OntoJIT: Parsing Native OWL DL into Executable Ontologies in an Object Oriented Paradigm

Sohaila Baset and Kilian Stoffel

Information Management Institute
University of Neuchatel,
Neuchatel, Switzerland
sohaila.baset@unine.ch,
Kilian.Stoffel@unine.ch

Abstract. Despite meriting the growing consensus between researchers and practitioners of ontology modeling, the Web Ontology Language OWL still has a modest presence in the communities of "traditional" web developers and software engineers. This resulted in hoarding the semantic web field in a rather small circle of people with a certain profile of expertise. In this paper we present OntoJIT, our novel approach toward a democratized semantic web where we bring OWL ontologies into the comfort-zone of end-application developers. We focus particularly on parsing OWL source files into executable ontologies in an object oriented programming paradigm. We finally demonstrate the dynamic code-base created as the result of parsing some reference OWL DL ontologies.

Keywords: Ontologies, OWL, Semantic Web, Meta Programming,Dynamic Compilation

1 Background and Motivation

With a stack full of recognized standards and specifications, the Web Ontology Language OWL has made long strides to allocate itself a distinctive spot in the landscape of knowledge representation and semantic web. Obviously, OWL is not the only player in the scene; over the couple of last decades many other languages have also emerged in the ontology modeling paradigm. Most of these languages are logic-based formalisms with underlying constructs in first order logic [5][7][8][11] or in one of the description logic fragments like OWL itself [3][4] and its predecessor DAML+OIL [10]. Some frame-based languages have also seen some success in that area [12][13][14], in particular KL-One has integrated the automated deductive reasoning of logic-based languages into hierarchical semantic networks[9].

If we look at OWL characteristics; beside its strong expressive capabilities and logic based formalism, OWL has also got many flavors that are tailored to fulfill the different needs of ontology systems stakeholders[4]. These characteristics allowed OWL to stand out among its counterparts and OWL ontologies

became dominant in a wide range of application domains. From the perspective of traditional software developers, however, these very same characteristics have contributed to a certain extent in augmenting the complexity surrounding OWL ontologies and logic-based formalisms in general.

We raise the issue of democratized semantics where a wider range of developers are invited to actively participate in the making process of semantic applications. Addressing this issue involves certainly more aspects than what we can cover in a single paper. In this paper, we rather start by whetting developers' appetite for ontologies by expressing them in a programming language –or paradigm– that the developers are already comfortable with. For that purpose, we sketch our tool OntoJIT that parses existing OWL DL ontologies into executable fragments of code in C# while maintaining their semantics. We demonstrate the parsing results obtