitions, nothing precise can be said, and no conclusions can be drawn, either retrospectively or prospectively, about language design. Yet the convention of organizing languages into paradigms is so pervasive, I am asked to justify not adhering to it[2]". On the other extreme end, some developers are very zealous about one specific paradigm over the others to the point of being religious. Our humble view is

2. http://www.cambridgeblog.org/2017/05/what-if-anything-is-a-programming-paradigm/

that neither *paradigm atheism* nor *paradigm rigidity* is an answer to any of the recurrent questions in software design and development but *paradigm agility* maybe is. Fortunately, this view seems to be the common wisdom both in industry as well as in academia and multi-paradigm programming is on the rise.

3.4 Choice of the target language

Most of the propositions and ideas discussed in this thesis are applicable using many mainstream programming languages. Yet, a choice still had to be made for a programming language that, when used as a representation means, is most helpful in achieving our goal in democratizing access to ontologies. In the light of our literature review and the technicalities of the translation problem, our retained language was C#. In the following paragraphs, we motivate our choice by recapping some of the main reasons. We will deliberately leave out some more secondary reasons as these will become more apparent in the chapters where we cover the technical details of the translation.

C# is a strongly-typed language but provides strong support for loose typing. One of the problems developers usually face with programmable access to ontologies, and other RDF sources in general, is that existing tools provide a generic access that lacks the ability to perform compile-time error checking. Without this safety net, type errors can pop up at runtime. Translating the concepts of the source ontology into types in a strongly-typed language solves this problem. On the other hand, explicit type information can either be unavailable beforehand in the source ontology or implicit and only deductively inferable. A flexible loosely-typing mechanism is hence needed in the target language. Generic typing, equally supported in C# and Java, can only partially meet this requirement. C# supports the possibility for strongly-typed yet implicit variable declaration that is coupled with a dynamic type resolution offered by the compiler. A combination that comes in handy for translating implicit type information in the source.

C# is a multi-paradigm language. In its very first release in the early 2000s, C# was conceived as an imperative object-oriented language. Over the many successive releases, however, the language has evolved into a multi-paradigm language. Features such as lambda expressions, extension methods and anonymous classes made it possible to adhere to a more declarative (functional) programming paradigm while using C#. For our objective of expressing ontologies in a general-purpose programming language and given the purely declarative nature of ontologies, the fact that the language can support more than one programming paradigm can ease the integration. In that sense, Scala could have been an equally good option as it is a multi-paradigmic language by-design. However, with the language popularity and a large established base of developers among our criteria, Scala, though with a positive slope on the popularity index [3], still lags behind compared to other mainstream languages.

The Language Integrated Query (LINQ) mechanism. One of the most recurrent tasks in KR systems is querying the knowledge base for answering questions that involve some deductive reasoning. Having the query mechanism directly integrated in the programming environment is an important factor in minimizing the learning curve the developer has to overcome when writing this new kind of queries. LINQ queries, written as native C# code statements, are agnostic to the underlying, possibly heterogeneous, data sources. The expressiveness of LINQ is comparable to other query languages such as SQL but the fact that LINQ can be used in combination with C# lambda expressions and anonymous classes gives the developer the possibility to express more advanced querying scenarios that can involve entailment reasoning.

3. TOIBE programming language popularity index: https://www.tiobe.com/tiobe-index/

3.5 Summary and outlook

This chapter served as a brief, and rather selective, review on some aspects of programming languages and paradigms. We tried to limit our review to the aspects that are most important when considering the approach proposed in this thesis, i.e. the use of programming languages as means to democratizing access to ontologies.

In the next chapter, we are set to dig a bit deeper into the literature in order to identify the state-of-the-art on ontology integration approaches before proposing our own approach.

Chapter 4

The Pursuit of Δ

"In the good old days physicists repeated each other's experiments, just to be sure. Today they stick to FORTRAN, so that they can share each other's programs, bugs included."

Edsger Dijkstra

4.1 Introduction

In software development, like in other engineering disciplines, model sharing is always an encouraged practice. It explains the industry's constant pursuit of open standards for modeling languages that allow for seamless incorporation of models pertaining to a certain modeling school into another. Ontological modeling is no exception. After a few predecessors, Ontolingua [40] [72] and DAML+OIL [43], the Web Ontology Language (OWL) [50] [48] is now the standard language for developing and sharing ontologies in the semantic web as well as many other fields such as the biomedical domain. In software and knowledge engineering literature, there exist many attempts at integrating ontologies into mainstream development. These attempts vary from loose integration, i.e. accessing ontologies from a programming language, to a more solid transformation from OWL ontologies into software models. The synergies between ontologies and software models might seem so evident that in many cases an effortless mapping between the two paradigms is taken for granted. This assumption is further supported by a considerable number of proposed development frameworks such as the Ontology Driven Software Development (ODSD) [73] or Ontology-oriented Programming [57]. However, shifting a bit from the research state-of-the-art into the circles of conventional software development, we observe a different image than the one painted in research papers. Despite the many integration tools proposed, developers are still reluctant to incorporate ontologies into their code repositories.

This chapter has for an objective the identification of the different reasons behind the modest adoption of ontologies in software development communities. It will take the form of a survey that covers existing integration approaches between ontologies and software modeling languages and practices. The survey output is a classification framework that can be used by researchers interested in the topic as well as an analysis of some of the common challenges and implicit reasons beyond what have been addressed in the literature.

4.2 The survey scope and method

Given the qualitative nature of the literature and the diversity of the technical spaces in which ontology integration is implemented, it is difficult to establish a unified criterion for automatic classification of existing mapping approaches. In order to define a concise and coherent scope, we will limit our review to the question of integrating ontologies into an object-oriented programming language. This automatically excludes papers on integrating ontologies into a specific profile programming language such as LOOM [1] or Prolog.

The question we are concerned with in this survey was first addressed in three of the earliest papers in the domain: Eberhart 2002 [74], Kalynapur et al. 2004 [75] and Knublauch 2004 [73]. While the mentioned papers are relatively old, adopting these three papers as seeds for our search allows for a more inclusive time window when collecting other related papers. Using Google scholar, we harvested all papers that cited at least one of the seeds. This resulted in around 310 papers. We first grouped similar papers in sets (e.g. papers originating from the same author and/or proposing the same tool) and we chose a representative paper of each group. We dropped irrelevant papers such as papers accentuating the application domain rather than the integration approach. We then used Google citations metrics (i.e. the number of citations per paper) as an indicative measure of the impact of a paper to pivot our manual inspection of more papers to include. Using the high impact papers as seeds, we collected -manually this second round- some more relevant papers that were not automatically harvested. At the end of the process we retained 24 approach papers that we are surveying in this chapter besides 3 surveys and position papers. While by no means an exclusive list of all papers in the field, the retained papers give a good indicator on the current state of research.

When reviewing papers, we focused on certain aspects like the extent of integration and

1. LOOM is a frame-based language that has a formal semantics for mapping its declarations to set theory and First Order Logic.
https://www.isi.edu/isd/LOOM/

the challenges the authors faced rather than focusing on the motivation behind each contribution which was, more or less, common behind most of the reviewed papers.

4.3 A Classification of ontology integration approaches

Ontologies, by design, are not intended as standalone software units [76]. They need to be considered in the context of an application that is responsible for accessing and manipulating the concepts they represent. For that reason, it is essential to provide some mapping between the content of an ontology and the application environment in which it resides.

Scanning the literature, we can identify some kind of a classification of the possible embeddings of an ontology in a programming environment. In [57], Goldman opposes generic application development using ontologies to ontology-specific programming. Generic programming requires no knowledge of the semantics of specific ontologies (like in query answering engines and consistency checkers) whereas "ontology-specific programming deals with models based on one or more specific ontologies". Puleston et al. [56] also differentiate between three approaches: A direct approach where ontologies are transformed to code and an indirect one where access to ontologies is provided via APIs and a third hybrid approach that is proposed by the authors themselves. Many other authors proposed their vision on integrating ontologies into conventional software development with different acronyms and terms to denote this integration; as examples we have the afford mentioned Ontology Driven Software Development (ODSD) by Knublauch [73] and Ontology-Oriented Programming by Clark [77] and many others. In general, we can differentiate between two main camps: domain-aware vs. domain-neutral application development. The more we move toward domain-aware application development the more it becomes important to abandon generic-access and to provide the developer with a transparent means to access and manipulate the content of the ontology.

In this section we propose a classification of the surveyed approaches for ontology integration. We build on top of what has been done in the literature in order to provide a comprehensive taxonomy of the existing tools and approaches based on their technical characteristics and their resulting artifacts. For the sake of simplicity, since most of the surveyed approaches addressed integrating OWL ontologies into an object-oriented paradigm, we will be using the term OWL-to-OOP mapping as an umbrella term to root the taxonomy despite the fact that this term is a stretch for a few of the approaches surveyed in this chapter. Table 4.1 provides a list summary of the surveyed tools and approaches. The proposed taxonomy, as shown in Figure 4.1, is constructed using this list of tools. The inner levels of the tree are based on the "Mode" column whereas leaf nodes represent the surveyed tools.

In the following sub-sections we traverse the taxonomy explaining the logic behind it and providing an overview of leave nodes in each branch.

4.3.1 Passive OWL-to-OOP mapping

Passive OWL-to-OOP mapping depicts a generic and a rather loose mapping between the content of an ontology and its programming environment. Nevertheless, this approach is more dominant for most applications on the semantic web. In this approach, ontologies are integrated into the mainstream language simply by loading them into memory. Loading is achieved by an ontology loader that transforms the ontology from its syntactic form, e.g. RDF/XML, into an in-memory representation. This in-memory representation can be an Abstract Syntax Tree (AST) like in the case of OWL API loader [96] [79], or an RDF triple-based structure like the one used in Jena [80]. In either case, the loaded ontology is treated as data and will reside in the data segment of the program allocated memory, hence the name passive approach.

This approach of providing a generic access to the content of ontologies might be less

Table 4.1 – List of main tools and approaches for OWL to OOP mapping.

Year	Tool/Approach	Source	Target	Mode	Reasoning
2002	OntoJava [78]	RDF(S)	Java	Active/Static	None
2003	Goldman [57]	OWl	C#	Active/Static	None
2003	OWL API [79]	OWL	Java	passive	External
2004	Knublauch [73]	OWL	Java	–	–
2004	HarmonIA [75]	OWL	Java	Active/Static	None
2004	Jena [80]	OWL	Java	Passive	External
2005	SWCLOS [81]	OWL	CommonLisp	Active/Dynamic	Limited
2005	RDFReactor [82]	RDF/RDF(S)	Java	Active/Static	None
2006	Atkinson et al. [55]	OWL	UML	–	–
2006	Babik [83]	OWL	Python	Active/Dynamic	Limited
2006	Clark et al. [77]	–	Go!	–	Supported
2007	ActiveRDF [84]	RDF(s)	Ruby	Active/Static	None
2007	Athanasiadis [85]	RDF / OWL	JavaBeans	Active/Static	–
2007	Owlet [86]	OWL	Java	passive	Supported
2008	Puleston et al. [56]	OWL	OWL/Java	–	–
2009	OWL2Java [87]	OWL	Java	Active/Static	None
2009	O3L [88]	OWL	Java	passive	Supported
2009	SWOBE [89]	OWL	Java	Active/Dynamic	Limited
2011	Sapphire [90]	OWL	Java	Active/Dynamic	Limited
2011	Paar et al.[91]	OWL	Zhi#	Active/Dynamic	supported
2011	KOMMA [92]	RDF	Java	passive	–
2014	LITEQ [93]	RDF(s)	F#	Active/Dynamic	Limited
2017	Owlready [94]	OWL	Python	Active/Dynamic	Indirect
2017	Leinberger et al.[95]	OWL	λ DL	Active/Dynamic	Supported

challenging as there are no constrains imposed by the target programming language on the kind of data structures used to encapsulate the concepts of the source ontology. On the other hand, it forces the developer to write verbose lengthy code even for simple programming tasks. Moreover, it does not provide any kind of static typing or error checking. Under this approach we classify the well-established Jena and OWL API as well as the more recent Owlet system. The following paragraphs briefly describe each of them.

Jena [80] is a Java-based and RDF-centric system; it can read RDF serializations of an ontology to produce an in-memory RDF graph representation. It is then possible to use Jena's rules-based reasoner to perform entailment reasoning on the RDF graphs. If we

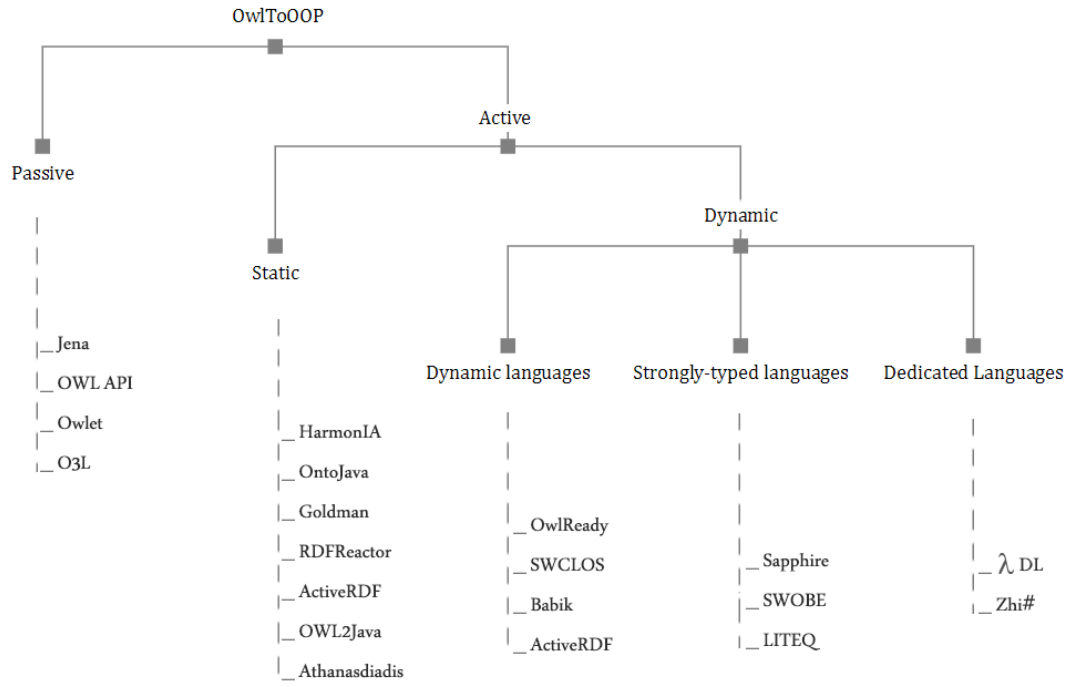


Figure 4.1 – A taxonomy of existing OWL-to-OOP mapping tools and approaches.

put the problem of verbose code and lack of compile-time error checking aside, Jena offers the most complete, and flexible, system for semantic application development. Beside parsing ontology languages, it provides support for querying, database persistence as well as some visualization tools.

OWL API [79] is OWL-centric and, compared to Jena, is less RDF-friendly. It also reads the serialized form of an ontology to produce an in-memory representation but instead of RDF graphs, it produces Abstract Syntax Trees (ASTs) as the output of the ontology loading routine. Protégé API can be seen as an extension of OWL API. The OWL API code is open source and is available under either the LGPL or Apache Licenses.

Owlet [86] is an object-oriented software system for building and managing OWL ontologies in a heterogeneous distributed environment. The author describes Owlet as having a capability-aware load balancing feature where reasoning tasks are assigned to differ-

ent nodes based on their computation power but no details on how this feature is realized are provided. From the same author we also have the O3L framework [88] that is derived from Owlet. In both, the ontology is mapped to a set of instances of generic and predefined Java classes.

4.3.2 Active OWL-to-OOP mapping

In contrast to passive ontology mapping, the active mapping approach will transform an ontology from its serializable format into code statements in the target programming language. The resulting ontology is then executable and belongs to the code segment of the allocated memory. This approach is more challenging as it requires finding a native equivalent in the target programming language for each axiom in the source ontology; a requirement that happens to be problematic in many cases as we will demonstrate in the following sections. Beside providing the developer with a more transparent interface into domain objects, active ontology mapping has the advantage of compile-time error checking. The developer can rely on the compiler for detecting type errors that would have otherwise gone undetected until runtime using the generic access provided by the passive approach. Active OWL-to-OOP mapping can itself be further classified into static or dynamic.

Static mapping

In static mapping, the translation (i.e. code generation from OWL concepts, properties and individual axioms) is done in one shot as a separated prior step. In this case and depending on the target language support for dynamic typing, type checking is mostly limited to compile time. Many of the surveyed tools in this article are classified under this approach. We dedicate the following paragraphs for a brief summary of each.

HarmonIA by Kalynapur et al. [75] is one of the oldest but most important attempts at bridging the gap between ontologies and object-oriented languages. While most other approaches merely aimed at providing programmable access to ontologies, the work of Kalynapur et al. [75] attempts at translating OWL DL ontologies into Java classes all while maintaining their semantic profile (except for some cases like OWL inverse functional properties and "`rdfs : sameAs`" axioms). This is achieved by 1) utilizing interfaces with shadow classes for multiple inheritance, 2) custom-tailored design patterns for expressing OWL logical axioms and 3) special listeners on property accessors to enforce property restrictions. As for reasoning support, ABox reasoning was left for future work. Still, the mechanism in which the property listeners check for property changes and throw relevant exceptions can be seen as a limited form of reasoning. While this approach was shown to maintain the semantics of the source ontology, we believe that it suffers from the side effect of being overly restrictive in some cases. HarmonIA does not seem to be maintained at present day but many subsequent research and software tools are based directly on its ideas. Among these are the Sapphire tool covered in this survey as well as Jastor [2]: An open source tool for providing type-safe, ontology-driven RDF access from Java.

ActiveRDF [84] uses stateless transparent proxy objects to represent and manipulate RDF resources in the Ruby programming environment. The choice of Ruby is motivated by the dynamic features it offers. The authors argue that in order to resolve the impedance mismatch between RDF and OOP paradigms, one needs to remove some restrictions in the object-oriented language. Moreover, the mismatch between RDF and the object-oriented paradigm is smaller in scripting object-oriented languages than with the standard compiled languages. ActiveRDF makes use of dynamic typing, runtime introspection and Ruby meta-programming capabilities but the mapping is still done statically prior to runtime and hence is the classification of ActiveRDF under this category.

2. http://jastor.sourceforge.net/

RDFReactor aims at delegating the tasks of insuring type safety and providing auto completion to the Java compiler and the IDE programming environment. Its first working prototype was presented in a paper by Volkel et al. [82]. The prototype depicts a two-step mapping approach: An ontology is first mapped into an intermediate representation, a JModel, that provides a one-to-one mapping to the source ontlology. The JModel is still more expressive than ordinary Java constructs and may pose problems when generating code. Some algorithms are thus applied to make it stricter for "proper" Java code generation. Obviously, the applied transformations come at a the price of lost semantics. For example, multiple inheritance is avoided by simply considering the super class that has itself the most super classes as the only parent. According to the authors, this strategy entails no loss in functionality when it comes to conventional software development. Finally, queries to the original RDF source are answered by using hand written methods to translate method calls into triple-centric methods.

OWL2Java by Zimmerman[87] is a tool that aims at hiding the peculiarities of working with an ontology in a strongly-typed environment by automatically generating Java wrapper classes for OWL classes previously imported by Jena. The generated code uses Jena underneath; enabling the developer to exploit other Jena features such as SPARQL queries. To help overcome incomplete specifications in the source ontology such as missing domain and range axioms, OWL2Java provides the developer with some "structure-finding tools". These tools employ some kind of entailment to deduce missing information but they are still not regarded as reasoning tools in the formal sense of OWL reasoning as neither completeness nor soundness of the induced information is guaranteed.

OntoJava [74] maps ontologies written in basic RDF, RDFS or DAML+OIL into Java code. Even though RDFS is less expressive than OWL, it is still more expressive than Java and handling issues like multiple inheritance in the RDFS source is considered problematic. On the other hand, unlike OWL, the RDFS specification resembles object-oriented

languages in allowing a property to have multiple domains but only one single range, This can greatly facilitate the task of translating RDFS properties into Java in comparison to translating OWL multiple-range properties. It can also detect RDFS multiple-range modeling errors as the Java compiler automatically enforces this constraint. Along with OntoJava, we have its sister tool OntoSQL that translates rules in RuleML [3] into SQL in order to make them available for top-down evaluation by an SQL engine.

Goldman 's approach [57] is concerned with automatically generating software class libraries for ontology-specific application development in the .Net programming environment. It relies on the .Net abstract code model for generating assemblies. The proposed automatic mapping exploits .Net generic types. It uses interfaces for multiple inheritance and proxy wrappers for multiple-class object instantiation. As it is the case with the other surveyed tools that are .Net-based, integrating ontologies into .Net enjoys a certain degree of portability thanks to the support provided by the .Net Common Language Runtime (CLR) environment that allows for libraries written in one language to be loaded and used by other .Net languages.

Dynamic mapping

In this subcategory of active OWL-to-OOP mapping, the translation from the ontological source into code statements in the target programming language is done dynamically at runtime. Beside the challenge of lack of expressiveness of programming languages that is present in active mapping approaches, this approach is even more challenging to implement as it requires a certain degree of flexibility for performing inference tasks on the ontology in its executable form, e.g. entailing the class of an individual and assigning its type at runtime. Under this approach we can find many tools proposed in the literature and they all provide different degrees of reasoning support. Here again we differentiate

3. RuleML: A unifying system of families of languages for Web rules. More under: http://wiki.ruleml.org

between:

1) Tools that have a dynamic language as output and will rely on its dynamic typing features. Under this branch we have: SWCLOS [81], Owlready [94] and the approach proposed by Babik and Hluchy [83].

2) Tools that have a strongly-typed language as output but can still offer some degree of flexibility for type changes at runtime. The Sapphire tool [90] is a good example for this group.

3) And finally, some of the surveyed approaches have gone to the extent of proposing a dedicated programming language; like for example the Go! language [77] or the more recent λ DL language [95].

Owlready [94] is intended as a solution to answer specific practical needs of software projects in the biomedical domain; mainly the need to treat the ontology classes as objects and to support local closed-world reasoning [97]. To address the first need, Owlready relies on Python meta-classes to map ontological entities into Python objects. Owl classes are then treated as instances of Python meta-classes. Owlready adapts ontologies to the Python object model using "special methods" that are defined at metaclass level in case of mapping OWL classes or at class level if they are to be applied on instances. One example of these methods is the "Close_world()" method that can be called at different levels of the type tree to provide the necessary missing axioms to answer queries under a locally closed-world assumption. Automatic classification of classes and individuals can be performed by connecting to the HermiT reasoner[98] as an external reasoning component. Although very recent, Owlready is a promising proposition given the growing community of Python developers working on building software for the biomedical and bioinformatic domains.

LITEQ [93] stands for Language Integrated Types, Extensions and Queries. It represents a paradigm for integrating RDF triples into the context of F# programming environment. LITEQ provides a mechanism for querying the RDF data source then mapping and strongly typing the imported triples as variables of code types created in F#. Querying RDF sources is achieved using a special variable-free language that is called Node Path Query Language (NPQL). NPQL is syntactic sugar for querying RDF graphs for developers who are less familiar with SPARQL. NPQL allows for RDF graph traversal via three main operators: subtype navigation, property navigation and property restriction operators. NPQL makes no distinction between schema and data, both can be explored using query expression composed of these three operators. LITEQ is supported by Microsoft and is supposed to be packed under the FSharp.Data .Net library. It is also open source and available on Github [4] but the project seems to be frozen for a few years now.

λ DL [95] is a hybrid language resulting from integrating the $\mathcal{ALCOCI}$ fragment of Description Logics into λ-calculus. It can be considered as a dedicated type system for semantic data that treats concept expressions as types. Reasoning is supported by providing a query mechanism based on Description Logics. In the design principles of λ DL, accent is made on type-safe usage of queries meaning that query results must be properly typed. The authors provide a formal proof of soundness in the paper all while acknowledging certain limitations at the current stage of their work. The λ DL language is an elegant way of insuring type-checking over semantic data. However, it presumes that the developer is already familiar with at least lambda calculus or Description Logics and from the perspective of conventional software development, this presumption puts into question the usability of the proposed language.

Sapphire [90] builds on the work of Kalynapur et al. (HarmonIA) [75] and Zimmerman (Owl2Java) [87] as established techniques for mapping OWL classes to Java interfaces but

4. https://github.com/Institute-Web-Science-and-Technologies/Liteq

it differs from them in relying on dynamic proxies to provide a runtime approximate mapping for OWL type system. Each time an OWL type is added or removed, a corresponding wrapping or unwrapping operation is performed on the proxy object. Sapphire provides an option to support open-world logic within Java by replacing boolean values by a three-valued enumeration. A limited form of reasoning is realized using methods that interpret OWL axioms such as `owl : sameAs` or `owl : DifferentFrom`. These methods belong to the top interface (`Thing`) and are propagated into all extending interfaces. OWL entailment rules are not translated into production code but stored in an underlying quad store where a reasoner can be applied. The argument the authors provide for utilizing such a hybrid approach even if it imposes a certain performance overhead is that it keeps production code isolated from classifications that do not cleanly map into Java.

SWCLOS [81] is a semantic web processor built on top of Common Lisp Object System (CLOS). CLOS shares the same class semantics with most OOP languages but the mechanism of its class-subclass and class-instance relations is unique in the sense that they are based on transitivity and subsumption relations just like in RDFS. This resemblance between CLOS and RDFS semantics explains the author motivation for using CLOS. SWCLOS maps RDF(S) and OWL classes into meta-objects in CLOS. It uses dynamic programming, by means of meta-classes, to modify the behavior of meta-objects at runtime. Reasoning in SWCLOS is supported by implementing an extended version of the structural subsumption algorithm [4]. Like its original version, the extended algorithm is incomplete but is shown to be useful for most practical cases of OWL reasoning.

Babik et al. [83] proposed an integration approach that aims at making the most out of the dynamic features of Python; mainly python metaclasses, i.e. classes of classes, to reflect the set-theoretic semantics of OWL while preserving the classical object notations in object-oriented languages. The instance of a metaclass is itself a class and it represents the mapping of the OWL class to the intensional set. Babik proposes an interface to a

Java-based OWL reasoner. Reasoning under open-world semantics is not supported in this approach due to the binary nature of Python boolean values. The use of a Java-based reasoner in Python and the need to convert between virtual machines impose a certain performance penalty.

Athanasiadis et al. [85] propose mapping ontologies into a three-layered semantic-rich development architecture that uses conventional software development tools such as Java Beans and Hibernate. In this architecture, OWL models are used to express rich semantics and to connect to an external reasoner for logical inferences, Java Beans are used for end-user application development while Hibernate object-relational mapping are used for content persistence. To overcome the expressiveness gap between OWL and Java, the proposed mapping only considers non-anonymous OWL classes and ignores other OWL classes such as restrictions and union classes as they do not add more attributive specifications to the relational model.

Clark et al. [77] have introduced ontology-oriented programming in Go! Their work does not constitute an integration approach but rather a new custom-made language to represent ontological knowledge in a more general-purpose programming environment. The motivation the authors present for creating a whole new language is "cleanly integrating logic, functional, object-oriented and imperative programming styles", all in a multi-threaded agents environment. Knowledge in Go! is represented as "sets of labeled theories incrementally constructed using multiple-inheritance". The expressiveness of Go! is compared to that of OWL Lite but in general other OWL profiles are all more declarative than Go!. Go! was first introduced in 2006 and had good potentials as a general-purpose programming language that uses declarative style for functions and relations and imperative style for procedural actions. Reviewing the literature, however, we could not assert that the language has finally gained momentum.

SWOBE [89] proposes embedding semantic web languages: RDF/XML as data language, SPARQL as query language and SPARUL [5] as update language into Java to provide compile-time type checking and detect syntactic and semantic errors earlier. This approach results in a heterogeneous Java code snippets that the authors call SWOBE programs. The developer would write a SWOBE program containing a query in SPARQL and the SWOBE precompiler then checks -via a separate static program analysis step- and reports any syntactic or semantic errors detected at compile time. The precompiler finally transforms the SWOBE programe along with the embedded languages into standard Java classes: main, iterator and helper classes.

Zhi# [91] is a programming language that integrates OWL and Xml Schema Definition XSD into C#. Zhi# programs are inter-operable with .NET assemblies because they are compiled into C#. In fact, Zhi# is itself a superset of C# version 1.0. The syntactical extensions take the form of external types that can be imported using non-recursive import directives. This approach does not rely on code generation but rather on semantic-aware access to the underlying ontology management system. In order to support OWL subsumption, Zhi# extends the explicit subtyping system of OOP languages to permit value space-based subtyping where a type is a subtype of another type if its value-space (set of given values) is a subset of the value-space of the other type. The main difference Zhi# brings compared to other active dynamic approaches with a dedicated language is its "pay-as-you-go" compile strategy. When it comes to external OWL types, the programmer has access to all types in the source ontology but only the types eventually consumed by the developer will be mapped into C# code during compilation. On the other hand, runtime dynamic checking is required each time an instance of an external type is used and this could, at least marginally, penalize runtime performance. Finally, the issue of providing the developer with a transparent access to the ontology is only partially solved as the .Net developer still has to go through a certain syntactic twist to

5. SPARUL: SPARQL Update Language. https://www.w3.org/wiki/SparqlUpdateLanguage

access the underlying source via Zhi# code.

4.3.3 Summary

In this section, we iterated through the approaches that were retained within the OWL-to-OOP scope defined earlier. The goal was not to go through all the technical details of the implementations but to briefly summarize what defines each of these approaches and what differentiates them from one another. Yet, the resulting summary was somewhat overwhelming. To better encapsulate our review, we grouped the surveyed approaches into a taxonomy of hierarchical categories that we illustrated in Figure 4.1.

Zooming out from this hierarchical classification, one can observe some form of an impediment, a semantic gap, that is present in most of these approaches regardless of their variations. In the following section, we try to look at this semantic gap a bit more in details.

4.4 Semantic gap

The most prominent challenge that is present in OWL-to-OOP mapping, particularly in active mapping approaches, is the semantic gap between the ontological and object-oriented paradigms. This gap is most prominently present when considering the different assumptions on which modeling in the two paradigms is based; or in the cases where formal programming languages lack the expressive means to represent certain ontological constructs.

4.4.1 Expressiveness and different interpretations

The semantic richness of ontological languages makes it very difficult to find an OOP counterpart for each OWL semantic construct. One of the most obvious examples is perhaps the different interpretation of class inheritance. OWL, or Description Logics in general, has a looser interpretation of a class being the subclass of another. In OWL, the term `rdfs : subclassOf` is the manifestation of the subsumption operator of DL. An OWL class is allowed to have many parent classes (named or anonymous) as long as it is subsumed by all these parents. On the other hand, pure OOP languages like Java or C# have a stricter definition of class inheritance. OOP classes are disjoint by design and a class cannot be a subclass of two different disjoint parent classes and consequently multiple inheritance is, generally speaking, not supported. Multiple inheritance is not the only example of the missing semantic equivalence. A similar argument goes for other OWL axioms such as `owl : equivalentClass`, `owl : sameAs` or `owl : disjointWith`. In native Java or C# semantics, there is no equivalent options to express these axioms.

Many of the approaches we surveyed did not attempt at bridging this gap and have instead limited the mapping scope to what is expressible in the target programming languages. On the other hand, some of the active approaches proposed interesting solutions ranging from sophisticated approaches of stretching the expressiveness of modeling in Java to that of OWL DL by enforcing some constraints and design patterns like in HarmonIA [75] to extending the subtyping system of the host programming language to support Description Logics subsumption as in Zhi#.

4.4.2 The Open World Assumption

Most existing ontologies in the semantic web are OWL DL ontologies. I.e. they are based on the Open World Assumption (OWA) of Description Logics [4]. The OWA argues that since it is not possible for an agent to have complete knowledge, then it is not possible, via

deductive reasoning, for this agent to infer the truth value of a fact that is not, explicitly or implicitly, present in its base; irrespective of whether or not it is known to be true.

The OWA concerns reasoning rather than modeling; yet it has great impact on how models are written in the first place. OWA makes perfect sense in the context of Description Logics and other logic programming languages but it is still rather counter-intuitive and a major source of confusion for most conventional software developers; especially when contrasted to the Closed World Assumption (CWA) present in other common modeling paradigms where the absence of a statement in the database allows the agent to infer its falsity.

Earlier in the survey, we have seen that Owlready proposed closing the world locally by providing missing axioms at different levels of the type tree, or globally if provided starting from the root node. There exist many other calls in the literature for supporting closed-world ontological reasoning but covering them more in details falls outside the scope of this thesis. We forward interested readers into the work done in [99] [100] and [101].

4.4.3 Unique Name Assumption

Another basic assumption that can make a fundamental difference between modeling in object-oriented and ontological languages is the Unique Name Assumption (UNA). This assumption is a simplifying one that requires different names to always refer to different entities. It is the default assumption when it comes to programming languages: Different class names yield different classes and different variable names yield different variables. In ontological languages, however, this assumption is not made. Two OWL classes or individuals with different unrelated names may very well refer to the same entity. This difference explains the presence - or absence in OOP languages - of explicit OWL axioms (`owl:equivalentClass`, `owl:sameAs` or `owl:disjointWith`) to express the uniqueness of an entity.

4.5 Discussion

Some of the main reasons behind most of the work surveyed in this chapter were 1) the difficulty of manipulating ontologies in mainstream software development and 2) the scarcity of options for an ontology programming interface that provides static typing and error checking. Nevertheless, as we can see from earlier sections, a retrospective scan on work done in this area revealed a different story. Beside the tools and approaches already covered, there exist several other projects around the same topic: OntologyBeanGenerator [6], Alibaba [7], KOMMA [92], API a gogo [102] and others. Some of these tools are listed in Table 4.1, but we abstain from covering them in more details as the techniques they utilize largely intersect with those of the other tools covered already.

In other words, there exist already many options both for accessing and integrating ontologies in general-purpose programming paradigms, yet we still did not witness ontologies spanning new development territories and the question about the feasibility of mapping ontologies into the landscape of conventional software development remains open despite all the proposed integration approaches. Still seeking some answers, we tried to look beyond the technical aspects proposed in the literature into a more holistic view that takes into account the rational aspects of the integration. Below, we discuss some of the potential reasons we believe they contribute to the developers' shy adoption of ontologies:

Too many options: The real problem developers may have with integrating ontologies is not necessarily the lack of ontology programming interfaces - there exist well many - but rather the lack of consensus on a standardized option. Unlike the case of ontological modeling where OWL is "the language" and Stanford protégé is "the editor"; when it comes to integrating ontologies as conventional software models, there exist many options but none of them has reached a good level of maturity to gain community consensus. As

6. OntologyBeanGenerator Protégé plug-in: https://protegewiki.stanford.edu/wiki/OntologyBeanGenerator
7. Alibaba: https://bitbucket.org/openrdf/alibaba

a result, the developer has to go through the hassle of sorting them out before being able to judge on the pertinence of any of these options; a task that is cumbersome and not even affordable in most of today's agile software projects.

Paradigm shift: Although largely addressed in the literature, the paradigm shift the developer has to go through when integrating ontologies is still present. Providing tool support is one thing, but it takes much more to overcome the conceptual switch behind ontological modeling. Translating ontologies into a program does not change the fact that ontological modeling is explicit and is most of the time based on an open-world assumption in contrast to implicit closed-world modeling in UML and OOP languages. The same kind of argument also applies for the approaches that went as far as proposing dedicated languages for ontology-oriented programming. While we are sure these languages can easily target an audience with specific fine-tuned profiles, we are less sure whether such propositions would answer the question of democratizing semantic application development or if it would instead lead the developer into a "yet another language" syndrome.

Legacy projects: From a pure practical point of view, introducing ontologies to mainstream software projects is especially challenging when there is some legacy code to maintain and respect; which is the case of the majority of software projects in large scale organizations.

Resistance to change: Finally, for most "right-wing" software engineers, adopting ontological approaches, or generally speaking linked-data approaches from the semantic web, means, in a way or another, shifting towards a more volatile domain model. A move that may not be perceived as a positive step in the circles of software engineers with strong preferences for tidy and well-engineered domain models. It provokes a lot of philosophical discussions similar to the dynamic vs. static-style of coding in languages that permits the two possibilities. Eventually, such a change may very well be welcome, but just when the

right time comes.

4.6 Conclusion and outlook

We started this chapter with the objective of assessing the state-of-the-art of the integration of ontologies into mainstream software development. For this reason, we performed a systematic literature review for existing mapping frameworks and tools between OWL ontologies and general-purpose programming languages. This allowed us to construct a classification of the surveyed tools based on their mapping approaches and the characteristics of their resulting artifacts. We highlighted some of the common challenges encountered in the literature before finally providing our own reflection on why software engineers are still reluctant to incorporate ontologies into their code repositories. Unfortunately, most of the tools and prototypes, especially the early ones, did not yield a concrete body of use cases in the software industry. In fact, when trying to further trace the surveyed tools, we were unfortunate to find out that the vast majority have went idle for years.

Despite the discouraging results, having collected many lessons learned while surveying the literature, we believe we are now in a good position to propose our modest delta to the state-of-the-art.

Chapter 5

The Translation Function

Colorless green ideas sleep furiously.

Noam Chomsky,

Syntactic Structures 1957

5.1 Overview

The main objective we are concerned with in this thesis is democratizing the use of ontologies in conventional software development. To better assess the kind of alternatives we have in order to achieve this objective, we started the journey with a systematic scan of work done in that area. Despite a decent number of research articles on the topic, it was difficult to trace a concrete practical use for the majority of the proposed tools. In the previous chapter, we have discussed some of the reasons we believe are still standing between research efforts as depicted in literature and a successful democratization of ontologies. Trying to summarize the lessons we have learned we would say that a successful democratization of ontologies into a wider audience of software development communities is dependent upon minimizing the twist developers have to go through when adopting ontological modeling standards. The twist we are trying to minimize here is manifold:

syntactic, semantic and paradigmatic. While the paradigmatic aspects of the twist are intrinsic to the nature of the two different camps, the other syntactic and semantic aspects can still be minimized to a great extent. Though not explicitly categorized under this objective, some of approaches we have seen in the literature attempted at minimizing the twist by providing the developer with a transparent programmable access into approximations of ontologies.

Building on top of these attempts, the work we present in this chapter and the following one contributes to the state-of-the-art by providing a feasible implementation of a semantic-loss translation of ontologies from their passive form expressed in OWL into an active form directly expressible in a mainstream programming language that is already well-established in software engineering communities. To put things into perspective compared to the tools and frameworks surveyed in the last chapter, our work belongs under the active-dynamic mapping branch of the taxonomy. The following sections will further define our approach and justify some of the design decisions related to the proposed translation module. A more technical view on the latter is deferred to the next chapter.

5.2 Translating between the two representation

The task of translating OWL ontologies into an executable form boils down to defining a translation function that will map pairs of possible axiom/semantics constructs into their corresponding pairs of code statement/semantics in the target programming language. The executable form of the ontology is then equivalent to the closure of applying the mapping function on all axioms in the source ontology. Before discussing the details of the translation, it is important to define the stance taken in this thesis towards the translation problem. Theoretically speaking and given that the majority of computer languages are Turing complete, the problem of translating from one computer language to another is already solved. This is the assumption on which compilers, assemblers and

transpilers [1] are built. However, in the case of translating between ontologies and general-purpose programming languages, we are less inclined to adopt the same view for different reasons. First of all, OWL DL, or any other fragment of FOL used in this context, is an annotation language that is more suited for knowledge representation tasks and is not a computational method per se. Without some rules extension that provides a sort of conditional branching, such a language is not Turing complete. Further more, Turing completeness alone does not tell much about the equivalence of two languages. There is more to language design than that. If two languages are Turing complete, all what this means is that any problem that is computable using one of these two languages is also computable using the other, as well as using any other Turing complete language; but this tells nothing about the language features or design choices or what kind of programming styles are supported by the language. In other words, pure "grammatical" translations cannot guarantee readability nor usability of translated programs. That is why we tried to adopt a more holistic approach when tackling this problem by proposing a translation that respects the paradigmatic aspects of the source as well as the target language.

5.2.1 The translation function

Given a computer language $\mathbb{L}$, let A denote the set of all its possible syntactic constructs and let S denote the set of all possible associated semantics. The language $\mathbb{L} \subseteq A \times S$ can be denoted as:

$$\mathbb{L} = \{(a, s) \in A \times S \mid s \text{ is the semantic associated with } a\}$$

In our case, the ontological source language $\mathbb{L}_s$, can be denoted as:

$$\mathbb{L}_s = \{(a, s_s) \in A_s \times S_s \mid s_s \text{ is the ontological semantic of } \ a\}$$

1. Transpilers are computer programs that translate programs written in a high level programming language into another, also a high level one.

And the target general-purpose programming language $\mathbb{L}_t$ denoted as:

$$\mathbb{L}_t = \{(c, s_t) \in C_t \times S_t \mid c \text{ is the code construct with semantic } \ s_t\}$$

With $\mathbb{L}_s$ as the domain and $\mathbb{L}_t$ as the codomain, the translation function $\mathcal{T}$ is given as:

$$\mathcal{T} : \mathbb{L}_s \rightarrow \mathbb{L}_t : \mathcal{T}(a, s_s) = (c, s_t)$$

The above formulation depicts the translation as a simple mapping function $\mathcal{T}$ that can be implemented in a straightforward manner with the help of a look-up table of mapping rules. Two conditions need to be respected though:

1. $\mathcal{T}$ must be a total function with a known finite domain. In other words, the following statement must always hold:

$$\forall\ (a, s_s) \in \mathbb{L}_s \implies \exists\ (c, s_t) \in \mathbb{L}_t : \mathcal{T}(a, s_s) = (c, s_t)$$

2. Maintaining the semantics means the function $\mathcal{T}$ will only map pairs of the source language to pairs that have the same semantics in the target language:

$$\forall\ (a, s_s) \in \mathbb{L}_s\ \forall\ (c, s_t) \in \mathbb{L}_t \mid \mathcal{T}(a, s_s) = (c, s_t) \implies \ s_s \equiv s_t$$

Respecting the total function condition in isolation from the other semantic equivalence condition is not a problem. Knowing that an ontology $\mathcal{O}$ is a finite set of axioms that are constructed using a predefined limited number of constructors, it is easy to show that the domain of the function $\mathcal{T}$ is finite and known given $\mathcal{O}$. The problem arises when the two conditions need to hold together. In the case of an executable target language, it has been shown that there are no equivalent for certain ontological semantics, i.e. $S_s \cap S_t \subsetneq S_s$. To respect the semantic equivalence condition one can limit the application of the translation function to the pairs that have equivalent semantics in the target $\mathbb{L}_t$ but that

already violates the first total function condition as some pairs in $\mathbb{L}_s$ will be left out unmapped.

The description above is just another formulation of the semantic gap problem faced by some of the active mapping approaches discussed in the previous chapter. In the following sections, we will illustrate by the help of some examples this semantic mismatch. We will briefly revisit one of the approaches taken in the literature and then will move on to evaluate the different alternatives we have considered in order to overcome this limitation before finally concluding with the approach proposed in this thesis.

5.3 Semantic extensions

Although mapping OWL axioms to their programming counterparts seems self-evident in some cases like for example mapping `owl:class` to an OOP class, `rdfs:subClassOf` to an OOP class inheritance relation and OWL individuals to instances of OOP classes; finding the right mapping becomes more problematic when we consider OWL DL terms such as: `owl:disjointWith`, `owl:sameAs` and `owl:equivalentClass`. There are no counterparts in a general-purpose programming language that have the same semantics. If a losseless translation of ontologies into executables is what we are after, then this is a clear call for semantic extensions of the target programming language.

As discussed in the previous chapter, one possibility to provide such an extension is to create a syntactic language extension along with the necessary associated semantics. This is the approach taken by Leinberger et al. in λ DL [95] and Paar et al. in Zhi# [91] and is arguably one of the cleanest approaches in the sense that these extensions can be considered as domain specific languages tailored for the specific problem in hand. In the context of democratizing the use of ontologies, this approach suffers from a serious shortcoming. Instead of facilitating access into ontologies through the use of a pervasive language as means of representation, by introducing yet another language for the new

representation, we are only augmenting the learning curve for the developer who has now to learn the syntax of the new language as well as the paradigmatic aspects behind ontological modeling.

This problematic aspect of the syntactic extensions approach brings us to other possibility of providing semantic-only language extensions. Clearly enough, the motivation behind such extensions is minimizing the twist by avoiding the learning curve associated with the new syntax. Here again, different possibilities can be considered:

1. Design patterns: Simply put, software design patterns are repeatable solutions to recurrent software engineering problems. The simplest example is probably the singleton pattern: a pattern that is used to insure only one instance of a certain class exists during the whole lifetime of the application. To force the singleton constraint, the developer would hide the constructor of the class by making it private and would instead provide a public static `GetInstance()` method to return the current instance. It is then up to the `GetInstance()` method to decide if a new call to the private constructor is necessary or not. Design patterns vary in function of the problem they address. Similarly, in the case of ontological modeling, one can argue that the use of certain domain-specific patterns, created for this particular reason, can help force some constraints that are usually present in ontologies but not explicitly expressible using the native formal constructs of the programming language. In the paper by Kalynapur et al. [75] that we covered in our survey in the last chapter, this approach has been given a shot. The paper describes the use of some proposed new design patterns in a framework to generate Java classes from OWL ontologies. The work is well elaborated and merits investing more time to build on top of the proposed approach. After some attempts, however, many reasons led us to refrain from going that route. Design patterns, though useful in certain situations, can be tricky to get right and many of them are already tagged as anti-patterns [2], let alone that the proposed patterns are overly restrictive in

2. Anti patterns are design patterns that are overly used but are not needed in the first place.

many cases. Here we give as an example using property listeners in Java to enforce type information in domain and range axioms as proposed in the paper. This pattern works fine when multiple type information is provided but only one was actually intended. In ontological modeling, however, multiple type information for the range of a property is very often intended to be interpreted under the open-world assumption to allow the possibility for further deductive inference. The compile errors caused by enforcing the design patterns are simply irrelevant in this case. Furthermore, even when tolerating this kind of opposing interpretations, the scalability of such an approach is already at stake since it results in an unnecessarily complicated code that is hard to understand and to maintain.

2. Attributes, or annotation in Java, are a method of associating metadata and declarative information with code types, properties, methods or whole assemblies. In combination with the metaprogramming technique known as reflection, these attributes can be queried at runtime. Attributes can take arguments and one can associate code constructs with one or more attributes. As declarative pieces of meta information, attributes can be used to express OWL terms that are foreign to a general-purpose programming language. This could make for a reasonably good semantic extension. Indeed, we could have taken this approach for realizing the semantics extension we are looking for but given the amount of OWL axioms that need to be expressed as attributes and given that attributes are actually external to code type definitions, this would result in most of the modeled information about an object being laid outside of its type definition and not directly accessible via its properties.

3. The third alternative and the one finally adopted in our work is the simplest of all. The idea is to bootstrap the translation process with a meta layer of code that will form the base for any ontology in its executable form. More details on this meta-layer semantic extension are provided in the following section.

5.3.1 Meta-layer semantic extension

The meta-layer semantic extension is nothing but the top layer in the type hierarchy of the executable ontology. Like in the passive form of OWL ontologies where the top concept is represented as the `Thing` concept, the top class in the executable form is a class named `Thing`. OWL axioms related to concept definitions are represented as code properties of the class `Thing`. The same goes for OWL properties; a top class called `OWLProperty` is used to root the taxonomy of all translated OWL properties and the constrains axioms related to property restrictions are also represented as code properties of the class `OWLProperty`. Since the code properties in the top `Thing` and `OWLProperty` classes are used to augment the programming language vocabulary used to describe type information, these properties are called meta-properties. Meta-properties corresponding to terminological axioms are static (i.e. shared between all instances) whereas assertional meta-properties are non-static.

In the executable form, the translation will always start with creating the class `Thing` as the parent of all subsequent classes with meta-properties defined in the top hierarchy level and then inherited by all translated classes afterwards and masked where necessary.

The idea is that meta-properties would only cover up for the missing explicit semantics in the formal language constructs. Consequently, in cases where a native semantic construct is available in the target language, the native construct is used. The set of meta-properties is then limited to $S_{meta} = S_s \setminus S_t$ which is the minimum set needed in order insure the semantic equivalence condition discussed earlier in this chapter:

$$\forall\ s_s \in S_s \implies \exists\ s_t \in (S_{meta} \cup S_t) \ \text{ s.t. } \ s_s \equiv s_t$$

Figure 5.1 depicts the meta-layer semantic extension used to bootstrap the translation process whereas the mapping rules used in the translation are listed table 5.1.

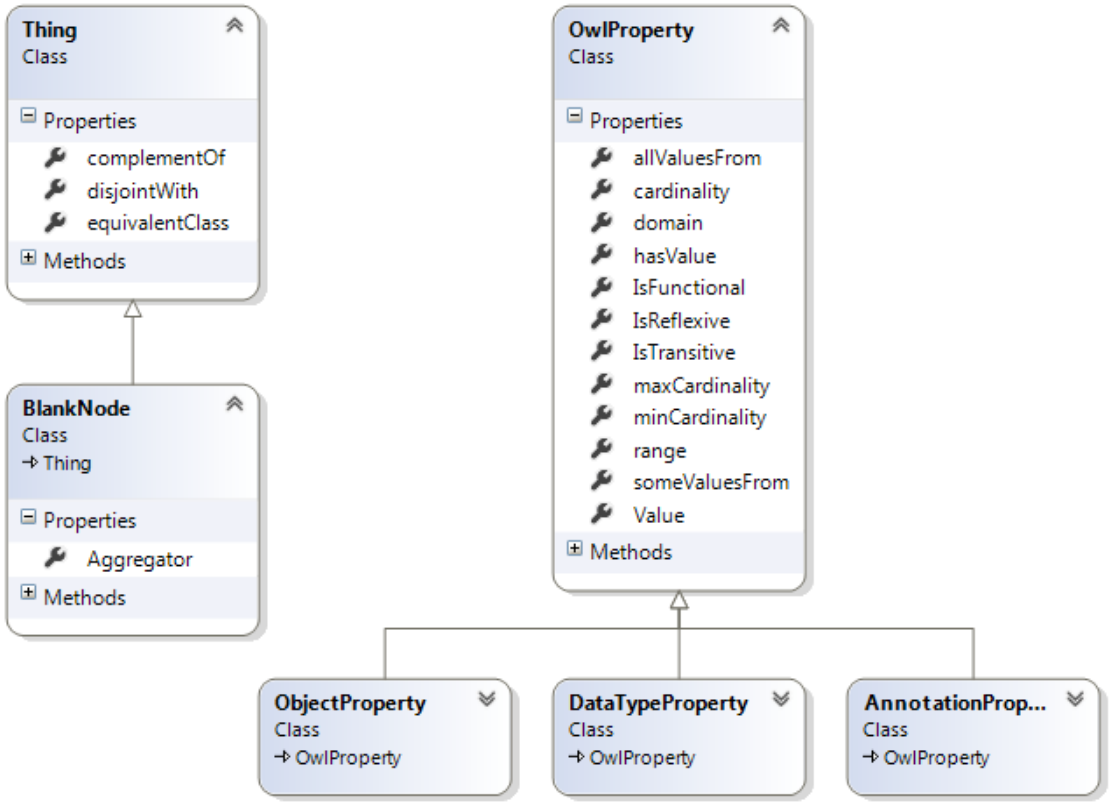


Figure 5.1 – The hierarchy of the meta-layer semantic extension used to bootstrap the translation of OWL ontologies into an executable form.

5.4 Summary and outlook

This chapter provided an abstract overview of the problem of translating OWL ontologies into an executable representation. Motivated by the need to minimize the manifold twist between the source and the target representations, we proposed a translation function that aims at maintaining the paradigmatic aspects of the source representation. The proposed translation function takes the form of a total forward mapping function. The chapter concluded with a summary of the function's look-up table including the meta-layer semantic extension used to bridge the semantic gap in the target representation. A more technical view on the implementation of this function follows in the next chapter.

Table 5.1 – A mapping summary of OWL DL axioms and their C# counterparts

Axiom	OWL	OntoJIT Counterpart
Ontology	owl:Ontology	Code namespace
Class	owl:class	C# class
	rdfs:subclass	C# class inheritance
Class Description	rdfs:equivalentClass	Static meta properties
	owl:intersectionOf	
	owl:unionOf	
	owl:complementOf	
	owl:disjointWith	
Individual	Individual	object instance
	owl:AllDifferent	Non-static meta properties
	owl:differentFrom	
	owl:sameAs	
Property	owl:ObjectProperty	C# class
	owl:DataTypeProperty	
	rdfs:subPropertyOf	C# class inheritance
Property Association	rdfs:range	Static meta properties
	rdfs:domain	
Property Restriction	rdfs:cardinality	Static meta properties
	rdfs:hasValue	
	rdfs:someValuesFrom	
	rdfs:allValuesFrom	
Property Description	owl:FunctionalProperty	Static meta properties
	owl:InverseFunctionalProperty	
	owl:SymmetricProperty	
	owl:TransitiveProperty	
Property Relations	owl:inverseOf	Static meta properties
	owl:subPropertyOf	
	owl:equivalentProperty	

Chapter 6

The Translation Module

> Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.
>
> Albert Einstein

6.1 Introduction

The previous chapter served as an abstraction of the problem of translating ontologies into a new executable representation. This chapter will move forward towards a more concrete description of the implementation of the translation function. The chapter starts with an overview of the structure of the translation module. The following sections will then provide more details on the mapping rules and the meta-layer semantic extension followed by a discussion on some of the issues related to parsing OWL serializations and some particularities of the translation process. A summary of the translation results for some selected ontologies is also provided at the end of the chapter.

6.2 Overview of the translation module

The translation module was written entirely in C#. It takes as input ontologies in RDF/XML or OWL/XML formats and emits the ontologies in their excutable form either as C# source files (.csharp) or already compiled executables (.exe or .dll formats). The runtime compilation of the translation results is achieved with the help of the .Net JIT Compiler. The possibility to translate and generate a compiled executable ontology in one shot at runtime inspired our choice for the name of the translation component OntoJIT as an acronym for "Ontologies, Just In Time". Figure 6.1 depicts a high-level overview of the OntoJIT translation module and its main parts. The parser is responsible for reading the XML serialization of the source and deciding on the batch, or chunks, of axioms to be sent to the translator which in turn will consult the mapping tables before forwarding a virtual representation of the corresponding code to be generated to the code generator. The code generator is an encapsulation of the .Net CodedomProvider component [1]. It receives translation results from the translator in batches and attach them into in an in-memory graph representation. When the translation process is over, the code generator can now generate the final C# code out of the in-memory graph. The output is a .csharp file containing the source of the executable form of the ontology. If, on the other hand, an executable form is needed directly at runtime, then one can simply use the Just-In-Time compile unit from the .Net Common Language Runtime (CLR) environment to compile the generated code (or directly the in-memory graph representation) and generate the executable files.

1. https://docs.microsoft.com/en-us/dotnet/api/system.codedom.compiler.codedomprovider?view=netframework-4.7.2

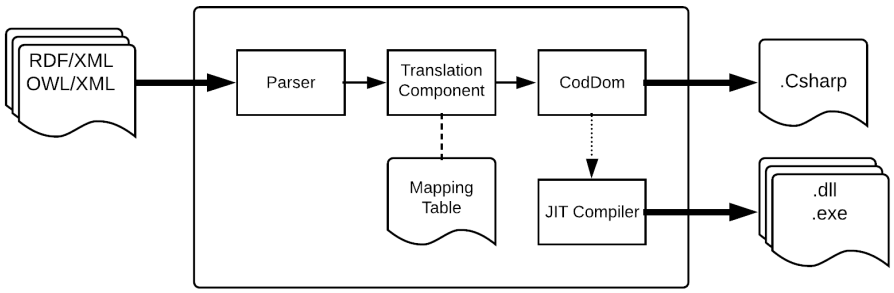


Figure 6.1 – OntoJIT translation component

6.3 Parsing OWL Serializations

6.3.1 Serialization formats

Apart from having many sub-profiles, OWL has a variety of supported syntax formats. These are listed in Table 6.1. In practice, one syntax format is sufficient and the developer usually has the choice of which syntax to work with. For readability, all examples given throughout this thesis are written using the Turtle format. In the implementation, however, we consider two formats: The RDF/XML syntax format since it is a mandatory format for all conformant OWL software and the OWL/XML format since it is also one of the most popular formats for writing and sharing ontologies.

Table 6.1 – OWL supported syntactic formats

Name of Syntax	Status	Purpose
RDF/XML	mandatory	Interchange (read/written by all conformant OWL 2 software)
OWL/XML	optional	Easier to process using XML tools
Functional Syntax	optional	Easier to see the formal structure of ontologies
Manchester Syntax	optional	Easier to read/write DL Ontologies
Turtle	optional	Easier to read/write RDF triples

Reproduced from OWL 2 specification document - W3C OWL Working Group publication: http://www.w3.org/TR/2012/REC-owl2-overview-20121211/

6.3.2 OWL Graph Traversal

Most existing OWL parsing tools use a recursive Depth First Search (DFS) algorithm to perform a one-pass traversal of the OWL source. This seems like an elegant approach for a streaming-like parsing: the DFS serves as a de-serialization technique where for each node visited in the source, a corresponding node is attached to the OWL graph being constructed. The output in this case is then an in-memory graph data structure and the parsing routine has no further constraints to satisfy. In the case when the parsing is followed by the translation to an executable, the pure DFS approach is insufficient. The parser needs to pass all relevant information in one batch to the translator in order for the latter to be able to decide on the corresponding code construct to emit. In OWL XML Serializations this information might be scattered across multiple nodes and the parser needs to collect it from all relevant nodes before passing the control to the translator. To meet this requirement we have two options and a design decision has to be made. The first option is to use multiple-pass traversal together with an intermediate buffer where the parser can keep the information collected until it has all the necessary information to pass to the translator. The second option is to combine pre-order DFS traversal algorithm with look up operations when necessary. This option relies on the inter-node associativity patterns present in OWL source documents. The look-up operations are simply forward jumps within the RDF/XML child nodes where the missing piece of information is supposed to be. Both options have some pros and cons. The multiple-pass DFS traversal option offer a cleaner solution but can become quickly too expensive both in time and space complexity as it requires maintaining the intermediate state of nodes being parsed over many passes. The option of hybrid DFS with look-up operations is more efficient but has the cons of hard coding XML source associativity patterns into the parser which can put the readability and maintainability of the parsing code at a stake. Weighing the pros and cons of the two options, efficiency won over maintainability. We chose to abandon the multiple-pass traversal option and to carefully implement the look-up operation with

the DFS traversal algorithm.

6.3.3 Import Closure

Ontology modeling practices share some of the design principles with software engineering, mostly with regards to the re-usability of existing ontologies. An ontology is not isolated from other ontologies, it builds up on top of other already existing ones. In ordinary programming languages, this corresponds to importing packages or libraries; and in OWL, to using import keyword to allow the usage of terms defined in the imported namespaces. Keeping on with this analogy, the OntoJIT parser treats imported namespaces in OWL source as namespaces in the target output code. When the parser reads an `owl:imports` term, it triggers a recursive call to the main parsing routine for all imported ontologies until an import closure is realized [103].

6.4 Translating Anonymous OWL concepts

In OWL, not all concepts are defined by name. Class expressions can also be used to describe concepts without necessarily assigning the concept a defined name. The concept in this case is defined by placing constraints on its extension. A constraint can be a property restriction axiom, an exhaustive enumeration of individuals, a complement of a class description or finally the union or the intersection of two or more class descriptions. This style of anonymously defining concepts, even though very common for DL modeling, has no counterpart in other data-modeling backgrounds. This again requires finding an alternative when generating the executable form of the ontology.

To further illustrate the use of anonymous class expressions to define concepts, let's consider the simplified example of the following sentence in natural language:

"White wine is a subset of all things that have a white color."

In DL terms this is the same as stating:

$$WhiteWine \sqsubseteq \top \wedge \exists hasColor.White$$

To define this concept in OWL, one would need to use the `rdfs:subClassOf` axiom along with an `owl:Restriction` axiom to define the parent class as an anonymous class. Figure 6.2 shows the corresponding OWL definition for white wine written in Turtle syntax.

```
:WhiteWine
a owl:Class ;
rdfs:subClassOf
  [ a owl:Restriction ;
    owl:onProperty :color ;
    owl:hasValue white^^<http://www.w3.org/2001/XMLSchema#string>
  ] .
```

Figure 6.2 – White wine as a subclass of an anonymous class

Some closely related structures are RDF blank nodes. This type of RDF nodes have a more general sense than anonymous class expressions in OWL but they can still be used to denote anonymous OWL concepts in OWL/RDF and XML/RDF syntax. An RDF blank node can denote any resource without an identifier, a class, an individual or simply an RDF container for other multi-component structures. An example of using RDF blank nodes to represent OWL anonymous concepts is shown in Figure 6.3.

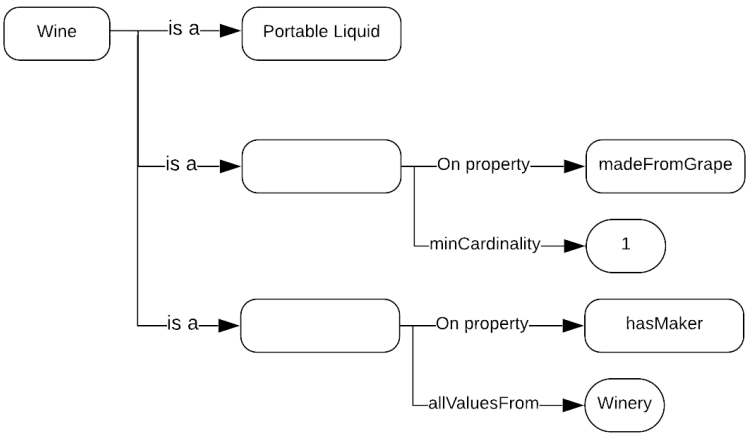


Figure 6.3 – Wine as a subclass of multiple anonymous classes (RDF blank nodes).

Following the mapping rules defined earlier, an OWL concept, named or anonymous, would be translated into a C# class. Given an anonymous OWL concept, the question is whether the automatically generated C# class should also be anonymous or not. C# does allow this possibility but we chose to de-anonymize classes in the executable representation for two reasons: First, the arbitrary use of anonymous classes will unnecessarily complicate the generated code and make it less maintainable without making any difference on the semantic profile of the resulting ontology. The other reason is related to an optimization technique that we will introduce in the next section. In the translation module, we propose materializing anonymous concepts by translating them into ordinary class definitions and assigning them automatically-generated and deterministic names. Naming anonymous blank nodes insures they are available for potential inference tasks over the executable form. On the other hand, since these nodes are not explicitly part of the source ontology named class definitions; these classes get the private access modifier and are therefore invisible from outside the C# namespace they belong to.

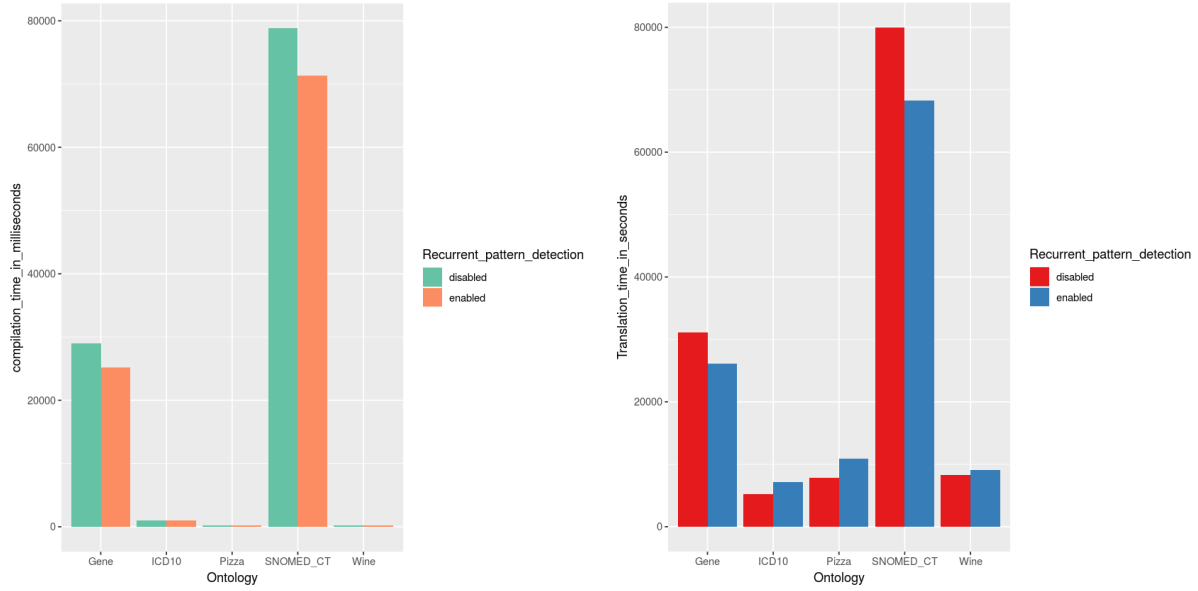
6.4.1 Reuse of recurrent anonymous patterns

In some cases involving the translation of large ontologies, materializing anonymous OWL expressions is necessary in order to enable some optimization techniques that can reduce the translation time as well as the size of the resulting executable ontology. Some medical ontologies, such as SNOMED CT or the gene ontology, can serve as good examples to illustrate the idea. In this category of very large ontologies, certain kind of anonymous concepts are repetitive. It is actually enough to inspect any OWL serialization of the SNOMED CT ontology to see that the anonymous axioms used to express constraints on concepts form clear repetitive patterns. Beside the already very large number of named concepts in SNOMED CT, repeating the same XML coding for the same constraints over and over again results in even larger serialization files that are heavy to load and difficult to manipulate.

In the translation module, materializing anonymous OWL expressions, or RDF blank nodes in general, allows these concepts to have names and to be referred to and reused instead of creating new ones on the fly each time the same set of axioms is needed. The reuse of existing blank nodes can have a positive impact on both the performance of the translator and the size of the resulting executable. The positive impact on the size of the resulting executable is explained by the reduction of redundant anonymous concepts. The impact on the performance of the parsing routine depends on the size of the ontology and the percentage of recurrent patterns. Anonymous expressions can be, and often are, composed in different order and at different levels of granularity. Deciding on reusing an equivalent materialized pattern requires the parsing routine to have the ability to judge whether a certain pattern has already been seen by the parser. For this reason we need to convert the parsed sub-tree of axioms composing the pattern to a canonical (flattened) form in order to be able to compare it with the patterns parsed so far. This, evidently, comes at a price. More precisely a worst-case time complexity of $\mathcal{O}(n^2)$ where n is the number of anonymous expressions in the source ontology. This imposed performance overhead can only pay off once the size of the ontology is large enough so that it contains enough recurrent patterns. In that case, the performance gain resulting from reusing existing classes exceeds the time lost for pattern checking. Figures 6.4a and 6.4b illustrate this effect of performance gain/penalty depending on the source ontology. A somewhat more detailed summary of the translation results and the effect of reusing recurrent patterns is provided in the next section.

6.5 Translation results

To validate the translation results, we applied the translation function to a number of ontologies from various domains. These ontologies were selected to have varying size and profile expressiveness. A summary of the selected ontologies is provided in Tables 6.2 and 6.3. Some minor pre-processing steps were necessary to avoid compilation errors



(a) Compilation time (b) Translation time

Figure 6.4 – The effect of reusing recurrent patterns on time performance.

related to concept names such as names containing white spaces or names starting with a number which are illegal class names in C#. This was particularly the case for ICD10 and SNOMED CT ontologies. Otherwise the source ontologies were directly fed to the translation module with no further modifications.

6.5.1 The selected ontologies

Below we provide a brief summary of the selected ontologies to demonstrate the translation results:

The pizza ontology from Stanford: A rather small ontology that is based on the $\mathcal{SHOIN}$ fragment of DL. $\mathcal{SHOIN}$ is an extension of the $\mathcal{ALC}$ attributive DL language that adds support for role hierarchy $\mathcal{H}$, nominals $\mathcal{O}$, inverse properties $\mathcal{I}$ and unqualified cardinality restriction $\mathcal{N}$. With such an expressive profile, the pizza ontology, though small in size, is very useful for validating the translation results.

Wine ontology: Also an expressive $\mathcal{SHOIN}$ ontology. It is different from the pizza ontology in that it is geared more towards *ABox* modeling. This explains its relatively large numbers of individuals compared to other ontologies.

The Systematized Nomenclature of Medicine – Clinical Terms (SNOMED CT) Ontology is a "standardized, multilingual vocabulary of clinical terminology that is used by physicians and other health care providers for the electronic exchange of clinical health information". The ontology is distributed by the Health Terminology Standards Development Organization (IHTSDO) under Affiliate license [2]. The OWL format of the SNOMED CT ontology is not readily available. Instead, a Perl script is made available in order to generate OWL out of one of SNOMED original formats [3]. To note that for this ontology, the counts provided in Table 6.2 are more prone to minor variations depending on the version of the package distribution and the Perl script conversion errors.

The Gene ontology (GO): A large ontology that has around 43585 terms and 93265 relations with respect to three aspects: molecular functions, cellular components and biological processes [20]. With the addition of full existential quantifiers, the $\mathcal{ALE}$ profile on which the GO is built is slightly more expressive than the attributive language $\mathcal{ALC}$.

The International Classification of Diseases, Version 10 (ICD10) Ontology has the least expressive $\mathcal{ALC}$ profile among the selected ontologies but is one of the most used in practice. The OWL version of ICD10, among other formats, is maintained by the World Health Organization (WHO).

2. https://ihtsdo.freshdesk.com/support/solutions/articles/4000039015-how-do-i-obtain-a-snomed-ct-affiliate-license-

3. SNOMED CT can use more than one URIs language syntax: https://confluence.ihtsdotools.org/display/SLPG/SNOMED+CT+URI+Standard

Table 6.2 – A metric summary of the selected ontologies ordered from most to least expressive.

Ontology	DL Expressiveness	Total No. Axioms	Total No. Concepts	Total No. Roles	Total No. Individuals
Pizza	$\mathcal{SHOIN}$	801	100	20	5
Wine	$\mathcal{SHOIN}$	1046	138	20	206
SNOMED	$\mathcal{ALER}$	1540462	339520	116	0
Gene	$\mathcal{ALE}$	1665862	47125	10	21
ICD10	$\mathcal{ALC}$	97169	14502	2	0

Table 6.3 – Percentage of recurrent anonymous expressions in the serialized OWL format of the selected ontologies.

Ontology	Anonymous expressions	Recurrent anonymous expressions	recurrence percentage
Pizza	242	113	47 %
Wine	206	87	42 %
SNOMED	761662	253888	33 %
Gene	411546	148156	36 %
ICD10	22	21	95 %

6.5.2 Discussion of the results

Different criteria can be used to asses the feasibility of the translation process and the quality of the output results. The semantic equivalence between the source and the target representations is evidently one of them. In the case of executable ontologies, given the meta-layer semantic extension and the complete set of mapping rules as defined by the translation function, the semantic profile of the source representation is maintained in the target executable representation. This is because the translation process for any ontology is bootstrapped with the creation of the meta-code layer that defines the complete-set semantics regardless of whether or not all meta-properties in this set will be used in translating the subsequent concept and role axioms. Furthermore, the empirical results we obtained querying the executable form of one of the most expressive ontologies came out to confirm this equivalence, more details on the conducted experiment and the obtained

results are given in the next chapter.

Another criterion to look at is the readability and maintainability of the resulting code. This criterion is a tricky one. Not only because it is subjective to a certain extent but also because code generated by a tool can seldom meet the consensus of all members in a development team. Furthermore and as discussed at length on previous occasions throughout this work, the source and target representations of the ontology belong to two different modeling paradigms and a successful semantic translation between the two is not a synonym to a successful paradigmatic translation. The paradigmatic twist would still be present; potentially making the code less easier to grasp at first. The overuse of static (i.e. class) variables is probably a good example. In an OOP paradigm, the use of static variables is a practice that is spared to the rare cases where the variable is supposed to be shared between all instances of a class. If many variables in a class are to be shared between all its instances, this is usually a clear call for re-factoring the design. In an ontological paradigm, this is not the case. The use of static variables is the mechanism used by the translation module to reflect the relations between the $TBox$ and $RBox$ of the source ontology, i.e. the mechanism to say that all individuals of a given concept C can fill in a given role R. In turn, the role R itself has certain properties that should hold for all occurrences in which it appears. Figure 6.5 shows an example excerpt for an auto-generated class definition from the executable pizza ontology.

Moving on to the more quantitative measures: the resulting executable size and the execution time necessary to produce it. Tables 6.4 and 6.5 provide a summary of the metrics obtained applying the translation module on the reference ontologies. The most notable result to highlight here is the obtained reduction in the size of the source file and the .dll executable. Quite to the expectation, by avoiding the redundancy in the serialized XML format, the generated C# source file is up to 38% smaller in size than its XML counterpart. One thing to clarify here is that the redundancy in the source is also an inherent drawback in the syntax of the XML standard itself and not entirely linked to

```
public class American : NamedPizza
{

    public static hasTopping hasTopping;
    public static object subClassOf;
    public static hasCountryOfOrigin hasCountryOfOrigin;
    public static string label;

    static American()
    {
        label = "en:American ; pt:Americana";
        hasCountryOfOrigin = new hasCountryOfOrigin();
        hasTopping = new hasTopping();
        subClassOf = MozzarellaTopping;
        hasTopping.someValuesFrom =
            new List<PizzaTopping>()
                { MozzarellaTopping,
                  PeperoniSausageTopping,
                  TomatoTopping
                };
        hasTopping.allValuesFrom = MozzarellaTopping;
        hasCountryOfOrigin.hasValue = America;
    }

}
```

Figure 6.5 – An example excerpt from the executable source code of the Pizza ontology

recurrent axiom patterns. Yet, by detecting and referring to recurrent patterns, one can still get an extra size reduction of up to 9% for C# source files and 5% for the executable .dll files.

A somewhat less favourable results where obtained for the the translation time measure. Understandably, the most severe performance overhead was paid translating the SNOMED CT ontology where it takes the translation module around 22 hours before emitting the complete output C# source files. That is said, the translation module presented in this thesis amount to a first prototype version but in more applied settings, there is still a good margin to maneuver from working on an a parallel version of the translation algorithm to more resourceful deployment plans. The reported results were achieved running the translation module on a Windows 10 machine with an intel core i7-3770 2.50 ghz CPU and 16 gigabyte of RAM. To note that the numbers provided in this sections are based on best empirical results obtained. To fully analyze the performance results and the executable size reduction, one needs to disentangle all other contributing

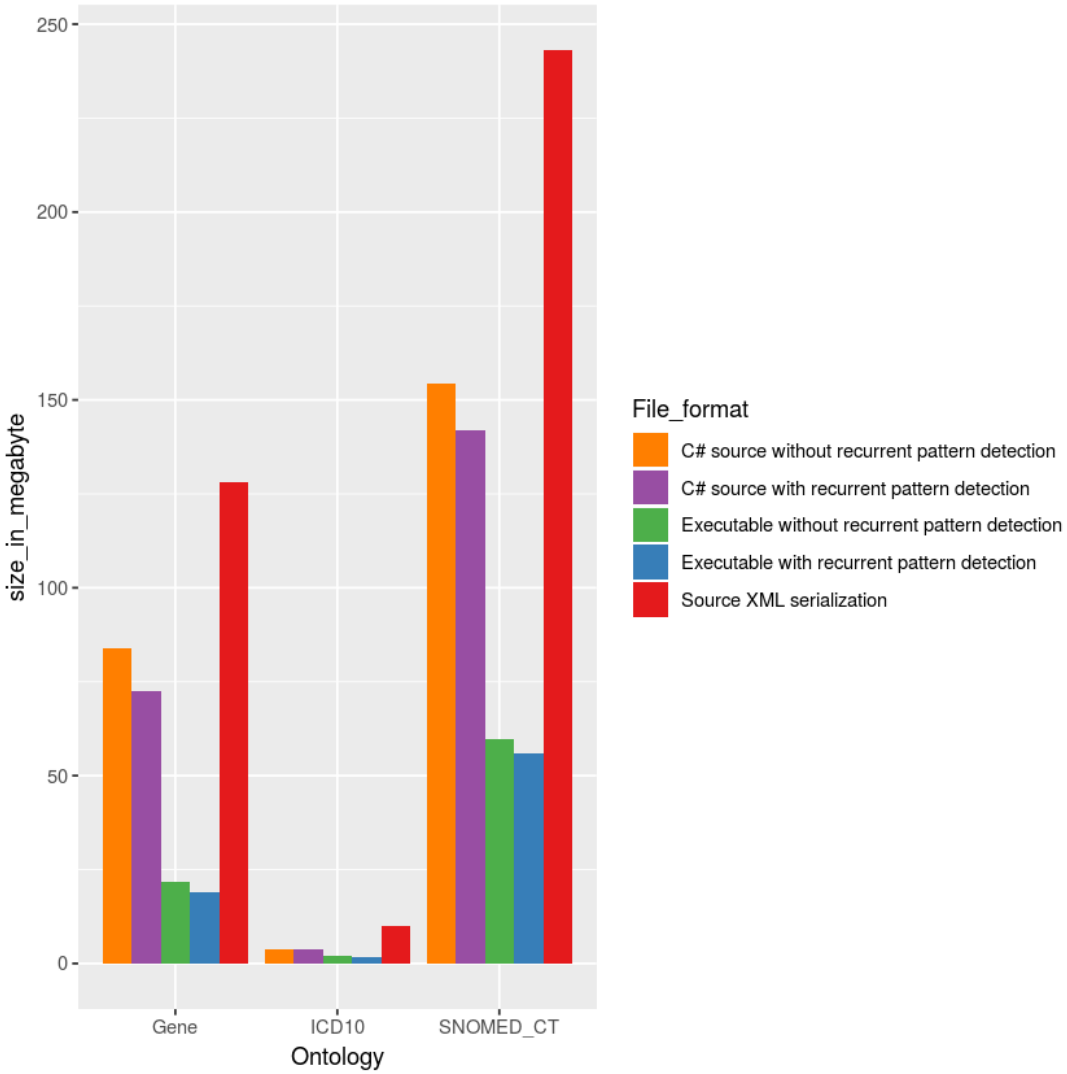


Figure 6.6 – A size comparison of executable ontologies and their source OWL XML serializations depending on the file format and the translation technique

factors apart from the known algorithmic ones. This includes the full CLR specifications, potential compiler optimizations and the underlying operating system support. Such an extensive analysis is considered for future work though.

6.6 Summary and outlook

In this chapter we covered the implementation of the translation function first defined in chapter 5. We started with a description of the translation module written in C# and

Table 6.4 – A summary of translating the terminologies ($TBox$ axioms) of some standard ontologies.

Ontology	Concept classes	Property classes	Anonymous classes	total classes	Translation time (second)	Compilation time (millisecond)	Source XML Size (k byte)	.cs file size (k byte)	executable file size (k byte)
Pizza	101	24	36	161	7.015	198	134	70	32
Wine	139	24	1	164	8.245	223	77	40	16
Gene	47'126	14	7188	54'328	31'074	28'945	131'456	83'809	21'827
ICD10	14'503	6	22	14'531	5'227	1'019	10'575	3639	1'286
SNOMED	339'521	120	13'662	353'263	79'932	98'185	248'910	154'320	59'719

Table 6.5 – A summary of parsing results for some standard ontologies - with recurrent patterns materialization

Ontology	Concept classes	Property classes	Anonymous classes	total classes	Translation time (second)	Compilation time (millisecond)	Source XML Size (k byte)	.cs file size (k byte)	executable file size (k byte)
Pizza	101	24	36	161	10.243	198	134	70	32
Wine	139	24	1	164	9117	223	77	40	16
Gene	47'126	14	6491	53'631	26'127	25'245	131'456	72'592	19'257
ICD10	14'503	6	22	14'531	6'173	1'019	10'575	3'639	1'286
SNOMED	339'521	120	12'136	351'737	68'221	71'291	248'910	141'981	56'905

then moved to cover some particularities of the translation process that are related to efficiently traversing the source format and other potential optimization techniques. We also demonstrated the translation results using a selection of ontologies of different sizes and expressiveness profiles. In the next chapter, we demonstrate a couple of potential applications of the translation results obtained.

Chapter 7

Applications of Executable Ontologies

> Any item may be caused at will to select immediately and automatically another. This is the essential feature of the memex. The process of tying two items together is the important thing.
>
> Vannevar Bush, As We May Think 1945

7.1 Introduction

Now that we have our initial question about translating ontologies into an executable form answered. The question that naturally arises next is what can such a new executable representation bring us? An executable representation of ontologies can be seen as embedding ontologies in the incubator of the programming environment where ontologies are blended into the source code rather than treated as passive external data models. The potentials here are wide open but as a starting point for a sensible answer to the utility question,

one can expect potential applications of executable ontologies to address the difficulties of expanding the use of ontologies into conventional enterprise software development.

Despite their proven utility in various areas of knowledge engineering, ontologies are falling short of reaching an equal position as formal domain models in the landscape of enterprise software development. This shy adoption of ontologies in software engineering communities is partially due to the opposing semantics we discussed in the previous chapters but also due to the technical stacks behind ontological applications and conventional enterprise software which are also very different from languages and editors to infrastructure support. Furthermore, in comparison to other modeling languages such as UML, ontologies are more suitable for capturing the static aspects of modeling such as taxonomies and relations between concepts, but they lag behind when it comes to capturing the dynamic behavior of the interacting components in the domain under consideration. For that purpose, UML still offers better options with a variety of diagrams: state, collaboration and sequence diagrams.

In this chapter we illustrate two potential applications of executable ontologies that we believe can alleviate, to a certain extent, the difficulties of using ontologies in conventional development environments. The first proposal we have is to make use of the support general-purpose programming languages have for the imperative paradigm to extend the executable ontology with the missing procedural control. The other proposal is related to light-weight reasoning services brought to conventional enterprise applications out-of-the-box thanks to the embedding of ontologies into the application code. The following sections provide more details on both proposals.

7.2 Procedural extensions

Except for Horn-like rules such as the ones provided by the SWRL extension [104], ontologies are mostly declarative. While in the knowledge engineering community this adherence

to the declarative paradigm is commonly perceived as a positive practice, we argue that it is one of the reasons behind the limited utility of ontologies in conventional software development. From a software development perspective, when modeling a target domain, it is essential to capture all aspects of the entities of that domain including their behavior and their possible interactions all while taking into account their sequential, or potentially parallel, temporal order. With the pure declarative paradigm ontologies offer, the developer is unable to capture the dynamic interactive parts of the system. Expressing ontologies in a language that permits the developer to switch from the pure declarative paradigm to a more imperative style when necessary can come very handy. Once in their executable form, it is now possible to extend ontologies by defining and implementing relevant static or instance object-oriented methods that will cover up for the missing procedural control.

A running example

To illustrate the idea we will use an example from the medical insurance domain. More specifically, let's consider the case of using AI techniques for detecting fraud attempts in claim reimbursement. In the medical insurance domain, a fraud attempt is committed when a health provider submits false or misleading information with the goal of augmenting the amount of health benefit that is payable by the health insurance company. Some examples of false claim information include billing for uncovered services as covered ones, incorrect reporting of diagnosis or procedures and performing unnecessary medical tests.

Given the additional costs fraud induces, it is no wonder that medical insurance companies would seek all means to detect fraud attempts as early as possible. Machine learning algorithms can help in detecting fraud patterns, but with one shortcoming: these algorithms, very often, provide no evidence on how the results are obtained. For that reason, rule-based reasoning on medical ontologies in combination with domain-specific ontologies are usually used for case-by-case claim processing [105].

In a typical health insurance information system, the enterprise application code is usually written in a robust general-purpose programming language, namely Java or C#. The enterprise application code comprises an object-oriented model of the domain that encodes both the static and dynamic aspects of the system. Domain-specific ontologies, also encoding most of the domain terminology, lay external to the main code repository leaving developers to deal with code redundancy and maintainability overhead. By translating the domain ontology into the main application programming language we are not only reducing the complexity of the technical stack for using ontologies; but also omitting the redundancy in the domain model by reusing the terminology already modeled in the ontology as a skeleton for the object-oriented model. It is then possible to complete the missing part of the model that were not possible to express using the ontological language by adding the proper method signatures along with their implementations.

Back to the claim reimbursement example, the *claim* as a concept in the source ontology is one of the main entities in the domain model. The ontological view of a claim concept is a static one. It states what a claim is supposed to be and lists all constraints to be respected by claim objects. In a real use case scenario, however, a claim object has different states and when modeling it in enterprise settings, it should be possible to dynamically change the state of the object to reflect the claim life-cycle from the moment it was received till the moment it is paid and closed.

Reusing the claim concept as depicted in the ontology could have lifted the need to re-engineer the claim class in the enterprise data model except that this reuse of concepts cannot cover the dynamic aspects of claims. What is missing in the ontological claim concept to form a self-contained interface that deals with claim objects is the procedural control of activities to be performed as part of the associated business processes. Beside this, it should also be easy to orchestrate the interactions of the model with other non-functional components of the system such as transaction management, notifications, database updates, etc. These are equally important factors to insuring the integrity and

proper execution of business processes. After translating the claim concept into an OOP language, it would be possible to add the missing procedural control to properly manage claim states as method signatures, such as `Validate()`, `Pay()` and `Close()`, in the generated interface. The implementation of these methods is then to be added to the generated claim class. The interface signatures along with their implementation serve as procedural extensions to the ontology concept in its executable form. This insures capturing all aspects of the interactions between entities in the domain as well as a proper execution of all related operations.

7.3 Light-weight reasoning services

By reversing the perspective from which we looked at the procedural extensions from previous section, we can perceive yet another application of executable ontologies. Just like an ontology, an Enterprise Data Model (EDM) captures the semantics of business operations within an organization. It has all what it takes to represent a domain from entities to relations and business logic. Compared to an ontology, however, the EDM lacks the power of reasoning and knowledge-based inference capabilities. In principle, integrating some form of ontological modeling and reasoning capabilities can always offer some advantages for an enterprise. First of all, it enhances the interoperability of the enterprise information system and second, it allows the organization to exploit inference-based reasoning to discover implicit knowledge out of the explicitly modeled information. On the other hand, integrating ontologies is no free lunch. Many factors contribute to the difficultly of integrating an ontology into the enterprise information system of an organization. Among these factors are the completely different technical stacks of knowledge-based systems involving ontologies and the ontological paradigm twist in comparison to conventional software development. Both result in a steep learning curve for most software developers. In many cases, the rapidly accumulating costs of building an ontology, the technical integration and human resource training yield a negative cost-benefit ratio [106] [107] that

pushes many organizations to abandon the idea in the first place.

Having shortly demonstrated the motivation behind, it is now time to move on to describe our proposal for a knowledge-based system that is built on top of the new executable representation. The following sections are reserved to an illustration of a prototype of a C# knowledge-based system made possible thanks to the translation module covered in chapters 5 and 6.

7.3.1 OntoJIT Proptotype

In theory, as long as the semantic profile of an ontology is maintained, the set of reasoning tasks that were decidable in its OWL version should also be decidable in its new executable form. This is mainly because the entailment procedure, expressed in a Turing complete language, is orthogonal to the different syntactic representation formats of OWL concepts. In that case and in order to get the full spectrum of logic-based inference reasoning, one has to adapt existing implementations of reasoning algorithms such as the tableau algorithm or structural subsumption algorithm to read the new representation format. In doing so, the technical stack is already simplified as there is no need anymore to connect to an external reasoning service but the complexity of the reasoning remains intact. What we are rather interested in is exploring other out-of-the-box possibilities to realize the inference-based question-answering feature of an ontological knowledge-base system.

One interesting point that kind of guided us in our quest is an observation we made about many of the well established ontologies in the literature. In fact, these ontologies are mostly, if not merely, composed of terminological axioms and the majority of queries concerning them are TBox queries. In the case of executable ontologies this means queries about classes rather than instances. A mechanism to treat classes as objects is thus necessary in order to write queries and properly handle the results. A partial answer to the problem lies in exploiting some meta-programming technique that allows a program to

have knowledge of itself and gives it access into its type information. In C# and some other .Net languages this technique is called reflection [108][109]. As for the query mechanism, the C# Language Integrated Query (LINQ) [1] offers a very reasonable candidate for the task. These last two mechanisms together with the translation module discussed in chapter 6 are the main components in the prototype we are presenting in this section.

Figure 7.1 depicts the general architecture for the knowledgebase system that was entirely realized in C#. The system has two input possibilities: 1) using the translation module, one can translate existing ontologies into C# code. 2) one can as well directly express ontologies in plain C#. This is possible as long as the classes the developer adds inherit from the top class `Thing` (or `OWLProperty` in case of a property) and conforms to the conventions established as translation rules.

Depending on the application specific needs, if necessary, procedural extensions in the form of object-oriented methods can be added at this point. Next comes the compile unit layer of the prototype. This layer is nothing but a virtual representation of the translated ontology that can be compiled during the runtime thanks to the .Net Common Language Runtime (CLR) compiler. Once compiled, the compile unit yields the ontology in its executable form as a dynamic linking library file (.dll extention). From now on, all concepts of the ontology are accessible via reflection and it is now possible to formulate questions against the ontology as LINQ queries. Finally, in cases where abstracting the technical details of LINQ queries is desired, an optional encapsulating query API completes the prototype. This layer makes it possible for external client applications to specify query terms without having to deal with the corresponding internal LINQ expressions.

1. LINQ: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

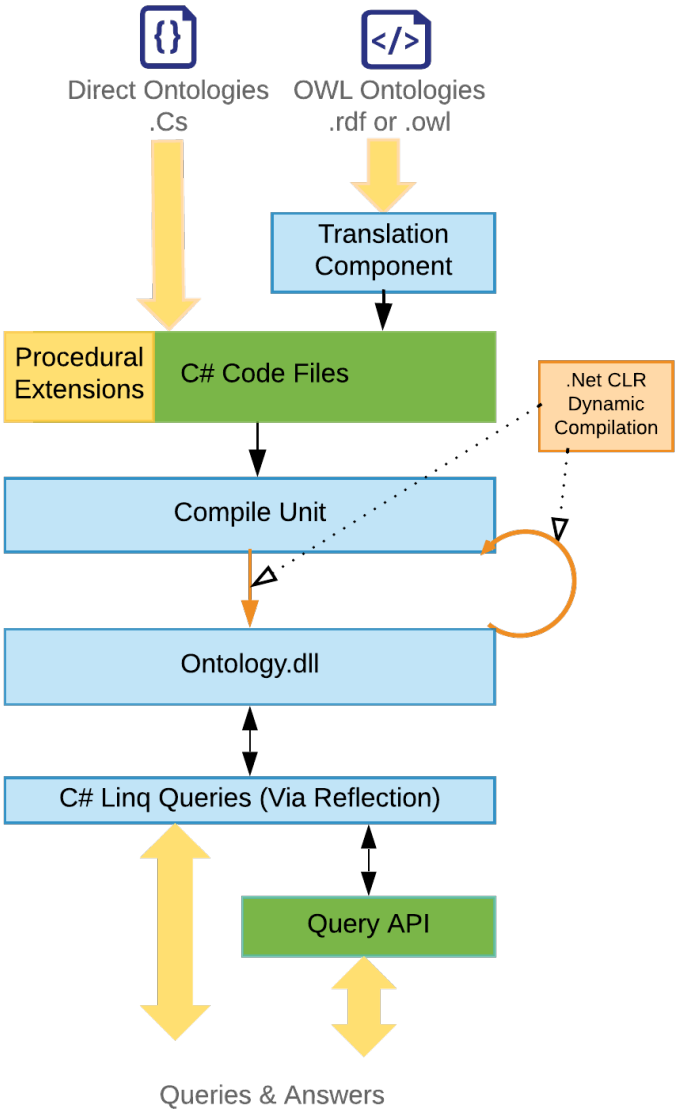


Figure 7.1 – A prototype for an ontological knowledgebase system based on .Net Common Language Runtime

7.3.2 Semantic query entailment

In convectional KB systems containing OWL ontologies there are two main options to answer queries involving any form of entailment reasoning: 1) Outside the programming environment e.g. using Protégé DL query editor. In this case query answering is an "offline" operation and results need to be exported from Protégé as data and re-integrated into the program. 2) Within the programming environment by connecting to an external reasoning component via OWL API and manually casting from and to the corresponding

types in the embedding program code. In both cases, it is also necessary to run the connected reasoner to perform a global classification of the ontology before evaluating any query. More recently, SPARQL endpoints are also gaining popularity as they allow the exportation of query results in JSON format. Here again, the problem of manually casting JSON data into the corresponding code types is left to the developer. Further more, SPARQL itself is a query language that is used to formulate queries against an RDF graph but the entailment regime used to answer these queries is dependant on the back-end implementation of the endpoint and query results obtained may very well be limited to the facts explicitly stated in the source ontology. In other words, SPARQL itself provides no universal guarantee on the completeness of the results.

In the C# executable form, it is not any more necessary to switch editors, to use an API or to cast queries or results. The query and the results are both expressible as native C# code within the "everyday" programming environment the developer is acquainted to. To take things to the next level, our aim was to test what kind of entailment reasoning is possible in these new settings without running a reasoner on the whole ontology. To that end and without tempering with LINQ query entailment mechanism, we tested a reasoning on-the-go approach where the necessary supplementary query semantics, if any, can be provided by the developer in a case-per-case basis.

This section adopts an empirical approach to demonstrate some of the new possible forms of entailment queries using examples from two of the well-known reference ontologies in the domain of knowledge engineering. For brevity and to keep in line with the ideas discussed in the previous section, we will limit the scope of our examples to querying the terminological part of the ontology. Other forms of assertional queries have been tested successfully though. All queries presented here are entirely based on the integrated query mechanism of C#. No other query language, editor nor engine were required.

```
static IList<string> GetTransitiveClosure(IList<string> set)
{

    Queue<string> queue = new Queue<string>();
    set.ToList().ForEach(item => queue.Enqueue(item));
    IList<string> closure = new List<string>();
    string type = null;
    while (queue.Count > 0)
    {
        type = queue.Dequeue();
        closure.Add(type);
        FindSubTypes(type).ToList().ForEach(t => queue.Enqueue(t));
        FindEquivalentTypes(type).ToList().ForEach(t => closure.Add(t));
    }
    return closure;
}
```

Figure 7.2 – A code snippet depicting the transitive closure over the subclass and equivalentClass meta-properties.

Example 1. The first example is drawn from the Gene ontology. To start simple, we will first consider the DL query to find all subtypes of the concept chromosome. In this query all we had to consider is the transitive closure over two relations: the class inheritance and equivalent class. The inheritance relation is, most of the time, readily available via the type information in the programming environment or accessible via the `subClass` meta-property. The equivalent class information is accessible directly via its corresponding meta-property. To better illustrate the idea, a simple implementation of the transitive closure is depicted in Figure 7.2. `GetTransitiveClosure()` is a function that uses a queue data structure to collect and return the transitive closure of a set of types over the `subClass` and `equivalentClass` meta-properties. The complete implementation of the `FindSubTypes()` and `FindEquevalentTypes()` methods is included in appendix A.

It is worth mentioning here that this implementation is not unique to the query in example 1. The transitive closure depicted here constitutes an essential part of the mechanism we used for semantic query entailment and it is used throughout all the examples provided in this chapter. The LINQ query we used for finding all chromosomes is shown in Figure 7.3 and its Results are shown in Figure 7.4.

```
var transitiveClosure = GetTransitiveClosure(typeof(chromosome));

var chromosomes = from    type in Assembly.GetExecutingAssembly().GetTypes()
                  where   transitiveClosure.Contains(type)
                  select  type;

```

Figure 7.3 – LINQ query for the query term 'chromosome' from Example 1.

Example 2. Also drawn from the gene ontology but with a slightly more complicated query to answer: Find all chromosomes that are part of a cytoplasm. This translates into the conjunctive DL query:

$$Answers \leftarrow \exists x.\big(\text{chromosome}(x) \wedge \text{is\_part}(x.y) \wedge \text{cytoplasm}(y)\big)$$

The corresponding LINQ query is shown in Figure 7.5. Like in the previous example, before running the query it is necessary to get the transitive closure of concepts appearing in the query. This step is the alternative to running the reasoner to insure implicit types are also taken into account. The necessary axioms to answer this query from the source gene ontology are translated into C# type information and are accessible via reflection. The four chromosomes satisfying the query are shown in Figure 7.6.

7.3.3 A test experiment

The ontology we will use to further demonstrate light-weight reasoning possibilities on the new executable representation is the pizza ontology from Manchester university. Our choice was motivated by the fact that this ontology is based on the $\mathcal{SHOIN}$ fragment of DL. $\mathcal{SHOIN}$ is a highly expressive profile that would constitute a good benchmark for

GO_0000228: nuclear chromosome
GO_0000229: cytoplasmic chromosome
GO_0000262: mitochondrial chromosome
GO_0000793: condensed chromosome
GO_0000794: condensed nuclear chromosome
GO_0000803: sex chromosome
GO_0000804: W chromosome
GO_0000805: X chromosome
GO_0000806: Y chromosome
GO_0000807: Z chromosome
GO_0001740: Barr body
GO_0001741: XY body
GO_0005700: polytene chromosome
GO_0009508: plastid chromosome
GO_0030849: autosome
GO_0042648: chloroplast chromosome
GO_0098577: inactive sex chromosome
GO_0098579: active sex chromosome

Figure 7.4 – Query results for the query term 'chromosome' from Example 1.

comparing query entailment results. The modest size of the $TBox$ in the Pizza ontology also permits a more systematic approach for query results' validation. As a fixed reference model, we used the query template of the Pizza Finder application [2] to suggest pizzas that have some desired toppings but none of the specified excluded toppings. In the following sub-sections, we describe the experiment we conducted more in details from query term generation to query evaluation and results. Finally we provide a brief discussion of our findings.

2. The Pizza Finder application is a tutorial application developed by Manchester university to demonstrate OWL semantics and reasoning capabilities http://owl.cs.manchester.ac.uk/research/co-ode/pizza-finder/

```
var chromosomes = GetTransitiveClosure(typeof(chromosome));
var cytoplasms  = GetTransitiveClosure(typeof(cytoplasm));

var types = from    type in Assembly.GetExecutingAssembly().GetTypes()
            let     property = type.GetField("part_of")
            where   chromosomes.Contains(type) &&
                    cytoplasms.Intersect(ExpandPropertyChain(property)).Count() > 0
            select  type;
```

Figure 7.5 – C# LINQ query of example 2: All chromosomes that are part of a cytoplasm.

GO_0000229: cytoplasmic chromosome
GO_0000262: mitochondrial chromosome
GO_0009508: plastid chromosome
GO_0042648: chloroplast chromosome

Figure 7.6 – Results for the LINQ query from example 2.

Query generation

In order to automate query term generation, we wrote a small script to randomly pick up distinct combinations of query terms from a pool of pizza topping concepts as provided by the source ontology. We used the script to generate pairs of included/excluded topping combinations of different sizes starting from 1 term per combination up to 5 terms. Our goal in setting up the experiment was to include as much diverse cases as possible. Therefore, the term pick-up mechanism we used is blind to any potential relations between the chosen concepts. In different settings where the goal is to optimize query execution time, it is recommended to use a query rewriting mechanism that can detect relations such as subsumption or disjointness and substitutes, or omits, the relevant and redundant terms correspondingly. Finally, we fed the same generated pairs of included/ excluded toppings in batches of 10000 pairs into the query templates in the OntoJIT prototype as well as in the Pizza Finder Java application.

Query evaluation

Even though both query templates are expected to return the same results, the semantics used to formulate and evaluate queries in OntoJIT is different from that of the Pizza Finder application. Query evaluation in the Java application is based on the DL subsumption, i.e. it starts by creating a temporary concept of conjunctive terms and negated terms; it then invokes the connected reasoner in order to classify the concept created under the proper level of the taxonomy. The query is then evaluated against the just classified ontology and the results are all the concepts in the ontology that are subsumed by the concept created temporarily. In OntoJIT, queries were formulated as the relative complement $A \setminus B$ of two sets A and B; where A and B are the sets denoting the transitive closures over the subclass and equivalent class meta-properties of included and excluded topping classes respectively.

Query results

Automatically comparing the result sets returned by OntoJIT to the benchmark results returned by the OWL API in the Pizza Finder application, no false negatives were left out; meaning that for each of the queries, all results returned by the Pizza Finder application were also returned by the OntoJIT prototype. For some of the queries, however, few extra results were returned by OntoJIT but not by its Java counterpart and a manual inspection of the results was necessary to judge whether the extra results qualify for false positives. By tracing the query execution one can see that the extra results returned by OntoJIT may (or may not) be considered as false positives as they are subject to the interpretation of the underlying assumption. Here again, the opposing semantics of the OWA in ontological modeling vs the CWA in conventional software modeling are to blame. It is in fact straightforward to verify that the results that were left-out by the OWL API were missing due to the open-world assumption used in DL entailment which forbids entailing the truth value of a candidate fact in the absence of all corresponding axioms

from the KB. In OntoJIT, the adopted closed-world assumption allows a more decisive query evaluation by considering 'False' as the value for any missing fact. This explains yielding a more relaxed result set in some cases where the "background knowledge" axioms are missing.

As a concrete example, we can consider the query to find all pizzas that have meat among other toppings but are not spicy. The query results returned using OWL API and OntoJIT are: `{"American","FourSeason" }` and `{ "American", "Capricciosa", "FourSeason", "LaReine", "Paremense", "Siciliana" }` respectively. The set relative complement approach of OntoJIT assumes reasoning in a closed world where sets are complete. The query evaluation would thus consider pizzas that have any kind of meat and would similarly exclude all spicy pizzas according to the facts present in the knowledge base. This means that it does not consider `"LaReine"` as a spicy pizza since nothing related to spiciness was present in the KB. Contrarily, the DL reasoner would look for axioms stating spiciness information and when it fails to find any, it forbids deducing further conclusions about the concept and `"LaReine"` would not therefore belong to the query results. To further confirm this hypothesis, we repeated the experiment after tweaking the settings so that OWL API would run the queries under the CWA assumption. Closing the world, locally or globally, is possible by providing default values for the missing axioms in the source ontology [94] [99] [100]. Repeating the experiment this second time after completing the "pizzaTopping" definitions with the default facts yielded identical result sets for all query terms.

As for query execution time, all queries were run on a machine running Windows 10 with an intel core i7-3770 2.50 ghz CPU and 16 gigabyte of RAM. Table 7.1 provides an aggregate view of the results obtained running the experiment. This includes the average OntoJIT query execution time in milliseconds. We do not include the same metric for OWL API because by using a client Java application we wrote to pass the query terms and get the results from OWL API, we were unable to isolate the net query execution time and

Table 7.1 – An aggregate view of batch query evaluation results under OWA (Batch size =10000 queries.)

Terms per combination	Execution time avg. (milliseconds)	Matching result sets	Missmatch rate %	False positives avg. count
1	4.16	9551	4.49	2.121
2	3.137	9649	3.51	1.913
3	1.115	9823	1.77	0.871
4	2.009	9902	0.98	0.292
5	1.689	9989	0.11	0.018

listing the gross recorded times would not be a fair comparison. To to give an approximate indication, the average recorded overall time it took the Java client to receive results from the OWL API (regardless to the number of terms per query) was 13.46 milliseconds. The matching result set column lists the number of queries, out of 10000, that had the exact same result set obtained in both query engines. The same information is presented differently in the mismatch rate column. Finally, the last column represents the average number of false positives per query (under an open-world assumption). The improved results obtained in queries with higher number of terms can be explained by the fact that with a random query term generation, it gets more difficult to find pizzas satisfying queries with more toppings to include/exclude. For that reasons most the result sets obtained are empty and the probability for returning false positives drops consequently.

Discussion

The experiment we presented in this chapter was intended to provide a "first feeling" of the feasibility of exploiting the programming environment support in answering semantic queries against ontologies in their executable form. As it has been empirically shown in this section, exclusively relying on the native features of the programming language along with some basic entailment mechanism such as the transitive closure of query terms, one can expect reliable query result sets under a closed-world assumption. Further more, the obtained empirical results are encouraging given that reasoning under an OWA does not

constitute the majority of use cases in real-world scenarios.

That said, it is equally important to point out that relying on LINQ can be a double-edged sword. LINQ offers a convenient and flexible way for query formulation regardless of the underlying data source making it possible for developers who have never used ontologies to write semantic queries out-of-the-box using the very same query language they are familiar with. Further more, when used in combination with C# lambda expressions and anonymous types as predicates in the body of the query, LINQ offers a powerful set of possibilities for formulating highly expressive queries. This last strength point, however, is also where overly relying on LINQ can become problematic. In order to enable shifting towards a more compact functional programming style, what LINQ provides here is a higher level of abstraction of all the operations and data iterations executed under the hood; which means hiding complexity cost of the query or in best case scenarios leaving it to the hands of experienced developers to properly take care of complexity analysis.

7.4 Summary and conclusion

In this chapter we presented two potential applications of translating ontologies into an executable representation. Both applications were geared towards addressing the difficulties of fully exploiting ontologies in conventional development environments. We first proposed bridging the imperative control gap by providing executable ontologies with procedural extensions in the form of object-oriented methods. We then moved to present a more elaborated prototype of a simplified stack for a knowledge base system that is based on the new executable representation with no external tools or environment dependencies. We concluded the chapter with an experimental analysis on the feasibility of using the proposed prototype in answering semantic queries.

Chapter 8

Conclusion

> My interest is not the Moon. To me the Moon is as dull as a ball of concrete, but we're not going to have a research base on Mars until we can learn how to do it on the Moon first. The Moon provides a blueprint to Mars.
>
> Chris McKay, NASA astrobiologist

8.1 Summary of thesis achievements

Adopting ontologies in software engineering practices is a subject that has received high appraisal from a multitude of research papers over the last two decades. However, shifting a bit from research towards the circles of conventional software engineering reveals a rather conservative mindset on how ontologies can be actively integrated into the software development life cycle. This thesis had the objective of democratizing the access to ontologies into a wider audience of conventional software developers.

Given that the academic literature on the topic spans a couple of decades, most of the related work was scattered over papers pertaining to different emerging research arenas that in many times first appear to be non-relevant. That is why the first contribution in this thesis was the survey presented in chapter 4. This survey helped in establishing the state-of-the-art by systematically collecting, classifying and analyzing the different approaches used to integrate ontologies into more general development environments.

Chapter 5 proposed a loss-less translation function that would facilitate the task of integrating ontologies into conventional code repositories by expressing them directly in the same general-purpose language already in use. Such a translation does not only enhance the portability of ontologies but also contributes to a substantial reduction in both the steep learning curve and the overhead of the technical stacks usually associated with ontologies.

This proposition was further crystallized in chapter 6 by an implementation of the translation function with C# as the target language of the translation. The chapter also suggested a compressing technique that is particularly useful in the bio-medical domain where the sheer size of ontologies is itself an issue. The technique is not unique to executable ontologies but can also be generalized to other syntactic formats of OWL ontologies.

Finally, chapter 7 presented procedural extensions and light-weight reasoning services as two potential direct applications of executable ontologies.

8.2 Threads to validity

The proof by construction approach constituted a solid starting point that permitted the validation of the core ideas proposed but it goes with out saying that for some of the contributions, further theoretical analysis is now due in order to fully formalize the propositions. This is mainly the case for the work related to light-weight query entail-

ment. While the empirical results obtained provides the necessary evident to support the feasibility of the approach, the link between the two forms of reasoning is yet to be formally established. As discussed earlier, relying on the integrated query mechanism of the programming environment lifts the necessity for external reasoning services but it also shifts the responsibility of query complexity analysis to the developer plate. The completeness and correctness of the obtained results were also measured empirically but a full proof is yet to be formally established.

8.3 Future Work

Beside the ongoing work on a parallel version of the translation module and the formal analysis mentioned in the previous section, another interesting research direction for future work is the potential role executable ontologies can play in improving the requirement engineering phase in software engineering projects.

The motivation here is that for many application domains, the business logic (i.e. domain knowledge) is already captured by the domain ontologies. That, together with a reliable means to translate ontologies into executables would make for an out-of-the-box recipe for automated pipeline of requirement engineering and code generation. By automatically translating the $TBox$ axioms into executable type information, the discrepancy between the intended model and the code implementation, which is usually a recurrent problem in software development, is much less evident with executable ontologies. Further more, in agile project settings, this "the model is the code" approach can also offer a more graceful solution facing repeated episodes of changing requirements.

This research direction is partially inspired by the ODSD ideas discussed in the survey chapter [73] and is supposed to build on top of the research efforts spent on utilizing ontologies as domain knowledge for requirements elicitation including the work in [110], [111] and [112].

8.4 End word

With research being a collaborative act in nature, an authorial "we" was used throughout the text. Having reached the end, however, it is now time to disclose a more personal narrative.

Myself, A humble software engineer starting a PhD in the area of knowledge representation, I can very well remember feeling uneasy when first in the presence of ontologies. It was rather clear from the very beginning what kind of calling was waiting for me. Highly motivated, I took democratizing access to ontologies as a mission. And so, the journey starts with a high spirit but along the way it turns out that pursuing that goal was anything but an endless roller coaster ride. I spent the last few years alternating between the feeling of being overwhelmed, accomplished or in doubt. Countless are the moments where it was all void.

Now that the journey has come to its end, I am happy to have taken the opportunity, pleased with the outcomes and reassured realizing that what I have been through has less to do with lack of resilience and more with how the PhD endeavor is meant to be. At this moment all I have left to declare is my hope that the work presented in this thesis will have a positive impact, great or slight, on other software engineers hesitating to approach the wonderland of ontologies.

Bibliography

[1] Jean E Sammet. Programming languages: history and future. *Communications of the ACM*, 15(7):601–610, 1972.

[2] AD Carpenter and Gail Fine. Plato on knowledge and forms: selected essays, 2008.

[3] Gregory Vlastos. Parmenides' theory of knowledge. In *Transactions and Proceedings of the American Philological Association*, pages 66–77. JSTOR, 1946.

[4] Franz Baader. *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[5] R.J. Brachman and H.J. Levesque. *Knowledge Representation and Reasoning*. The Morgan Kaufmann Series in Artificial Intelligence Series. Morgan Kaufmann, 2004.

[6] Ben Goertzel and Cassio Pennachin. *Artificial general intelligence*, volume 2. Springer, 2007.

[7] J Lighthill. A report on artificial intelligence. *UK Science and Engineering Research Council*, 1973.

[8] John McCarthy. Review of artificial intelligence: A general survey. *Artificial Intelligence*, 5(3), 1974.

[9] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[10] M Ross Quillian. Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral science*, 12(5):410–430, 1967.

[11] John F Sowa. *Principles of Semantic Networks: Explorations in the representation of knowledge*. Morgan Kaufmann, 2014.

[12] Marvin Minsky. A framework for representing knowledge. 1975.

[13] Daniele Nardi, Ronald J Brachman, et al. An introduction to description logics. In *Description logic handbook*, pages 1–40, 2003.

[14] Fritz Lehmann. Semantic networks. *Computers & Mathematics with Applications*, 23(2-5):1–50, 1992.

[15] Ora Lassila and Deborah McGuinness. The role of frame-based representation on the semantic web. *Linköping Electronic Articles in Computer and Information Science*, 6(5):2001, 2001.

[16] Sigmund Freud. The dynamics of transference. *Classics in Psychoanalytic Techniques*, 1912.

[17] M.T. Jones. *Artificial Intelligence: A Systems Approach: A Systems Approach*. Jones & Bartlett Learning, 2015.

[18] Ronald J Brachman and James G Schmolze. An overview of the kl-one knowledge representation system. *Cognitive science*, 9(2):171–216, 1985.

[19] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM (JACM)*, 42(4):741–843, 1995.

[20] Vinay K Chaudhri, Adam Farquhar, Richard Fikes, Peter D Karp, and James P Rice. Okbc: A programmatic foundation for knowledge base interoperability. In *AAAI/IAAI*, pages 600–607, 1998.

[21] Peter Clark, Bruce Porter, and Boeing Phantom Works. Km—the knowledge machine 2.0: Users manual. *Department of Computer Science, University of Texas at Austin*, 2:5, 2004.

[22] Steve Hanks and Drew V McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *AAAI*, volume 86, pages 328–333, 1986.

[23] Robert C Moore. *The role of logic in knowledge representation and commonsense reasoning*. SRI International. Artificial Intelligence Center, 1982.

[24] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM.

[25] Robert A. Kowalski. The early years of logic programming. *Commun. ACM*, 31(1):38–43, January 1988.

[26] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[27] Chitta Baral. Answer set programming. 2004.

[28] Jan Wielemaker, S Ss, and I Ii. Swi-prolog 2.7-reference manual. 1996.

[29] Jan Wielemaker, Wouter Beek, Michiel Hildebrand, and Jacco van Ossenbruggen. Cliopatria: A swi-prolog infrastructure for the semantic web. *Semantic Web*, 7(5):529–541, 2016.

[30] Jan Wielemaker, Michiel Hildebrand, Jacco Van Ossenbruggen, et al. Using prolog as the fundament for applications on the semantic web. *Proceedings of ALPSWS2007*, pages 84–98, 2007.

[31] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *The knowledge engineering review*, 18(03):197–207, 2003.

[32] Michael Kifer. Nonmonotonic reasoning in flora-2. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 1–12. Springer, 2005.

[33] Zhiming Pan and I Horrocks. *Description Logics: reasoning support for the Semantic Web*. University of Manchester, 2004.

[34] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial intelligence*, 48(1):1–26, 1991.

[35] Mark A Musen. The protégé project: a look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[36] Ullrich Hustadt, Renate A Schmidt, and Lilia Georgieva. A survey of decidable first-order fragments and description logics. *Journal of Relational Methods in Computer Science*, 1(251-276):3, 2004.

[37] Peter F Patel-Schneider, Deborah L McGuinness, Ronald J Brachman, and Lori Alperin Resnick. The classic knowledge representation system: Guiding principles and implementation rationale. *ACM SIGART Bulletin*, 2(3):108–113, 1991.

[38] Deborah L McGuinness and Lori Alperin Resnick. Description-logic based configuration for consumers. In *Proceedings of DL-95, 4th International Workshop on Description Logics*, pages 109–111, 1995.

[39] Aldo Gangemi, Geri Steve, and Fabrizio Giacomelli. Onions: An ontological methodology for taxonomic knowledge integration. In *ECAI-96 Workshop on Ontological Engineering, Budapest*, volume 95, 1996.

[40] Thomas R Gruber et al. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[41] Willem Nico Borst. *Construction of engineering ontologies for knowledge sharing and reuse*. Universiteit Twente, 1997.

[42] Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. *ACM Sigmod Record*, 35(3):34–41, 2006.

[43] Ian Horrocks et al. Daml+oil: A description logic for the semantic web. *IEEE Data Eng. Bull.*, 25(1):4–9, 2002.

[44] Deborah L McGuinness, Richard Fikes, James Hendler, and Lynn Andrea Stein. Daml+ oil: an ontology language for the semantic web. *IEEE Intelligent Systems*, 17(5):72–80, 2002.

[45] Ora Lassila and Ralph R Swick. Resource description framework (rdf) model and syntax specification. 1999.

[46] S. Powers. *Practical RDF: Solving Problems with the Resource Description Framework*. O'Reilly Media, 2003.

[47] Boris Motik, Peter F Patel-Schneider, and Bernardo Cuenca Grau. Owl 2 web ontology language direct semantics. *W3C recommendation*, 27, 2009.

[48] Boris Motik, Peter F Patel-Schneider, and Bernardo Cuenca Grau. Owl 2 web ontology language direct semantics. *W3C recommendation*, 27, 2009.

[49] Ian Horrocks. Owl: A description logic based ontology language. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 5–8, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[50] Ian Horrocks, Peter F Patel-Schneider, and Frank Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Web semantics: science, services and agents on the World Wide Web*, 1(1):7–26, 2003.

[51] Brian McBride. The resource description framework (rdf) and its vocabulary description language rdfs. In *Handbook on ontologies*, pages 51–65. Springer, 2004.

[52] Eric Prud, Andy Seaborne, et al. Sparql query language for rdf. 2006.

[53] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006.

[54] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. Sparql 1.1 query language. *W3C recommendation*, 21(10), 2013.

[55] Colin Atkinson, Matthias Gutheil, and Kilian Kiko. On the relationship of ontologies and models. *WoMM*, 96:47–60, 2006.

[56] Colin Puleston, Bijan Parsia, James Cunningham, and Alan Rector. Integrating object-oriented and ontological representations: A case study in java and owl. In *International Semantic Web Conference*, pages 130–145. Springer, 2008.

[57] Neil M Goldman. Ontology-oriented programming: static typing for the inconsistent programmer. In *International Semantic Web Conference*, pages 850–865. Springer, 2003.

[58] Stephen Cranefield. Uml and the semantic web. 2001.

[59] Dragan Gašević, Nima Kaviani, and Milan Milanović. Ontologies and software engineering. In *Handbook on Ontologies*, pages 593–615. Springer, 2009.

[60] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[61] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[62] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009.

[63] Francisco Ortin, Daniel Zapico, J Baltasar García Pérez-Schofield, and Miguel Garcia. Including both static and dynamic typing in the same programming language. *IET software*, 4(4):268–282, 2010.

[64] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, March 1974.

[65] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J Eggers, and Brian N Bershad. Fast, effective dynamic compilation. In *ACM SIGPLAN Notices*, volume 31, pages 149–159. ACM, 1996.

[66] Chandra J Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, 2001.

[67] Thomas S Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012.

[68] Peter Van Roy et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104:616–621, 2009.

[69] Peter Van-Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[70] R. Harper. *Practical Foundations for Programming Languages*. Titolo collana. Cambridge University Press, 2016.

[71] Selena Baset and Kilian Stoffel. Object-oriented modeling with ontologies around: A survey of existing approaches. *International Journal of Software Engineering and Knowledge Engineering*, 28(11n12):1775–1794, 2018.

[72] Adam Farquhar, Richard Fikes, and James Rice. The ontolingua server: A tool for collaborative ontology construction. *International journal of human-computer studies*, 46(6):707–727, 1997.

[73] Holger Knublauch. Ontology-driven software development in the context of the semantic web: An example scenario with protege/owl. In *1st International workshop on the model-driven semantic web (MDSW2004)*, pages 381–401. Monterey, California, USA.[WWW document] http://www. knublauch. com/publications/MDSW2004. pdf, 2004.

[74] A Eberhart. Ontojava–applying mainstream technology to the semantic web. In *Workshop on Semantic Web-based E-Commerce and Rules Markup Languages at the ICEC, Vienna (Austria)*, 2001.

[75] Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A Padget. Automatic mapping of owl ontologies into java. In *SEKE*, volume 4, pages 98–103. Citeseer, 2004.

[76] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based integration of information-a survey of existing approaches. In *IJCAI-01 workshop: ontologies and information sharing*, volume 2001, pages 108–117. Citeseer, 2001.

[77] Keith L Clark and Frank G McCabe. Ontology oriented programming in go! *Applied Intelligence*, 24(3):189–204, 2006.

[78] Andreas Eberhart. Automatic generation of java/sql based inference engines from rdf schema and ruleml. In *International Semantic Web Conference*, pages 102–116.

Springer, 2002.

[79] Sean K Bechhofer and Jeremy J Carroll. Parsing owl dl: trees or triples? In *Proceedings of the 13th international conference on World Wide Web*, pages 266–275. ACM, 2004.

[80] Jeremy J Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83. ACM, 2004.

[81] Seiji Koide and Hideaki Takeda. Owl-full reasoning from an object oriented perspective. In *Asian Semantic Web Conference*, pages 263–277. Springer, 2006.

[82] Max Völkel and York Sure. Rdfreactor-from ontologies to programmatic data access. In *Poster Proceedings of the Fourth International Semantic Web Conference*, page 55, 2005.

[83] Marian Babik and Ladislav Hluchy. Deep integration of python with web ontology language. In *In Proceedings of the 2nd Workshop on Scripting for the Semantic Web*. Citeseer, 2006.

[84] Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. Activerdf: object-oriented semantic web programming. In *Proceedings of the 16th international conference on World Wide Web*, pages 817–824. ACM, 2007.

[85] Ioannis N Athanasiadis, Ferdinando Villa, and Andrea-Emilio Rizzoli. Ontologies, javabeans and relational databases for enabling semantic programming. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 341–346. IEEE, 2007.

[86] Agostino Poggi. Owlet: an object-oriented environment for owl ontology. In *Proceedings of the 11th WSEAS International Conference on Computers*, pages 44–49. World Scientific and Engineering Academy and Society (WSEAS), 2007.

[87] Michael Zimmermann. Owl2java: a java code generator for owl, website, 2009.

[88] Agostino Poggi. Developing ontology based applications with o3l. *WSEAS Trans. on Computers*, 8(8):1286–1295, 2009.

[89] Sven Groppe, Jana Neumann, and Volker Linnemann. Swobe-embedding the semantic web languages rdf, sparql and sparul into java for guaranteeing type safety, for checking the satisfiability of queries and for the determination of query result types. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1239–1246. ACM, 2009.

[90] Graeme Stevenson and Simon Dobson. Sapphire: Generating java runtime artefacts from owl ontologies. In *International Conference on Advanced Information Systems Engineering*, pages 425–436. Springer, 2011.

[91] Alexander Paar and Denny Vrandečić. Zhi#–owl aware compilation. In *Extended Semantic Web Conference*, pages 315–329. Springer, 2011.

[92] Ken Wenzel. Ontology-driven application architectures with komma. In *Workshop on Semantic Web Enabled Software Eng*, 2011.

[93] Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Semantic web application development with liteq. In *International Semantic Web Conference*, pages 212–227. Springer, 2014.

[94] Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28, 2017.

[95] Martin Leinberger, Ralf Lämmel, and Steffen Staab. The essence of functional programming on semantic data. In *European Symposium on Programming*, pages 750–776. Springer, 2017.

[96] Sean Bechhofer, Raphael Volz, and Phillip Lord. Cooking the semantic web with the owl api. In *International Semantic Web Conference*, pages 659–675. Springer, 2003.

[97] Patrick Doherty, Witold Lukaszewicz, and Andrzej Szalas. Efficient reasoning using the local closed-world assumption. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 49–58. Springer, 2000.

[98] Rob Shearer, Boris Motik, and Ian Horrocks. Hermit: A highly-efficient owl reasoner. In *OWLED*, volume 432, page 91, 2008.

[99] Raymond Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1-2):81–132, 1980.

[100] Matthias Knorr, José Júlio Alferes, and Pascal Hitzler. Local closed-world reasoning with description logics under the well-founded semantics. *Artificial Intelligence*, 175:1528, 2011.

[101] Chintan Patel, James Cimino, Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Aaron Kershenbaum, Li Ma, Edith Schonberg, and Kavitha Srinivas. Matching patient records to clinical trials using ontologies. In *The Semantic Web*, pages 816–829. Springer, 2007.

[102] Fernando Silva Parreiras, Carsten Saathoff, Tobias Walter, Thomas Franz, and Steffen Staab. Apis a gogo: Automatic generation of ontology apis. In *Semantic Computing, 2009. ICSC'09. IEEE International Conference on*, pages 342–348. IEEE, 2009.

[103] Sohaila Baset and Kilian Stoffel. Ontojit: Parsing native owl dl into executable ontologies in an object oriented paradigm. In *OWL: Experiences and Directions–Reasoner Evaluation*, pages 1–14. Springer, 2016.

[104] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79, 2004.

[105] Panos Alexopoulos, Xanthi Benetou, Tassos Tagaris, Panos Georgolios, and Kostas Kafentzis. Iwebcare: An ontological approach for fraud detection in the healthcare domain. In *International Conference on Information Technologies*, 2007.

[106] Daniel Oberle. How ontologies benefit enterprise applications. *Semantic Web*, 5(6):473–491, 2014.

[107] Elena Paslaru Bontas Simperl, Christoph Tempich, and Malgorzata Mochol. Cost estimation for ontology development: applying the ontocom model. In *Technologies for Business Information Systems*, pages 327–339. Springer, 2007.

[108] Pattie Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987.

[109] Herbert Schildt. *C# 4.0: the complete reference*. Tata McGraw-Hill Education, 2010.

[110] Haruhiko Kaiya and Motoshi Saeki. Using domain ontology as domain knowledge for requirements elicitation. In *Requirements Engineering, 14th IEEE International Conference*, pages 189–198. IEEE, 2006.

[111] Karin K Breitman and Julio Cesar Sampaio do Prado Leite. Ontology as a requirements engineering product. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 309–319. IEEE, 2003.

[112] Diego Dermeval, Jéssyka Vilela, Ig Ibert Bittencourt, Jaelson Castro, Seiji Isotani, Patrick Brito, and Alan Silva. Applications of ontologies in requirements engineering: a systematic review of the literature. *Requirements Engineering*, 21(4):405–437, 2016.

Appendix A

Code Snippets from the PizzaFinder Experiment

```
 1  using OntoJIT.DynamicCode;
 2  using OntoJIT.OwlMapping;
 3  using OntoJIT.Parsers;
 4  using OntoJIT.Reasoning;
 5  using System;
 6  using System.Collections.Generic;
 7  using System.IO;
 8  using System.Linq;
 9  using System.Reflection;
10  using System.Threading;
11
12
13  namespace OntoJIT
14  {
15      class PizzaFinderExperiment
16      {
17
18          static void Main(string[] args)
19          {
20              int stackSize = 1024 * 1024 * 2; // increase stack size
21
22              SetupConsoleOutput(); // Redirect console output to a log file.
23
24              Thread th = new Thread(() => // Run the experiment in a separate
                  thread
25              {
26                  RunExperiment();
27                  Console.WriteLine("Press any key to exit.");
28                  Console.ReadKey();
29              }, stackSize);
30              th.Start();
31              th.Join();
32          }
33
34
35          static void RunExperiment()
36          {
37              Type[] allTypes=Assembly.GetAssembly(typeof(Thing)).GetTypes();
38              var watch = System.Diagnostics.Stopwatch.StartNew();
39              IEnumerable<Type> types;
40              Random random = new Random();
41              double elapsedTime = 0;
42              int noRepititions = 10000;
43
44              List<string> poolOfChoices = new List<string>() {
45                        "MozzarellaTopping", "HotGreenPepperTopping",
                        "PizzaTopping",
46                        "FishTopping", "AnchoviesTopping", "VegetableTopping",
47                        "ArtichokeTopping", "AsparagusTopping", "HerbSpiceTopping",
```

```
48                      "CajunSpiceTopping", "CaperTopping", "GoatsCheeseTopping",
49                      "CheeseTopping", "CheeseyVegetableTopping", "MeatTopping",
50                      "ChickenTopping", "GarlicTopping", "FourCheesesTopping",
51                      "FruitTopping", "LeekTopping", "GorgonzolaTopping",
52                      "PepperTopping", "GreenPepperTopping", "HamTopping",
53                      "HotSpicedBeefTopping", "JalapenoPepperTopping",
54                      "MixedSeafoodTopping", "MushroomTopping", "NutTopping",
55                      "OliveTopping", "OnionTopping", "ParmaHamTopping",
56                      "ParmesanTopping", "PeperonataTopping",
                    "PeperoniSausageTopping",
57                      "PetitPoisTopping", "PrawnsTopping", "RedOnionTopping",
                    "RocketTopping",
58                      "RosemaryTopping", "SauceTopping", "TomatoTopping",
59                      "SlicedTomatoTopping", "SpicyTopping", "SpinachTopping",
60                      "SultanaTopping", "SundriedTomatoTopping",
                    "SweetPepperTopping", "VegetarianTopping" };
61              IList<string> includedToppings = new List<string>();
62              IList<string> excludedToppings = new List<string>();
63              for (int i = 1; i<= 5; i++)
64              {
65                  watch.Restart();
66                  elapsedTime = 0;
67                  for (int j = 0; j < noRepititions; j++)
68                  {
69
70                      includedToppings = GetTransitiveClosure(poolOfChoices.OrderBy
                        (item => random.Next()).Take(i).ToList());
71                      excludedToppings = GetTransitiveClosure(poolOfChoices.OrderBy
                        (item => random.Next()).Take(i).ToList());
72                      int minIntersectCount = includedToppings.Count();
73                      Console.WriteLine("\n Toppings to include:");
74                      includedToppings.ToList().ForEach(t => Console.Write
                        (string.Format(" {0} ", t)));
75                      Console.WriteLine("\n Toppings to Exclude:");
76                      excludedToppings.ToList().ForEach(t => Console.Write
                        (string.Format(" {0} ", t)));
77                      watch.Start();
78                      types = FindPizza(includedToppings, excludedToppings);
79                      watch.Stop();
80                      elapsedTime += watch.ElapsedMilliseconds;
81                      Console.WriteLine(string.Format("\n {0} suggested
                        Pizzas",types.Count()));
82                      types.ToList().ForEach(t => Console.Write(string.Format(" {0}
                          ", t.Name)));
83
84                  }
85
86                  Console.WriteLine(string.Format("\n Finished answering {0}-term
                      queries. Average query execution time was {1} milliseconds", i,
```

```
                    (elapsedTime /noRepititions)));
 87             }
 88         }
 89
 90
 91         private static void SetupConsoleOutput()
 92         {
 93             FileStream outputStream;
 94             StreamWriter writer;
 95             try
 96             {
 97                 outputStream = new FileStream("./log.txt", FileMode.OpenOrCreate,
                       FileAccess.Write);
 98                 writer = new StreamWriter(outputStream);
 99             }
100             catch (Exception e)
101             {
102                 Console.WriteLine("Cannot open log.txt for writing");
103                 Console.WriteLine(e.Message);
104                 return;
105             }
106             Console.SetOut(writer);
107         }
108         private static List<Type> FindPizza(IList<string> includedToppings,
              IList<string> excludedToppings)
109         {
110
111
112             int minIntersectCount = includedToppings.Count();
113
114             var pizzas = from type in
115                          Assembly.GetAssembly(typeof(Thing)).GetTypes()
116                          let property = type.GetField("hasTopping",
                                BindingFlags.Public | BindingFlags.Static |
                                BindingFlags.FlattenHierarchy)
117                          let value = (ObjectProperty)property.GetValue(null)
118
119                           where type.IsClass
120                                 && property != null
121                                 && value.someValuesFrom != null
122                                 && includedToppings.Intersect(UnwrapPropertyValue
                          (value.someValuesFrom)).Count() >= minIntersectCount
123                                 && excludedToppings.Intersect(UnwrapPropertyValue
                          (value.someValuesFrom)).Count() == 0
124                          select type;
125
126             return pizzas.ToList();
127         }
128
```

```
129         static IList<string> GetTransitiveClosure(IList<string> set)
130         {
131
132             Queue<string> queue = new Queue<string>();
133             set.ToList().ForEach(item => queue.Enqueue(item));
134             IList<string> closure = new List<string>();
135             string type = null;
136             while (queue.Count > 0)
137             {
138                 type = queue.Dequeue();
139                 closure.Add(type);
140                 FindSubTypes(type).ToList().ForEach(t => queue.Enqueue(t));
141                 FindEquivalentTypes(type).ToList().ForEach(t => closure.Add(t));
142             }
143             return closure;
144         }
145
146         static  private IList<string> FindSubTypes(string typeName)
147         {
148             var types = from type in
149             Assembly.GetAssembly(typeof(Thing)).GetTypes()
150
151                         where type.BaseType.Name == typeName
152                         select type.Name;
153
154             return types.ToList();
155         }
156
157         static private IList<string> FindEquivalentTypes(string typeName)
158         {
159
160             Type selectedType = Assembly.GetAssembly(typeof(Thing)).GetTypes
                  ().Where(t => t.Name == typeName).FirstOrDefault();
161             FieldInfo property = selectedType.GetField
                  (OwlVocabulary.EquivalentClass, BindingFlags.Public |
                  BindingFlags.Static | BindingFlags.FlattenHierarchy);
162             var value = (property != null ? property.GetValue(null) : null);
163             IList<string> EquivalentTypes = (value == null ? new List<string>() :
                   (value.ToString().Split(';').ToList()));
164
165
166             var types = Assembly.GetAssembly(typeof(Thing))
167                         .GetTypes().Where(t => EquivalentTypes.Intersect(new
                        List<string>() { t.Name }).Count() > 0)
168                         .Select(t => t.Name).ToList();
169
170             foreach (string type in types.Clone())
171             {
172                 if (type.StartsWith("Blank"))
```

```
173                 {
174                     types.Remove(type);
175                     types.AddRange(DigestBlankNode(type));
176                 }
177             }
178 
179             return types.ToList();
180         }
181 
182         private static IList<string> DigestBlankNode(string typeName)
183         {
184             ObjectProperty owlProperty;
185 
186             Type selectedType = Assembly.GetAssembly(typeof(Thing)).GetTypes
                  ().Where(t => t.Name == typeName).FirstOrDefault();
187             List<FieldInfo> fields = selectedType.GetFields(BindingFlags.Public |
                   BindingFlags.Static).ToList();
188             List<string> aggregatedConcepts = new List<string>();
189             foreach (FieldInfo field in fields)
190             {
191                 if (field.Name == "Aggregator")
192                 {
193                     aggregatedConcepts = fields.Where(f => f.Name ==
                      "CollectionItems").FirstOrDefault().GetValue(null).ToString
                      ().Split(';').ToList();
194                     aggregatedConcepts.RemoveAll(x => x == "");
195                     foreach (string type in aggregatedConcepts.Clone())
196                     {
197                         if (type.StartsWith("Blank"))
198                         {
199                             aggregatedConcepts.Remove(type);
200                             aggregatedConcepts.AddRange(DigestBlankNode(type));
201                         }
202                     }
203                 }
204                 else
205                 {
206                         owlProperty = (ObjectProperty)field.GetValue(null);
207                         var matchingClasses = from type in
208                          Assembly.GetAssembly(typeof(Thing)).GetTypes()
209                                               let property = type.GetField
                      (field.Name, BindingFlags.Public | BindingFlags.Static |
                      BindingFlags.FlattenHierarchy)
210                                               let someValuesFrom = (property !=
                      null ? ((ObjectProperty)property.GetValue
                      (null)).someValuesFrom.ToString() : null)
211                                               where type.IsClass
212                                                       && someValuesFrom ==
                      ((ObjectProperty)field.GetValue
```

```
                        (null)).someValuesFrom.ToString()
213                                                 select type.Name;
214
215                         aggregatedConcepts.AddRange(matchingClasses.ToList());
216                         break;
217                 }
218             }
219             return aggregatedConcepts;
220         }
221
222         private static IList<string> UnwrapPropertyValue(object propertyValue)
223         {
224             if (propertyValue == null)
225                 return new List<string>();
226             var splitedTypes = propertyValue.ToString().Split(';').ToList();
227             splitedTypes.RemoveAll(x => x == "");
228
229             foreach (string type in splitedTypes.Clone())
230             {
231                 if (type.StartsWith("Blank"))
232                 {
233                     splitedTypes.Remove(type);
234                     splitedTypes.AddRange(DigestBlankNode(type));
235                 }
236             }
237             return splitedTypes;
238
239         }
240
241
242     }
243
244
245
246 }
247
```

Appendix B

OntoJIT Parser Code Snippets

B.1 OntoJIT wrapper for CodeDomProvider

```
 1  using Microsoft.CSharp;
 2  using System;
 3  using System.CodeDom;
 4  using System.CodeDom.Compiler;
 5  using System.Collections.Generic;
 6  using System.IO;
 7  using System.Linq;
 8  using System.Data;
 9  using System.Reflection;
10  using System.Linq.Expressions;
11  using OntoJIT.OwlMapping;
12
13  namespace OntoJIT.DynamicCode
14  {
15      public class CodeDomWrapper
16      {
17          #region variables
18          CodeCompileUnit _compileUnit = null;
19          CodeNamespace _nameSpace = null;
20          #endregion
21
22          #region Singleton Pattern
23          private static CodeDomWrapper instance = null;
24          private static readonly object padlock = new object();
25          CodeDomWrapper()
26          {
27              Initialize();
28          }
29          public static CodeDomWrapper Instance
30          {
31              get
32              {
33                  lock (padlock)
34                  {
35                      if (instance == null)
36                      {
37                          instance = new CodeDomWrapper();
38                      }
39                      return instance;
40                  }
41              }
42          }
43
44          public object ImplementationTypes { get; private set; }
45
46          void Initialize()
47          {
48              _compileUnit = new CodeCompileUnit();
49              _nameSpace = new CodeNamespace("OntoJIT");
```

```
50              _nameSpace.Imports.Add(new CodeNamespaceImport("System"));
51              _compileUnit.Namespaces.Add(_nameSpace);
52              AddEntryPoint();
53          }
54          #endregion
55
56          #region Public Interface
57
58
59          public void GenerateCSharpCode(string fileName)
60          {
61              CodeDomProvider provider = CodeDomProvider.CreateProvider("CSharp");
62              CodeGeneratorOptions options = new CodeGeneratorOptions();
63              options.BracingStyle = "C";
64              using (StreamWriter sourceWriter = new StreamWriter(fileName))
65              {
66                  provider.GenerateCodeFromCompileUnit(
67                      _compileUnit, sourceWriter, options);
68              }
69          }
70
71          public bool CompileCSharpCode(string sourceFile, string dllFile)
72          {
73              CSharpCodeProvider provider = new CSharpCodeProvider();
74
75              // Build the parameters for source compilation.
76              CompilerParameters cp = new CompilerParameters();
77
78              // Add an assembly reference.
79              cp.ReferencedAssemblies.Add("System.dll");
80
81              // Generate an executable instead of
82              // a class library.
83              cp.GenerateExecutable = false;
84
85              // Set the assembly file name to generate.
86              cp.OutputAssembly = dllFile;
87
88              // Save the assembly as a physical file.
89              cp.GenerateInMemory = false;
90
91              // Invoke compilation.
92              CompilerResults cr = provider.CompileAssemblyFromFile(cp,
                  sourceFile);
93
94              if (cr.Errors.Count > 0)
95              {
96                  // Display compilation errors.
97                  Console.WriteLine("Errors building {0} into {1}",
```

```
 98                     sourceFile, cr.PathToAssembly);
 99                 foreach (CompilerError ce in cr.Errors)
100                 {
101                     Console.WriteLine("  {0}", ce.ToString());
102                     Console.WriteLine();
103                 }
104             }
105             else
106             {
107                 Console.WriteLine("Source {0} built into {1} successfully.",
108                     sourceFile, cr.PathToAssembly);
109             }
110 
111             return true;
112         }
113 
114         public bool CompileCSharpCode(CodeCompileUnit ccu)
115         {
116             CSharpCodeProvider provider = new CSharpCodeProvider();
117 
118             // Build the parameters for source compilation.
119             CompilerParameters cp = new CompilerParameters();
120 
121             // Add an assembly reference.
122             cp.ReferencedAssemblies.Add("System.dll");
123 
124             // Generate an executable instead of
125             // a class library.
126             cp.GenerateExecutable = false;
127 
128             // Set the assembly file name to generate.
129             cp.OutputAssembly = "OntoJITAssembly.dll";
130 
131             // Save the assembly as a physical file.
132             cp.GenerateInMemory = false;
133 
134             // Invoke compilation.
135             CompilerResults cr = provider.CompileAssemblyFromDom(cp, ccu);
136 
137             if (cr.Errors.Count > 0)
138             {
139                 // Display compilation errors.
140                 Console.WriteLine("Errors building {0} into {1}",
141                     "", cr.PathToAssembly);
142                 foreach (CompilerError ce in cr.Errors)
143                 {
144                     Console.WriteLine("  {0}", ce.ToString());
145                     Console.WriteLine();
146                 }
```

```
147                 return false;
148             }
149             else
150             {
151                 Console.WriteLine("Source {0} built into {1} successfully.",
152                     "", cr.PathToAssembly);
153             }
154 
155 
156             return true;
157         }
158 
159         public string AddPublicClass(string className, string parent = null, bool
               createConstructor = false)
160         {
161             if (className == null)
162                 className = GenerateClassName();
163             CodeTypeDeclaration createdType = AddTypeDeclaration(className,
                  parent, TypeAttributes.Public | TypeAttributes.Class,
                  createConstructor);
164             return createdType.Name;
165         }
166 
167         public CodeTypeDeclaration AddPublicStaticClass(string className, string
              parent = null)
168         {
169             return AddTypeDeclaration(className, parent, TypeAttributes.Class |
                  TypeAttributes.Public | TypeAttributes.Sealed);
170         }
171 
172         public void AddNameSpace(string nsName)
173         {
174             if (!NamespaceExists(nsName))
175             {
176                 _compileUnit.Namespaces.Add(new CodeNamespace(nsName));
177             }
178         }
179 
180         public void AddLambdaExpression(string className, object expression)
181         {
182             CodeTypeDeclaration targetClass =
                  _nameSpace.Types.Cast<CodeTypeDeclaration>().Where(t => t.Name ==
                  className).FirstOrDefault();
183 
184             Expression<Func<Type, bool>> expr = x => x.IsInterface;
185             string s = expr.ToString();
186             int count = targetClass.Members.Cast<CodeTypeMember>().Where(m =>
                  m.Name.StartsWith("LambdaExp")).Count();
187             CodeTypeReference type = new CodeTypeReference(typeof(object));
```

```
188             CodeMemberField field = new CodeMemberField(type, string.Format
                  ("LambdaExp{0}", ++count));
189             field.Attributes = MemberAttributes.Public | MemberAttributes.Static;
190             field.InitExpression = new CodePrimitiveExpression(expr.ToString());
191             targetClass.Members.Add(field);
192
193
194
195         }
196
197         public void AddNamespaceProperty(string className, string propertyName,
              string propertyValue)
198         {
199             Type type = Type.GetType("System.String");
200             SetProperty(className, propertyName, propertyValue);
201         }
202
203         public bool TypeExists(string typeName)
204         {
205             return (_nameSpace.Types.Cast<CodeTypeDeclaration>().Where(t =>
                  t.Name == typeName).Count() != 0);
206         }
207
208         public bool MetaDataExists(string typeName)
209         {
210             CodeTypeDeclaration targetClass =
                  _nameSpace.Types.Cast<CodeTypeDeclaration>().Where(t => t.Name ==
                  typeName).FirstOrDefault();
211             return targetClass.Members.Count > 1;
212         }
213
214         public void AddEntryPoint()
215         {
216             CodeEntryPointMethod start = new CodeEntryPointMethod();
217             CodeTypeDeclaration entryClass = AddPublicStaticClass("Program");
218             entryClass.Members.Add(start);
219         }
220
221         public void UpdateBaseType(string className, string newParent)
222         {
223
224             CodeTypeDeclaration targetClass = FindType(className);
225             if (newParent.Contains("#"))
226             {
227                 newParent = newParent.Substring(newParent.IndexOf('#') + 1);
228             }
229             if (targetClass == null)
230                 AddTypeDeclaration(className, newParent, TypeAttributes.Public);
231             else
```

```
232             {
233                 CodeTypeDeclaration newParentClass = FindType(newParent);
234                 List<CodeTypeReference> grandParents = new
                      List<CodeTypeReference>();
235                 GetMutualParents(targetClass, newParentClass, ref grandParents);
236 
237                 if (grandParents.Count == 0) // a multiple inheritance case
238                 {
239                     SetProperty(className, OwlVocabulary.subClassOf, newParent);
240                     return;
241                 }
242 
243                 foreach (CodeTypeReference grandParent in grandParents.ToList())
244                     targetClass.BaseTypes.Remove(grandParent);
245                 if (newParent.Contains('#'))
246                     newParent = newParent.Substring(newParent.IndexOf('#') + 1);
247                 targetClass.BaseTypes.Add(newParent);
248 
249             }
250 
251         }
252 
253         public CodeTypeDeclaration AddTypeDeclaration(string className, string
              parent, TypeAttributes attributes, bool createParameterConstructor =
              false)
254         {
255 
256             if (className.Contains("#"))
257             {
258                 className = className.Substring(className.IndexOf('#') + 1);
259             }
260             CodeTypeDeclaration targetClass =
                  _nameSpace.Types.Cast<CodeTypeDeclaration>().Where(t => t.Name ==
                  className).FirstOrDefault();
261             if (targetClass == null)
262             {
263                 targetClass = new CodeTypeDeclaration(className);
264                 AddConstructor(targetClass);
265                 if (createParameterConstructor)
266                     AddConstructor(targetClass, true);
267                 if (parent != null)
268                 {
269                     CodeTypeReference reference = new CodeTypeReference(parent);
270                     reference.UserData.Add("TypeName", parent);
271                     targetClass.BaseTypes.Add(reference);
272 
273                 }
274                 targetClass.TypeAttributes = attributes;
275                 _nameSpace.Types.Add(targetClass);
```

```
276             }
277             return targetClass;
278         }
279 
280 
281         public IList<CodeTypeDeclaration> GetTypes(string startsWith)
282         {
283             return _nameSpace.Types.Cast<CodeTypeDeclaration>().Where(t =>
                  t.Name.StartsWith(startsWith)).ToList();
284         }
285         #endregion
286 
287         #region private mathods
288 
289         private void GetMutualParents(CodeTypeDeclaration x, CodeTypeDeclaration
              y, ref List<CodeTypeReference> detected)
290         {
291             if (y.Name == "Thing" || y.Name == "OwlProperty")
292                 return;
293             else
294             {
295                 detected.AddRange(x.BaseTypes.Cast<CodeTypeReference>().Intersect
                      (y.BaseTypes.Cast<CodeTypeReference>(), new
                      TypeReferenceComparer()).ToList());
296                 foreach (CodeTypeReference rf in y.BaseTypes)
297                 {
298                     CodeTypeDeclaration yParent = FindType(rf.BaseType);
299                     GetMutualParents(x, yParent, ref detected);
300                 }
301             }
302         }
303 
304         public void SetProperty(string className, string propertyName, object
              propertyValue = null, bool isStatic = true, bool innerLevel = true)
305         {
306             CodeTypeDeclaration targetClass = FindType(className);
307             if (targetClass == null)
308             {
309                 targetClass = AddTypeDeclaration(className,
                      OwlVocabulary.ObjectProperty, TypeAttributes.Public);
310             }
311             CodeMemberField targetProperty = (CodeMemberField)
                  targetClass.Members.Cast<CodeTypeMember>().Where(p => p.Name ==
                  propertyName).FirstOrDefault();
312             if (propertyValue != null)
313                 if (propertyValue.ToString().Contains('#'))
314                 {
315                     string str = propertyValue.ToString();
316                     propertyValue = str.Substring(str.IndexOf('#') + 1);
```

```
317                 }
318             if (targetProperty == null)
319             {
320 
321                 targetProperty = new CodeMemberField();
322                 targetProperty.Name = propertyName;
323 
324                 targetProperty.Attributes = 0 | MemberAttributes.Public;
325                 if (isStatic)
326                     targetProperty.Attributes = targetProperty.Attributes |
                        MemberAttributes.Static;
327                 if (innerLevel)
328                     targetProperty.Attributes = targetProperty.Attributes |
                        MemberAttributes.New;
329                 targetClass.Members.Add(targetProperty);
330 
331             }
332             CodeTypeDeclaration propertyType = FindType(propertyName);
333             if (propertyType == null)
334                 targetProperty.Type = new CodeTypeReference(typeof(object));
335             else
336                 targetProperty.Type = new CodeTypeReference(propertyType.Name);
337             if (propertyValue != null)
338                     // type constructor
339                 UpdateStaticConstructor(targetClass, propertyName,
                        propertyValue);
340 
341         }
342 
343         public void AppendPropertyList(string className, string propertyName,
              object listItem)
344         {
345             CodeTypeDeclaration targetClass = FindType(className);
346             CodeMemberField targetProperty = (CodeMemberField)
                  targetClass.Members.Cast<CodeTypeMember>().Where(p => p.Name ==
                  propertyName).FirstOrDefault();
347 
348             if (targetProperty == null)
349             {
350                 SetProperty(className, propertyName, listItem);
351                 return;
352             }
353             CodeTypeDeclaration propertyType = FindType(propertyName);
354             if (propertyType != null)
355             {
356                 targetProperty.Type = new CodeTypeReference(propertyType.Name);
357                 CodeExpression leftSide = new CodeFieldReferenceExpression(null,
                      propertyName);
358 
```

```
359             }
360             else
361             {
362                 targetProperty.Type = new CodeTypeReference(typeof(object));
363             }
364 
365         }
366 
367         public void SetNestedProperty(string className, string
              parentPropertyName, string childPropertyName, object
              childPropertyValue)
368         {
369             if (childPropertyValue.ToString().Contains('#'))
370             {
371                 if (childPropertyValue.ToString().Count(c => c == '#') > 1)
372                     childPropertyValue = childPropertyValue.ToString();
373                 string str = childPropertyValue.ToString();
374                 childPropertyValue = str.Substring(str.IndexOf('#') + 1);
375             }
376 
377             CodeTypeDeclaration targetClass = FindType(className);
378             CodeMemberField parentProperty = (CodeMemberField)
                  targetClass.Members.Cast<CodeTypeMember>().Where(p => p.Name ==
                  parentPropertyName).FirstOrDefault();
379             if (parentProperty == null)
380             {
381                 SetProperty(className, parentPropertyName);
382                 parentProperty = (CodeMemberField)
                      targetClass.Members.Cast<CodeTypeMember>().Where(p => p.Name ==
                       parentPropertyName).FirstOrDefault();
383             }
384             // Static Constructor
385             UpdateStaticConstructor(targetClass, parentPropertyName,
                  childPropertyName, childPropertyValue);
386 
387         }
388 
389         private void AddConstructor(CodeTypeDeclaration targetClass, bool
              hasParameter = false)
390         {
391             // Declare the constructor
392             CodeConstructor constructor = new CodeConstructor();
393             constructor.Attributes =
394                 MemberAttributes.Public;
395             if (hasParameter)
396             {
397                 constructor.Parameters.Add(new CodeParameterDeclarationExpression
                      (typeof(object), "Value"));
398                 constructor.Statements.Add(new CodeAssignStatement(
```

```
399                     new CodeFieldReferenceExpression(new
                        CodeThisReferenceExpression(), "Value"), new
                        CodeFieldReferenceExpression(null, "Value")));
400             }
401             targetClass.Members.Add(constructor);
402         }
403
404         private void AddStaticConstructor(CodeTypeDeclaration targetClass)
405         {
406             // Declare the constructor
407             CodeTypeConstructor constructor = new CodeTypeConstructor();
408             constructor.Attributes = MemberAttributes.Static |
                  MemberAttributes.Public;
409             targetClass.Members.Add(constructor);
410         }
411
412         private void UpdateStaticConstructor(CodeTypeDeclaration targetClass,
              string outerPropertyName, string innerPropertyName, object
              innerPropertyValue)
413         {
414
415             CodeExpression leftSide, rightSide;
416
417             // Find the static constructor if it doesn't exist yet, create one
418             CodeTypeConstructor constructor = (CodeTypeConstructor)
                  targetClass.Members.Cast<CodeTypeMember>().Where(m =>
                  m.Name.EndsWith(".cctor") && ((m.Attributes &
                  MemberAttributes.Static) ==
                  MemberAttributes.Static)).FirstOrDefault();
419             if (constructor == null)
420                 AddStaticConstructor(targetClass);
421             constructor = (CodeTypeConstructor)
                  targetClass.Members.Cast<CodeTypeMember>().Where(m =>
                  m.Name.EndsWith(".cctor") && ((m.Attributes &
                  MemberAttributes.Static) ==
                  MemberAttributes.Static)).FirstOrDefault();
422             List<CodeAssignStatement> oldAssignments =
                  constructor.Statements.Cast<CodeStatement>().Where(s => s.GetType
                  ().Name == "CodeAssignStatement").Cast<CodeAssignStatement>
                  ().ToList();
423
424             var outerPropertyAssignment = oldAssignments.Where(s =>
                  s.Left.GetType().Name == "CodeFieldReferenceExpression" &&
                  ((CodeFieldReferenceExpression)s.Left).FieldName ==
                  outerPropertyName).FirstOrDefault();
425
426             if (outerPropertyAssignment == null)
427             {
428                 leftSide = new CodeFieldReferenceExpression(null,
```

```
                    outerPropertyName);
429                 rightSide = new CodeObjectCreateExpression(new CodeTypeReference
                      (outerPropertyName), new CodeExpression[] { });
430
431                 constructor.Statements.Insert(0, new CodeAssignStatement
                      (leftSide, rightSide));
432             }
433
434
435             var innerPropertyAssignment = oldAssignments.Where(s =>
                  s.Left.GetType().Name == "CodeFieldReferenceExpression" &&
                  ((CodeFieldReferenceExpression)s.Left).FieldName ==
                  innerPropertyName).FirstOrDefault();
436             object oldValue = null;
437             if (innerPropertyAssignment != null)
438             {
439                 oldValue = ((CodePrimitiveExpression)
                      innerPropertyAssignment.Right).Value;
440                 innerPropertyValue = string.Format("{0};{1}", oldValue,
                      innerPropertyValue.ToString());
441                 constructor.Statements.Remove(innerPropertyAssignment);
442             }
443             leftSide = new CodeFieldReferenceExpression(new
                  CodeFieldReferenceExpression(null, outerPropertyName),
                  innerPropertyName);
444             rightSide = new CodePrimitiveExpression(innerPropertyValue);
445
446             constructor.Statements.Insert(constructor.Statements.Count, new
                  CodeAssignStatement(leftSide, rightSide));
447
448         }
449
450         private void UpdateStaticConstructor(CodeTypeDeclaration targetClass,
              string propertyName, object propertyValue)
451         {
452             CodeExpression leftSide, rightSide;
453
454             // Find the static constructor if it doesn't exist yet, create one
455             CodeTypeConstructor constructor = (CodeTypeConstructor)
                  targetClass.Members.Cast<CodeTypeMember>().Where(m =>
                  m.Name.EndsWith(".cctor") && ((m.Attributes &
                  MemberAttributes.Static) ==
                  MemberAttributes.Static)).FirstOrDefault();
456             if (constructor == null)
457                 AddStaticConstructor(targetClass);
458             constructor = (CodeTypeConstructor)
                  targetClass.Members.Cast<CodeTypeMember>().Where(m =>
                  m.Name.EndsWith(".cctor") && ((m.Attributes &
                  MemberAttributes.Static) ==
```

```
                MemberAttributes.Static)).FirstOrDefault();
459             var oldAssignment = constructor.Statements.Cast<CodeStatement>()
460                                             .Where(s => s.GetType().Name ==
                        "CodeAssignStatement")
461                                             .Cast<CodeAssignStatement>()
462                                             .ToList().Where(s => s.Left.GetType
                        ().Name == "CodeFieldReferenceExpression" &&
                        ((CodeFieldReferenceExpression)s.Left).FieldName ==
                        propertyName)
463                                             .FirstOrDefault();
464
465
466             if (oldAssignment != null)
467             {
468                 var oldValue = ((CodePrimitiveExpression)
                      oldAssignment.Right).Value;
469                 propertyValue = string.Format("{0};{1}", oldValue,
                      propertyValue.ToString());
470                 constructor.Statements.Remove(oldAssignment);
471             }
472             leftSide = new CodeFieldReferenceExpression(null, propertyName);
473             // rightSide = new CodeObjectCreateExpression(new CodeTypeReference
                  (propertyName), new CodePrimitiveExpression(propertyValue));
474             rightSide = new CodePrimitiveExpression(propertyValue);
475
476             constructor.Statements.Insert(0, new CodeAssignStatement(leftSide,
                  rightSide));
477
478         }
479
480         private bool NamespaceExists(string nsName)
481         {
482             return (_compileUnit.Namespaces.Cast<CodeNamespace>().Where(ns =>
                  ns.Name == nsName).Count() != 0);
483
484         }
485
486         private CodeTypeDeclaration FindType(string className)
487         {
488            return _nameSpace.Types.Cast<CodeTypeDeclaration>().Where(t => t.Name
                 == className).FirstOrDefault();
489
490         }
491
492
493         private string GenerateClassName()
494         {
495             int count = _nameSpace.Types.Cast<CodeTypeDeclaration>().Where(t =>
                  t.Name.StartsWith("Blank")).Count();
```

```
496                 return string.Format("Blank{0}", count++);
497             }
498             #endregion
499         }
500 
501 }
502 
```

B.2 OWL base

```
 1  using OntoJIT.DynamicCode;
 2  using System;
 3  using System.CodeDom;
 4  using System.Collections;
 5  using System.Collections.Generic;
 6  using System.Linq;
 7  using System.Linq.Expressions;
 8  using System.Text;
 9  using System.Threading.Tasks;
10
11  namespace OntoJIT.OwlMapping
12  {
13      public class OwlBase
14      {
15
16          IDictionary _nameSpaces = new Dictionary<string, string>();
17
18          public IDictionary NameSpaces
19          {
20              get
21              {
22                  return _nameSpaces;
23              }
24
25              set
26              {
27                  _nameSpaces = value;
28              }
29          }
30
31          public OwlBase()
32          {
33              Initialize();
34          }
35
36          void Initialize()
37          {
38              InitializeOwlClassHierarchy();
39              InitializeOwlPropertiesHierarchy();
40          }
41
42          private void InitializeOwlPropertiesHierarchy()
43          {
44              CodeDomWrapper.Instance.AddPublicClass
                  (OwlVocabulary.TopPropertyClassName);
45              CodeDomWrapper.Instance.AddPublicClass
                  (OwlVocabulary.ObjectPropertyClassName,
                  OwlVocabulary.TopPropertyClassName);
46              CodeDomWrapper.Instance.AddPublicClass
```

```
                (OwlVocabulary.DataTypePropertyClassName,
                OwlVocabulary.TopPropertyClassName);
47              CodeDomWrapper.Instance.AddPublicClass
                (OwlVocabulary.AnnotationPropertyClassName,
                OwlVocabulary.TopPropertyClassName);
48
49              AddMetaProperties(OwlVocabulary.TopPropertyClassName);
50              AddMetaProperties(OwlVocabulary.ObjectPropertyClassName);
51              AddMetaProperties(OwlVocabulary.DataTypePropertyClassName);
52              AddMetaProperties(OwlVocabulary.AnnotationPropertyClassName);
53          }
54
55          private static void AddMetaProperties(string className)
56          {
57              bool metaDataExists = CodeDomWrapper.Instance.MetaDataExists
                (className);
58              if (!metaDataExists)
59              {
60                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.IsFunctionalPropertyName, null,innerLevel:false);
61                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.IsTransitivePropertyName, null, innerLevel:
                    false);
62                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.IsReflexivePropertyName, null, innerLevel:
                    false);
63
64                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.AllValuesFrom, null, false, innerLevel: false);
65                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.SomeValuesFrom, null, false, innerLevel: false);
66                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.HasValue, null, false, innerLevel: false);
67
68                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.MinCardinality, null, false, innerLevel: false);
69                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.MaxCardinality, null, false, innerLevel: false);
70                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.Cardinality, null, false, innerLevel: false);
71                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.QualifiedCardinality, null, false, innerLevel:
                    false);
72
73                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.Domain, null, innerLevel: false);
74                  CodeDomWrapper.Instance.SetProperty(className,
                    OwlVocabulary.Range, null, innerLevel: false);
75
```

```
76                  CodeDomWrapper.Instance.SetProperty(className,
                      OwlVocabulary.ValuePropertyName, null, false, innerLevel:
                      false);
77                  CodeDomWrapper.Instance.SetProperty(className,
                      OwlVocabulary.LabelPropertyName, null, false, innerLevel:
                      false);
78
79              }
80          }
81
82          private void InitializeOwlClassHierarchy()
83          {
84              CodeDomWrapper.Instance.AddPublicClass(OwlVocabulary.Concept);
85              CodeDomWrapper.Instance.AddPublicClass(OwlVocabulary.TopClassName,
                  OwlVocabulary.Concept);
86              CodeDomWrapper.Instance.AddPublicClass
                  (OwlVocabulary.BlankNodeClassName, OwlVocabulary.Concept);
87              CodeDomWrapper.Instance.SetProperty(OwlVocabulary.BlankNodeClassName,
                   OwlVocabulary.BlankNodeAggregatePropertyName);
88
89              CodeDomWrapper.Instance.AddPublicStaticClass
                  (OwlVocabulary.NamespaceCollectionClassName);
90              AddClassMetaData(OwlVocabulary.Concept);
91          }
92
93          private static void AddClassMetaData(string className)
94          {
95
96              bool metaDataExists = CodeDomWrapper.Instance.MetaDataExists
                  (className);
97              if (!metaDataExists)
98              {
99                  CodeDomWrapper.Instance.SetProperty(className,
                      OwlVocabulary.EquivalentClass, null);
100                 CodeDomWrapper.Instance.SetProperty(className,
                      OwlVocabulary.DisjointWith, null);
101                 CodeDomWrapper.Instance.SetProperty(className,
                      OwlVocabulary.ComplementOf, null);
102                 CodeDomWrapper.Instance.SetProperty(className,
                      OwlVocabulary.LabelPropertyName, null, false, innerLevel:
                      false);
103
104             }
105         }
106
107         public void MapNameSpaces()
108         {
109             foreach (KeyValuePair<string, string> ns in
                  NameSpaces.Cast<KeyValuePair<string, string>>())
```

```
110                 CodeDomWrapper.Instance.AddNamespaceProperty
                      (OwlVocabulary.NamespaceCollectionClassName, ns.Key, ns.Value);
111
112         }
113
114         public void SetNameSpaceValue(KeyValuePair<string, string> ns)
115         {
116             CodeDomWrapper.Instance.AddNamespaceProperty
                  (OwlVocabulary.NamespaceCollectionClassName, ns.Key, ns.Value);
117         }
118
119         public void AddType(string typeName, string parentName = null)
120         {
121             if (parentName == null)
122                 parentName = OwlVocabulary.TopClassName;
123             CodeDomWrapper.Instance.AddPublicClass(typeName, parentName);
124         }
125
126         public void AddObjectProperty(string propertyName, string
              parentPropertyName = null, bool hasConstructor = false)
127         {
128             if (parentPropertyName == null)
129                 parentPropertyName = OwlVocabulary.ObjectPropertyClassName;
130             CodeDomWrapper.Instance.AddPublicClass(propertyName,
                  parentPropertyName, hasConstructor);
131         }
132
133         public void AddDataTypeProperty(string propertyName, string
              parentPropertyName = null)
134         {
135             if (parentPropertyName == null)
136                 parentPropertyName = OwlVocabulary.DataTypePropertyClassName;
137             CodeDomWrapper.Instance.AddPublicClass(propertyName,
                  parentPropertyName);
138         }
139
140         public void AddOntology(string ontologyName)
141         {
142            CodeDomWrapper.Instance.AddNameSpace(ontologyName);
143         }
144
145         public void UpdateClassHeirarchy(string className, string
              parentClassName)
146         {
147             CodeDomWrapper.Instance.UpdateBaseType(className, parentClassName);
148         }
149
150         public void UpdatePropertyValue(string className, string propertyName,
              object propertyValue)
```

```
151         {
152             CodeDomWrapper.Instance.SetProperty(className, propertyName,
                  propertyValue);
153         }
154
155         public bool TypeExists(string typeName)
156         { return CodeDomWrapper.Instance.TypeExists(typeName); }
157
158         internal void AddRestriction(string restriction, string className, string
               propertyName, object propertyFiller, string owlTerm = null)
159         {
160             string filler = null;
161             if (owlTerm != null) // blank node detected
162             {
163                 filler = AddBlankNodeType(owlTerm, (List<string>)propertyFiller);
164             }
165             else
166                 filler = propertyFiller.ToString();
167             if (!CodeDomWrapper.Instance.TypeExists(propertyName))
168                 CodeDomWrapper.Instance.AddPublicClass(propertyName,
                      OwlVocabulary.ObjectPropertyClassName,true);
169             CodeDomWrapper.Instance.SetNestedProperty(className, propertyName,
                  restriction, filler);
170
171         }
172         public string AddBlankNodeType(string owlTerm, List<string> items)
173         {
174
175             string blankTypeName = CodeDomWrapper.Instance.AddPublicClass(null,
                  OwlVocabulary.BlankNodeClassName);
176
177             if (owlTerm != null)
178             {
179                 CodeDomWrapper.Instance.SetProperty(blankTypeName,
                      OwlVocabulary.BlankNodeAggregatePropertyName, owlTerm);
180                 string serliazedValue = SerializeList(items);
181                 CodeDomWrapper.Instance.SetProperty(blankTypeName,
                      OwlVocabulary.BlankNodeCollectionPropertyName, serliazedValue);
182             }
183             return blankTypeName;
184         }
185         public void AddInverseOf(string className, object propertyFiller, string
              owlTerm = null)
186         {
187             string filler = null;
188             if (owlTerm != null) // blank node detected
189             {
190                 filler = AddBlankNodeType(owlTerm, (List<string>)propertyFiller);
191             }
```

```
192             else
193                 filler = propertyFiller.ToString();
194 
195             CodeDomWrapper.Instance.SetProperty(className,
                  OwlVocabulary.InverseOf, filler);
196         }
197 
198         internal void AddEquivelantClass(string className, string equivelantName)
199         {
200             CodeDomWrapper.Instance.AppendPropertyList(className,
                  OwlVocabulary.EquivalentClass, equivelantName);
201         }
202         internal void AddDisjointClass(string className, string
              disjointClassName)
203         {
204 
205             CodeDomWrapper.Instance.AppendPropertyList
                  (className,OwlVocabulary.DisjointWith, disjointClassName);
206         }
207         internal void AddComplementOfClass(string className, string
              ComplementClassName)
208         {
209 
210             CodeDomWrapper.Instance.AppendPropertyList(className,
                  OwlVocabulary.ComplementOf, ComplementClassName);
211         }
212 
213         private string SerializeList(List<string> list)
214         {
215             string serializedString = null;
216             foreach (string entry in list)
217             {
218                 serializedString += string.Format("{0};", entry);
219             }
220             return serializedString;
221         }
222 
223         internal void SetLabelProperty(string className, string label)
224         {
225             CodeDomWrapper.Instance.SetProperty(className,
                  OwlVocabulary.LabelPropertyName, label);
226         }
227     }
228 }
229 
```

B.3 OntoJIT XML parser

```
using System;
using System.Xml;
using System.Collections;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using OntoJIT.OwlMapping;
using System.Text.RegularExpressions;

namespace OntoJIT.Parsers
{
    public class OwlXmlParser : OwlParser, IOwlXmlParser
    {
        OwlBase _owlBase = null;
        int nodeCounter = 0;

        #region Variables
        /// <summary>
        /// A Hashtable that stores declared rdf:ID values for quick lookup of duplicates
        /// </summary>
        private Hashtable _declID;

        /// <summary>
        /// A Collection of non syntactic elements in the OWL syntax
        /// </summary>
        private Hashtable _nonSyntacticElements;

        /// <summary>
        /// A Collection of syntactic elements in the OWL syntax
        /// </summary>
        private Hashtable _syntacticElements;

        /// <summary>
        /// A Collection of all RDF and XML properties
        /// </summary>
        /// <remarks>The members of this collection are rdf:about, rdf:resource, rdf:parseType, rdf:ID, rdf:nodeID, rdf:datatype, rdf:value, xml:lang, xml:base</remarks>
        private Hashtable _rdfXmlProperties;
        private readonly int lengthThreshold;

        #endregion

        #region voc
        /// <summary>
        /// Initializes an instance of the RdfParser class
        /// </summary>
        public OwlXmlParser()
```

```
47          {
48              _warnings = new ArrayList();
49              _errors = new ArrayList();
50              _messages = new ArrayList();
51
52
53              string[] rdfXmlAttrs = { "rdf:about", "rdf:resource",
                  "rdf:parseType", "rdf:ID", "rdf:nodeID", "rdf:type",
                  "rdf:datatype", "rdf:value", "xml:lang", "xml:base" };
54              _rdfXmlProperties = new Hashtable();
55              int len = rdfXmlAttrs.Length;
56              for (int i = 0; i < len; i++)
57                  _rdfXmlProperties.Add(rdfXmlAttrs[i], rdfXmlAttrs[i]);
58
59              string[] syntacticElements = { "rdf:RDF", "rdf:ID", "rdf:about",
                  "rdf:resource", "rdf:parseType", "rdf:nodeID" };
60              _syntacticElements = new Hashtable();
61              len = syntacticElements.Length;
62              for (int i = 0; i < len; i++)
63                  _syntacticElements.Add(syntacticElements[i], syntacticElements
                      [i]);
64
65              string[] nonSyntacticElements = { "rdf:type", "rdf:value",
                  "rdf:datatype", "rdf:List", "rdf:first", "rdf:rest", "rdf:nil",
                  "rdfs:comment", "rdfs:subPropertyOf", "rdfs:domain", "rdfs:range",
                   "rdfs:subClassOf", "owl:allValuesFrom",
                  "owl:backwardCompatibleWith", "owl:cardinality",
                  "owl:complementOf", "owl:differentFrom", "owl:disjointWith",
                  "owl:distinctMembers", "owl:equivalentClass",
                  "owl:equivalentProperty", "owl:hasValue", "owl:imports",
                  "owl:incompatibleWith", "owl:intersectionOf", "owl:inverseOf",
                  "owl:maxCardinality", "owl:minCardinality", "owl:oneOf",
                  "owl:onProperty", "owl:priorVersion", "owl:sameAs",
                  "owl:someValuesFrom", "owl:unionOf", "owl:versionInfo" };
66              _nonSyntacticElements = new Hashtable();
67              len = nonSyntacticElements.Length;
68              for (int i = 0; i < len; i++)
69                  _nonSyntacticElements.Add(nonSyntacticElements[i],
                      nonSyntacticElements[i]);
70
71              _declID = new Hashtable();
72
73              StopOnErrors = false;
74              StopOnWarnings = false;
75
76          }
77
78          #endregion
79
```

```
80          #region xmlParsers
81
82          #region methodOverloads
83
84          public void ParseOwl(XmlDocument doc)
85          {
86              //parses from the xml document
87              //if doc is null throws an ArgumentNullException
88              //looks for xml:base in doc
89              //if xml:base is not found then defaults to http://unknown.org/
90              if (doc == null)
91                  throw (new ArgumentNullException("The specified XmlDocument is a
                       null reference"));
92              ParseOwl(doc, null);
93          }
94
95          public void ParseOwl(Stream inStream, string xmlbaseUri)
96          {
97              //looks for xml:base in owl doc.
98              //if not found, then uses xmlbaseUri
99              //if xmlbaseUri is not a valid Uri it defaults to http://
                  unknown.org/
100             if (inStream == null)
101                 throw (new ArgumentNullException("The specified input stream is
                      a null reference"));
102
103             XmlDocument doc = new XmlDocument();
104             try
105             {
106                 doc.Load(inStream);
107             }
108             catch (XmlException xe)
109             {
110                 OnError(xe);
111                 return;
112             }
113             catch (Exception e)
114             {
115                 OnError(e);
116                 return;
117             }
118             ParseOwl(doc, xmlbaseUri);
119         }
120
121         public void ParseOwl(Stream inStream)
122         {
123             //looks for xml:base in Owl doc.
124             //if not found, then defaults to http://unknown.org/
125             ParseOwl(inStream, null);
```

```
126         }
127
128         public void ParseOwl(Uri uri)
129         {
130             if (uri == null)
131                 throw (new ArgumentNullException("The specified URI is a null
                      reference"));
132             //parses from a Uri.
133             XmlDocument doc = new XmlDocument();
134             try
135             {
136                 doc.Load(uri.ToString());
137             }
138             catch (XmlException xe)
139             {
140                 OnError(xe);
141                 return;
142             }
143             catch (Exception e)
144             {
145                 OnError(e);
146                 return;
147             }
148             ParseOwl(doc);
149         }
150
151         public void ParseOwl(string uri)
152         {
153             Uri srcUri = null;
154             try
155             {
156                 srcUri = new Uri(uri);
157             }
158             catch (UriFormatException)
159             {
160                 srcUri = new Uri(Path.GetFullPath(uri));
161             }
162             ParseOwl(srcUri);
163         }
164
165         #endregion
166
167         public void ParseOwl(XmlDocument doc, string xmlbaseUri)
168         {
169             //parses from the xml document
170             //if doc is null throws an ArgumentNullException
171             //looks for xml:base in doc
172             //if xml:base is not found in doc then uses the xmlbaseUri
173             //if xmlbaseUri is not a valid Uri it defaults to http://
```

```
                unknown.org/
174
175             if (doc == null)
176                 throw (new ArgumentNullException("The specified XmlDocument
                      object is a null reference"));
177
178             Warnings.Clear();
179             Errors.Clear();
180             //Start with the root
181             XmlElement root = doc.DocumentElement;
182
183             if (root.Name != "rdf:RDF")
184             {
185                 if (root.Name.ToLower() == "rdf")
186                     OnWarning("Unqualified use of rdf as the root element
                          name.");
187                 else
188                     OnWarning("Root element of an OWL document must be
                          rdf:RDF");
189             }
190             string oldXmlbaseUri = null;
191             if (_owlBase == null)
192             {
193                 _owlBase = new OwlBase();
194             }
195             else
196             {
197                 oldXmlbaseUri = (string)_owlBase.NameSpaces["xml:base"];
198                 _owlBase.NameSpaces.Remove("xml:base");
199             }
200
201             //its an OWL Document so now get the namespace info
202             int count = root.Attributes.Count;
203             for (int i = 0; i < count; i++)
204             {
205                 try
206                 {
207                     string nsName = root.Attributes[i].Name;
208                     int index = -1;
209                     index = nsName.IndexOf(":");
210                     if (index != -1)
211                         nsName = nsName.Substring(index + 1);
212
213                     // TODO: handle namespaces
214                     if (_owlBase.NameSpaces.Contains(nsName))
215                         OnWarning("Redefinition of namespace " + nsName);
216                     _owlBase.NameSpaces[nsName] = root.Attributes[i].Value;
217
218                 }
```

```
219                 catch (ArgumentException ine)
220                 {
221                     OnWarning(ine.Message);
222                 }
223             }
224 
225             _owlBase.MapNameSpaces();
226 
227             string xbUri = (string)_owlBase.NameSpaces["xml:base"];
228             if (xbUri == null)
229             {
230                 xbUri = doc.BaseURI;
231                 if (!IsValidUri(xbUri))
232                 {
233                     xbUri = xmlbaseUri;
234                     if (!IsValidUri(xbUri))
235                     {
236                         if (oldXmlbaseUri != null)
237                             xbUri = oldXmlbaseUri;
238                         else
239                         {
240                             OnWarning("Valid xml:base URI not found. Using
                    http://unknown.org/");
241                             xbUri = "http://unknown.org/";
242                         }
243                     }
244                 }
245             }
246 
247             //ignore and discard everything after the first # character
248             int pos = xbUri.IndexOf('#');
249             if (pos != -1)
250             {
251                 xbUri = xbUri.Remove(pos, xbUri.Length - pos);
252             }
253             //Now finally set the value of the xml:base Uri
254             _owlBase.NameSpaces["xml:base"] = xbUri;
255 
256             if (root.HasChildNodes)
257             {
258                 count = root.ChildNodes.Count;
259                 for (int i = 0; i < count; i++)
260                 {
261                     ProcessNode(root.ChildNodes[i]);
262                 }
263             }
264             return;
265         }
266 
```

```
267         private void ProcessNode(XmlNode node)
268         {
269             //if the node is a comment or anything else then totally ignore
270             if ((node.NodeType == XmlNodeType.Comment) || (node.NodeType ==
                  XmlNodeType.None) || (node.NodeType == XmlNodeType.Whitespace) ||
                  (node.NodeType == XmlNodeType.SignificantWhitespace))
271                 return;
272 
273             if (node.NodeType == XmlNodeType.Element)
274             {
275                 //first parse the Rdf attributes rdf:about, rdf:ID, rdf:nodeID
276                 string nodeID = ParseNodeRdfAttributes(node);
277                 ParseNodeSyntax(node, nodeID);
278                 Console.WriteLine(string.Format("Node {0}", ++nodeCounter));
279             }
280             else if ((node.NodeType == XmlNodeType.Text) || (node.NodeType ==
                  XmlNodeType.CDATA))
281             {
282                 //its a literal
283                 //so get the lang ID and the datatype from the parent node
284                 string datatypeUri = GetDatatypeUri(node.ParentNode);
285                 ///rNode = (OwlLiteral)_compileUnit.AddLiteral(node.Value,
                      langID, datatypeUri);
286             }
287 
288             if (node.HasChildNodes)
289             {
290                 int count = node.ChildNodes.Count;
291                 for (int i = 0; i < count; i++)
292                     ProcessChild(node.ChildNodes[i], node);
293             }
294             return;
295         }
296 
297         private void ParseNodeSyntax(XmlNode node, string nodeID)
298         {
299             //first get the NameSpace URI, NameSpace prefix and the LocalName
                  for the node
300             String nameSpaceURI = node.NamespaceURI;
301             String nameSpacePrefix = node.Prefix;
302             String localName = node.LocalName;
303             switch (localName)
304             {
305                 case OwlVocabulary.Ontology:
306                     _owlBase.AddOntology(ExtractNodeName(nodeID));
307                     break;
308                 case OwlVocabulary.Class:
309                 case OwlVocabulary.DatatypeProperty:
310                 case OwlVocabulary.ObjectProperty:
```

```
311                     String nodeName = ExtractNodeName(nodeID);
312                     AddOwlVocabularyMapping(nodeName, localName);
313                     break;
314                 case OwlVocabulary.Label:
315                     string className = ExtractResourceName(node.ParentNode);
316                     string labelValue = node.InnerText;
317                     _owlBase.UpdatePropertyValue(className, OwlVocabulary.Label,
                         labelValue);
318                     break;
319                 case OwlVocabulary.Restriction:
320                     //its an owl:Restriction will be processed at the child node
                         level
321                     break;
322                 case OwlVocabulary.EquivalentClass:
323                     AddEquivelantClass(node, node.ParentNode);
324                     break;
325                 //    break;
326                 default:
327                     break;
328             }
329 
330 
331             if (IsSyntacticElement(node))
332                 OnError("Cannot use " + node.Name + " as a node element name");
333         }
334 
335         private string ParseNodeRdfAttributes(XmlNode node)
336         {
337             int attrFound = 0;
338             XmlNode attr = node.Attributes["rdf:about"];
339             string retVal = null;
340             if (attr != null)
341             {
342                 //found an rdf:about attribute
343                 retVal = QualifyResource(attr.Value, GetXmlBase(node));
344                 attrFound = 1;
345             }
346             attr = node.Attributes["rdf:ID"];
347             if (attr != null)
348             {
349                 //found an rdf:ID attribute
350                 //check that its a valid xml name
351                 if (!IsXmlName(attr.Value))
352                     OnError(attr.Value + " is not an XML Name");
353                 //now check if its already been declared
354                 if (_declID[attr.Value] != null)
355                     OnError("Redefinition of rdf:ID " + attr.Value);
356                 else
357                     _declID[attr.Value] = attr.Value;
```

```
358                 retVal = PrependXmlBase(attr.Value, GetXmlBase(node));
359                 attrFound += 2;
360             }
361             attr = node.Attributes["rdf:nodeID"];
362             if (attr != null)
363             {
364                 //found an rdf:nodeID attribute
365                 //check if its an xml name
366                 if (!IsXmlName(attr.Value))
367                     OnError(attr.Value + " is not an XML Name");
368                 //     retVal = GetBlankNodeUri(attr.Value);
369                 attrFound += 4;
370             }
371             attr = node.Attributes["xml:lang"];
372             if (attr != null)
373             {
374                 //found an xml:lanf attribute
375                 //check if its an xml name
376                 if (!IsXmlName(attr.Value))
377                     OnError(attr.Value + " is not an XML Name");
378                 retVal = attr.Value;
379 
380             }
381             switch (attrFound)
382             {
383                 case 3:
384                     OnError("Cannot use rdf:about and rdf:ID together");
385                     break;
386                 case 5:
387                     OnError("Cannot use rdf:about and rdf:nodeID together");
388                     break;
389                 case 6:
390                     OnError("Cannot use rdf:ID and rdf:nodeID together");
391                     break;
392                 case 7:
393                     OnError("Cannot use rdf:about, rdf:ID and rdf:nodeID
                         together");
394                     break;
395             }
396             return retVal;
397         }
398 
399         private void ProcessChild(XmlNode node, XmlNode outerNode)
400         {
401             //ignore empty or comment nodes
402             if ((node.NodeType == XmlNodeType.Comment) || (node.NodeType ==
                XmlNodeType.None) || (node.NodeType == XmlNodeType.Whitespace) ||
                (node.NodeType == XmlNodeType.SignificantWhitespace))
403                 return;
```

```
404             if (node.NodeType == XmlNodeType.Element)
405             {
406                 //get the xml:base attribute...
407                 XmlAttribute xmlBaseAttr = node.Attributes["xml:base"];
408                 if ((xmlBaseAttr == null) && (outerNode != null))
409                 {
410                     //ok the child does not have an xml:base... and the parent
                        is not null i.e. this node is not a child of the rdf:RDF
                        root
411                     xmlBaseAttr = node.ParentNode.Attributes["xml:base"];
412                     if (xmlBaseAttr != null)
413                         node.Attributes.Append(xmlBaseAttr);
414                     string rdfResourceName = null;
415                     string className = null;
416 
417                     switch (node.LocalName)
418                     {
419 
420                         case OwlVocabulary.subClassOf:
421 
422                             string parentType = OwlVocabulary.Class;
423                             if (node.HasChildNodes && node.FirstChild.LocalName
                        == OwlVocabulary.Restriction)
424                             {
425                                 XmlNode restrictionNode = node.FirstChild;
426                                 if (restrictionNode.FirstChild.LocalName ==
                        OwlVocabulary.OnProperty)
427                                     parentType = OwlVocabulary.ObjectProperty;
428                             }
429 
430                             UpdateHeirarchy(node, outerNode, parentType);
431                             break;
432                         case OwlVocabulary.subPropertyOf:
433                             parentType = OwlVocabulary.ObjectProperty;
434                             UpdateHeirarchy(node, outerNode, parentType);
435                             break;
436                         case "type":
437                             rdfResourceName = ExtractRdfResourceName(node);
438                             className = ExtractResourceName(outerNode);
439                             switch (rdfResourceName)
440                             {
441                                 case OwlVocabulary.TransitiveProperty:
442                                 case OwlVocabulary.SymmetricProperty:
443                                 case OwlVocabulary.FunctionalProperty:
444                                 case OwlVocabulary.InverseFunctionalProperty:
445                                     _owlBase.UpdatePropertyValue(className,
                        rdfResourceName, true);
446                                     break;
447                                 default:
```

```
448                                 break;
449
450                             }
451                             break;
452                         case OwlVocabulary.Domain:
453                         case OwlVocabulary.Range:
454                             rdfResourceName = ExtractRdfResourceName(node);
455                             className = ExtractResourceName(outerNode);
456                             _owlBase.UpdatePropertyValue(className,
                        node.LocalName, rdfResourceName);
457                             break;
458                         case OwlVocabulary.Cardinality:
459                         case OwlVocabulary.MinCardinality:
460                         case OwlVocabulary.MaxCardinality:
461                         case OwlVocabulary.SomeValuesFrom:
462                         case OwlVocabulary.AllValuesFrom:
463                         case OwlVocabulary.QualifiedCardinality:
464                         case OwlVocabulary.HasValue:
465                             if (outerNode.ParentNode.LocalName ==
                        OwlVocabulary.subClassOf)
466                                 AddPropertyRestriction(node, outerNode);
467                             break;
468                         case OwlVocabulary.InverseOf:
469                             ProcessInverseOfNode(node, outerNode);
470                             break;
471                         case OwlVocabulary.Label:
472                             if (!outerNode.Name.Contains("DbXref") && !
                        outerNode.Name.Contains("Ontology")) //TODO HACK Flag
473                                 AddLabel(node, outerNode);
474                             break;
475                         case OwlVocabulary.OnProperty:
476                         case OwlVocabulary.UnionOf:
477                         case OwlVocabulary.OneOf:
478                         case OwlVocabulary.IntersectionOf:
479                             break; // processed with the Equivelant class or
                        restriction nodes...
480                         case OwlVocabulary.EquivalentClass:
481                             AddEquivelantClass(node, outerNode);
482                             break;
483                         case OwlVocabulary.DisjointWith:
484                             AddDisjointClass(node, outerNode);
485                             break;
486                         case OwlVocabulary.ComplementOf:
487                             AddComplementOfClass(node, outerNode);
488                             break;
489                         case OwlVocabulary.OnClass:
490                             break;
491                         case OwlVocabulary.Class:
492                             className = ExtractResourceName(node);
```

```
493                             _owlBase.AddType(className);
494                             break;
495                         default:
496                             break;
497                     }
498                 }
499             }
500 
501             if (ParseEdgeRdfAttributes(node)) //if the process attributes method returns true then process the children of this node
502             {
503                 if (node.HasChildNodes)
504                 {
505                     int count = node.ChildNodes.Count;
506                     for (int i = 0; i < count; i++)
507                         ProcessNode(node.ChildNodes[i]);
508                 }
509             }
510 
511             //coming back up the tree
512             ParseEdgeAttributes(node);
513 
514         }
515 
516         private void ParseEdgeAttributes(XmlNode node)
517         {
518             if (node.Attributes == null)
519                 return;
520             //go through all the attributes
521             int count = node.Attributes.Count;
522             for (int i = 0; i < count; i++)
523             {
524                 XmlAttribute attr = node.Attributes[i];
525 
526                 if ((!IsSyntacticElement(attr)) || (attr.Prefix == "rdf" && attr.LocalName == "value"))
527                 {
528                     if ((attr.NamespaceURI == null) || (attr.NamespaceURI.Length == 0))
529                     {
530                         OnError("Unqualified attribute: " + attr.LocalName);
531                         continue;
532                     }
533                     if ((attr.Name == "rdf:value"))
534                     {
535                         OnError("Cannot use rdf:value (" + attr.Value + ") as property for a literal (");
536                         continue;
537                     }
```

```
538
539
540                     string edgeID = attr.NamespaceURI + attr.LocalName;
541                     string literalValue = attr.Value;
542                     string datatypeUri = GetDatatypeUri(node);
543                 }
544             }
545         }
546
547         private bool ParseEdgeRdfAttributes(XmlNode node)
548         {
549             int attrFound = 0;
550             bool parseChildren = true;
551             if (node.Attributes == null)
552                 return parseChildren;
553             XmlNode attr = node.Attributes["rdf:resource"];
554             if (attr != null)
555             {
556                 ProcessRdfResource(node, attr.Value);
557                 attrFound = 1;
558                 parseChildren = false;
559             }
560             attr = node.Attributes["rdf:nodeID"];
561             if (attr != null)
562             {
563                 ProcessRdfNodeID(attr.Value);
564                 attrFound += 2;
565                 parseChildren = false;
566             }
567             attr = node.Attributes["rdf:parseType"];
568             if (attr != null)
569             {
570                 //ProcessRdfParseType(node, attr.Value);
571                 attrFound += 4;
572                 parseChildren = false;
573             }
574
575             switch (attrFound)
576             {
577                 case 3:
578                     OnError("Cannot use rdf:resource and rdf:nodeID together");
579                     break;
580                 case 5:
581                     OnError("Cannot use rdf:resource and rdf:parseType
                         together");
582                     break;
583                 case 6:
584                     OnError("Cannot use rdf:nodeID and rdf:parseType together");
585                     break;
```

```
586                 case 7:
587                     OnError("Cannot use rdf:resource, rdf:nodeID and
                        rdf:parseType together");
588                     break;
589             }
590             return parseChildren;
591         }
592 
593         private void ProcessRdfNodeID(string value)
594         {
595             throw new NotImplementedException();
596         }
597 
598         #endregion
599 
600         #region OwlMapping
601 
602 
603         private void AddOwlVocabularyMapping(string name, string owlVocabulary)
604         {
605             if (name == null)
606                 return;
607 
608             switch (owlVocabulary)
609             {
610                 case OwlVocabulary.ObjectProperty:
611                     _owlBase.AddObjectProperty(name, null, true);
612                     break;
613                 case OwlVocabulary.DatatypeProperty:
614                     _owlBase.AddDataTypeProperty(name);
615                     break;
616                 case OwlVocabulary.Class:
617                     _owlBase.AddType(name);
618                     break;
619                 case OwlVocabulary.Ontology:
620                     _owlBase.AddOntology(name);
621                     break;
622                 default:
623                     break;
624 
625             }
626 
627 
628         }
629 
630         private void AddPropertyRestriction(XmlNode node, XmlNode outerNode,
              string className = null)
631         {
632             string restriction = node.LocalName;
```

```
633             if (className == null)
634                 className = ExtractResourceName
                      (outerNode.ParentNode.ParentNode);
635             string propertyName = ExtractPropertyName(outerNode);
636             object propertyFiller = ExtractRdfResourceName(node);
637             string owlTerm = null;
638             if (propertyFiller == null)
639             {
640                 switch (restriction)
641                 {
642                     case OwlVocabulary.OnClass:
643                         break;
644                     case OwlVocabulary.Cardinality:
645                     case OwlVocabulary.MinCardinality:
646                     case OwlVocabulary.MaxCardinality:
647                     case OwlVocabulary.QualifiedCardinality:
648                         propertyFiller = node.InnerText;
649                         break;
650                     case OwlVocabulary.SomeValuesFrom:
651                     case OwlVocabulary.AllValuesFrom:
652                     case OwlVocabulary.HasValue:
653                         XmlNode collectionNode = node.ChildNodes[0];
654                         if (collectionNode.HasChildNodes)
655                         {
656                             collectionNode = collectionNode.ChildNodes[0];
657                             owlTerm = collectionNode.LocalName;
658                             propertyFiller = new List<string>();
659                             foreach (XmlNode child in collectionNode.ChildNodes)
660                             {
661                                 ((List<string>)propertyFiller).Add
                        (ExtractResourceName(child));
662                             }
663                         }
664                         else
665                         {
666                             owlTerm = collectionNode.LocalName;
667                             propertyFiller = new List<string>();
668                             foreach (XmlNode child in collectionNode.ChildNodes)
669                             {
670                                 ((List<string>)propertyFiller).Add
                        (ExtractResourceName(child));
671                             }
672                         }
673
674                         break;
675                     default:
676                         break;
677
678                 }
```

```
679
680             }
681             _owlBase.AddRestriction(restriction, className, propertyName,
                  propertyFiller, owlTerm);
682         }
683
684         private void AddEquivelantClass(XmlNode node, XmlNode outerNode)
685         {
686             List<KeyValuePair<string, string>> visitedChildern = new
                  List<KeyValuePair<string, string>>();
687             List<string> auxuliaryList = new List<string>();
688
689             string className = ExtractResourceName(outerNode);
690             string equivelantClass = null;
691             FindEquivelantClass(node, ref equivelantClass, ref visitedChildern,
                  ref auxuliaryList);
692
693             _owlBase.AddEquivelantClass(className, equivelantClass);
694         }
695
696         private void FindEquivelantClass(XmlNode node, ref string name, ref
              List<KeyValuePair<string, string>> visitedChildern, ref List<string>
              auxuliaryList)
697         {
698
699             if (!node.HasChildNodes)
700             {
701                 return;
702             }
703             foreach (XmlNode child in node.ChildNodes)
704             {
705                 FindEquivelantClass(child, ref name, ref visitedChildern, ref
                      auxuliaryList);
706                 VisitNode(child, ref name, ref visitedChildern, ref
                      auxuliaryList);
707             }
708
709         }
710
711         private void VisitNode(XmlNode node, ref string name, ref
              List<KeyValuePair<string, string>> visitedChildern, ref List<string>
              auxuliaryList)
712         {
713
714             switch (node.LocalName)
715             {
716                 case OwlVocabulary.Description:
717                     visitedChildern.Add(new KeyValuePair<string, string>
                          (node.ParentNode.LocalName, ExtractResourceName(node)));
```

```
718                     break;
719                 case OwlVocabulary.OneOf:
720                     auxuliaryList = (from c in visitedChildern where c.Key ==
                        OwlVocabulary.OneOf select c.Value).ToList();
721                     name = _owlBase.AddBlankNodeType(OwlVocabulary.OneOf,
                        auxuliaryList);
722                     visitedChildern = visitedChildern.Where(c => c.Key !=
                        OwlVocabulary.OneOf).ToList();
723                     visitedChildern.Add(new KeyValuePair<string, string>
                        (node.ParentNode.LocalName, name));
724                     break;
725                 case OwlVocabulary.UnionOf:
726                     auxuliaryList = (from c in visitedChildern where c.Key ==
                        OwlVocabulary.UnionOf select c.Value).ToList();
727                     name = _owlBase.AddBlankNodeType(OwlVocabulary.UnionOf,
                        auxuliaryList);
728                     visitedChildern = visitedChildern.Where(c => c.Key !=
                        OwlVocabulary.UnionOf).ToList();
729                     visitedChildern.Add(new KeyValuePair<string, string>
                        (node.ParentNode.LocalName, name));
730                     break;
731                 case OwlVocabulary.IntersectionOf:
732                     var buffer = auxuliaryList;
733                     auxuliaryList = auxuliaryList.Where(item => !item.StartsWith
                        ("Blank")).ToList();
734                     auxuliaryList.AddRange( (from c in visitedChildern where
                        c.Key == OwlVocabulary.IntersectionOf select c.Value).ToList
                        ());
735                     name = _owlBase.AddBlankNodeType
                        (OwlVocabulary.IntersectionOf, auxuliaryList);
736                     visitedChildern = visitedChildern.Where(c => c.Key !=
                        OwlVocabulary.IntersectionOf).ToList();
737                     visitedChildern.Add(new KeyValuePair<string, string>
                        (node.ParentNode.LocalName, name));
738                     break;
739                 case OwlVocabulary.Restriction:
740                     name = _owlBase.AddBlankNodeType(null, auxuliaryList);
741                     AddPropertyRestriction(node.ChildNodes[1], node, name);
742                     visitedChildern.Add(new KeyValuePair<string, string>
                        (node.ParentNode.LocalName, name));
743                     break;
744                 case OwlVocabulary.OnProperty: //processed with the restriction
                      cases
745                     break;
746                 case OwlVocabulary.Class:
747                     name = (from c in visitedChildern where c.Key ==
                        OwlVocabulary.Class select c.Value).ToList().FirstOrDefault
                        ();
748                     if (string.IsNullOrEmpty(name))
```

```
749                         name = ExtractResourceName(node);
750                     visitedChildern = visitedChildern.Where(c => c.Key !=
                        OwlVocabulary.Class).ToList();
751                     visitedChildern.Add(new KeyValuePair<string, string>
                        (node.ParentNode.LocalName, name));
752                     break;
753                 case OwlVocabulary.EquivalentClass:
754                     name = (from c in visitedChildern where c.Key ==
                        OwlVocabulary.EquivalentClass select c.Value).ToList
                        ().FirstOrDefault();
755                     break;
756                 case OwlVocabulary.ComplementOf:
757                     string propertyObject = ExtractRdfResourceName(node);
758                     if (propertyObject == null)
759                         auxuliaryList = (from c in visitedChildern where c.Key
                        == OwlVocabulary.ComplementOf select c.Value).ToList();
760                     else
761                         auxuliaryList = new List<string>() { propertyObject };
762                     name = _owlBase.AddBlankNodeType(OwlVocabulary.ComplementOf,
                         auxuliaryList);
763                     visitedChildern = visitedChildern.Where(c => c.Key !=
                        OwlVocabulary.ComplementOf).ToList();
764                     visitedChildern.Add(new KeyValuePair<string, string>
                        (node.ParentNode.LocalName, name));
765                     break;
766                 default:
767                     name = node.LocalName;
768                     break;
769
770
771             }
772         }
773
774         private void AddDisjointClass(XmlNode node, XmlNode outerNode)
775         {
776
777             string className = ExtractResourceName(outerNode);
778             string disjointClassName = ExtractRdfResourceName(node);
779             _owlBase.AddDisjointClass(className, disjointClassName);
780         }
781
782         private void AddComplementOfClass(XmlNode node, XmlNode outerNode)
783         {
784
785             string className = ExtractResourceName(outerNode);
786             string complementClassName = ExtractRdfResourceName(node);
787             _owlBase.AddComplementOfClass(className, complementClassName);
788         }
789
```

```
790         private void AddLabel(XmlNode node, XmlNode outerNode)
791         {
792
793             string className = ExtractResourceName(outerNode);
794             string label = node.FirstChild.Value;
795             if (className != null && label != null)
796                 _owlBase.SetLabelProperty(className, label);
797         }
798
799         private void ProcessInverseOfNode(XmlNode node, XmlNode outerNode)
800         {
801
802             string className = ExtractResourceName(outerNode);
803             object propertyFiller = ExtractRdfResourceName(node);
804             if (propertyFiller == null)
805             {
806                 XmlNode collectionNode = node.ChildNodes.Cast<XmlNode>().Where(n
                        => n.LocalName == "Class").FirstOrDefault().ChildNodes[0];
807                 propertyFiller = new List<string>();
808                 foreach (XmlNode child in collectionNode.ChildNodes)
809                 {
810                     ((List<string>)propertyFiller).Add(ExtractResourceName
                        (child));
811                 }
812
813
814             }
815             _owlBase.AddInverseOf(className, propertyFiller);
816         }
817
818         #endregion
819
820         #region Manipulators
821
822         private void UpdateHeirarchy(XmlNode node, XmlNode outerNode, string
              type)
823         {
824             string parentClass = null;
825             parentClass = ExtractRdfResourceName(node);
826             if (parentClass != null)
827             {
828                 if (!_owlBase.TypeExists(parentClass))
829                     if (type == "Class")
830                         _owlBase.AddType(parentClass);
831                     else if (type == "ObjectProperty")
832                         _owlBase.AddObjectProperty(parentClass, null, true);
833                 string className = ExtractResourceName(outerNode);
834                 _owlBase.UpdateClassHeirarchy(className, parentClass);
835             }
```

```
836         }
837
838         private string ExtractResourceName(XmlNode node)
839         {
840             string name = null;
841             if (node.Attributes["rdf:about"] != null)
842             {
843                 int index = node.Attributes["rdf:about"].Value.IndexOf("#");
844                 if (index == -1)
845                     index = node.Attributes["rdf:about"].Value.LastIndexOf("/");
846                 name = node.Attributes["rdf:about"].Value.Substring(index + 1);
847             }
848             else if (node.Attributes["rdf:ID"] != null)
849                 name = node.Attributes["rdf:ID"].Value;
850
851             if (string.IsNullOrWhiteSpace(name))
852             {
853
854                 switch (node.LocalName)
855                 {
856                     case OwlVocabulary.Restriction:
857                         string restrictionBlankNode = _owlBase.AddBlankNodeType
                        (null, null);
858                         AddPropertyRestriction(node.ChildNodes[1], node,
                        restrictionBlankNode);
859                         return restrictionBlankNode;
860                     default:
861                         name = node.LocalName;
862                         break;
863                 }
864             }
865             return name;
866
867         }
868
869         private string ExtractPropertyName(XmlNode node)
870         {
871             XmlNode targetChild = (node.ChildNodes.OfType<XmlNode>().Where(n =>
                n.LocalName == "onProperty")).FirstOrDefault();
872             string name;
873
874             try
875             {
876
877                 name = ExtractRdfResourceName(targetChild);
878             }
879             catch
880             {
881                 targetChild = (targetChild.ChildNodes.OfType<XmlNode>().Where(n
```

```
                    => n.Name == "owl:ObjectProperty" || n.Name ==
                    "owl:FunctionalProperty")).FirstOrDefault();
882                 name = ExtractResourceName(targetChild);
883             }
884
885             return name;
886         }
887
888         private string ExtractRdfResourceName(XmlNode node)
889         {
890             string name = null;
891             int splitIndex = -1;
892             string nameSpace = null;
893             try
894             {
895                 if (node.Attributes["rdf:resource"].Value.ToLower().StartsWith
                    ("http"))
896                     splitIndex = node.Attributes
                        ["rdf:resource"].Value.LastIndexOf('/');
897                 name = node.Attributes["rdf:resource"].Value.Substring
                    (splitIndex + 1);
898                 // check the index of #
899                 splitIndex = name.IndexOf("#");
900                 //TODO: Handle this when considering namespaces....
901                 if (splitIndex == 0)
902                     name = name.Substring(splitIndex + 1);
903                 if (splitIndex > 1) //
904                 {
905                     nameSpace = name.Substring(0, splitIndex);
906                     name = name.Substring(splitIndex + 1);
907                 }
908
909
910
911             }
912             catch
913             {
914                 if (node.InnerXml.Length > lengthThreshold)
915                 {
916                     try
917                     {
918                         string attribute = node.InnerXml.Split(' ').ToList
                        ().Where(att => att.StartsWith("rdf:ID") | att.StartsWith
                        ("rdf:about")).FirstOrDefault().ToString();
919                         name = attribute.Substring(attribute.IndexOf("=") +
                        2).TrimStart('#').TrimEnd('\"');
920                         splitIndex = -1;
921                         if (name.ToLower().StartsWith("http"))
922                             splitIndex = name.LastIndexOf('/');
```

```
923                         name = name.Substring(splitIndex + 1);
924                     }
925                     catch { }
926 
927                 }
928             }
929             return name;
930         }
931 
932         private string ExtractNodeName(string nodeID)
933         {
934             string nodeName = null;
935             if (nodeID != null)
936             {
937                 try
938                 {
939                     nodeName = nodeID.Substring(nodeID.IndexOf("#"));
940 
941                 }
942                 catch
943                 {
944                     nodeName = nodeID.Substring(nodeID.LastIndexOf("/"));
945                 }
946                 nodeName = nodeName.Substring(1);
947             }
948             return nodeName;
949 
950         }
951 
952         private string PrependXmlBase(string id, string xmlBaseUri)
953         {
954             if (IsValidUri(id))
955                 return id;
956             if (id == null)
957                 id = "";
958             if ((xmlBaseUri == null) || (xmlBaseUri.Length == 0))
959                 xmlBaseUri = _owlBase.NameSpaces["xml:base"].ToString();
960             if (id[0] == '#')
961                 id = xmlBaseUri + id;
962             else
963                 id = xmlBaseUri + "#" + id;
964             return id;
965         }
966 
967         private string GetXmlBase(XmlNode node)
968         {
969             XmlNode attr = node.Attributes["xml:base"];
970             if (attr == null)
971                 return null;
```

```
 972             string xmlBaseUri = attr.Value;
 973             if (!IsValidUri(xmlBaseUri))
 974                 return null;
 975
 976             //ignore and discard everything after the first # character
 977             int pos = xmlBaseUri.IndexOf('#');
 978             if (pos != -1)
 979             {
 980                 xmlBaseUri = xmlBaseUri.Remove(pos, xmlBaseUri.Length - pos);
 981             }
 982             return xmlBaseUri;
 983         }
 984
 985         private bool IsUnqualifiedRdfProperty(XmlAttribute xmlAttribute)
 986         {
 987             string prefix = xmlAttribute.Prefix;
 988             string localName = xmlAttribute.LocalName;
 989
 990             if (prefix.Length == 0)
 991             {
 992                 if (_rdfXmlProperties["rdf:" + localName] != null)
 993                     return true;
 994             }
 995             return false;
 996         }
 997
 998         private bool IsOwlRdfXmlProperty(XmlAttribute prop)
 999         {
1000             if (_rdfXmlProperties[prop.Name] != null)
1001                 return true;
1002             if ((prop.Prefix.ToLower().StartsWith("xml")) ||
                   (prop.LocalName.ToLower().StartsWith("xml")))
1003                 return true;
1004             if (((prop.Prefix == "rdf") || (prop.Prefix == "xml") ||
                   (prop.Prefix == "owl")) && (!IsNonSyntacticElement(prop) && (!
                   IsSyntacticElement(prop))))
1005                 OnWarning("Unknown OWL, RDF or XML property: " + prop.Prefix +
                       ":" + prop.LocalName);
1006             return false;
1007         }
1008
1009         private string QualifyResource(string val, string xmlBaseUriString)
1010         {
1011             if ((xmlBaseUriString == null) || (xmlBaseUriString.Length == 0))
1012                 xmlBaseUriString = _owlBase.NameSpaces["xml:base"].ToString();
1013
1014             //if val is blank or null, val = xml:base
1015             if ((val == null) || (val.Length == 0))
1016                 return xmlBaseUriString;
```

```
1017
1018            Uri xmlBaseUri = new Uri(xmlBaseUriString);
1019
1020            // if val starts with // or val starts with \\
1021            // val = scheme + ":"+ val;
1022            if (val.StartsWith("\\\\") || val.StartsWith("//"))
1023                return xmlBaseUri.Scheme + ":" + val;
1024            if (IsValidUri(val))
1025                return val;
1026
1027            // if val starts with #
1028            // val = xml:base+val
1029            if (val.StartsWith("#"))
1030                return xmlBaseUriString + val;
1031
1032            // if val starts with /
1033            // val = scheme + "://" + authority + val;
1034            if (val.StartsWith("/") || val.StartsWith("\\"))
1035                return xmlBaseUri.Scheme + "://" + xmlBaseUri.Authority + val;
1036
1037            string folderPath = GetFolderPath(xmlBaseUri.AbsolutePath);
1038            // if val starts with ../
1039            // val = scheme+"://" + authority + modpath {modpath =
                  GetAbsolutePath(folderpath,val)}
1040            if (val.StartsWith("../") || val.StartsWith("..\\"))
1041            {
1042                string absPath = GetAbsolutePath(folderPath, val);
1043                return xmlBaseUri.Scheme + "://" + xmlBaseUri.Authority +
                      absPath;
1044            }
1045            // if val doesnt start with anything
1046            // val = scheme+"://" + authority + folderPath + val
1047            return xmlBaseUri.Scheme + "://" + xmlBaseUri.Authority + folderPath
                  + val;
1048        }
1049
1050        private string GetFolderPath(string pathStr)
1051        {
1052
1053            if (!(pathStr.EndsWith("/") || pathStr.EndsWith("\\")))
1054            {
1055                int index = pathStr.LastIndexOfAny(new Char[] { '\\', '/' });
1056                pathStr = pathStr.Substring(0, index + 1);
1057            }
1058            return pathStr;
1059        }
1060
1061        private string GetAbsolutePath(string initPath, string relPath)
1062        {
```

```
1063            Regex regex = new Regex(@"[/\\](\w)*[/\\](\.\.)[/\\]",
                  RegexOptions.IgnoreCase | RegexOptions.Multiline |
                  RegexOptions.Compiled);
1064            string path = initPath + relPath;
1065            string oldPath = "";
1066            while (oldPath != path)
1067            {
1068                oldPath = path;
1069                path = regex.Replace(oldPath, "/");
1070            }
1071            return path;
1072        }
1073
1074        private bool IsXmlName(string value)
1075        {
1076            try
1077            {
1078                XmlConvert.VerifyNCName(value);
1079                return true;
1080            }
1081            catch (XmlException)
1082            {
1083                return false;
1084            }
1085        }
1086
1087        private bool IsSyntacticElement(XmlNode node)
1088        {
1089            if (_syntacticElements[node.Name] != null)
1090                return true;
1091            return false;
1092        }
1093
1094        private bool IsNonSyntacticElement(XmlNode prop)
1095        {
1096            if (_nonSyntacticElements[prop.Name] != null)
1097                return true;
1098            return false;
1099        }
1100
1101        private string GetDatatypeUri(XmlNode node)
1102        {
1103            XmlNode datatypeAttr = node.Attributes["rdf:datatype"];
1104            if (datatypeAttr != null)
1105                return PrependXmlBase(datatypeAttr.Value, GetXmlBase(node));
1106            return null;
1107        }
1108
1109        private void ProcessRdfResource(XmlNode node, string resourceUri)
```

```
1110          {
1111              string nodeID = QualifyResource(resourceUri, GetXmlBase(node));
1112          }
1113
1114
1115
1116          #endregion
1117      }
1118  }
1119
```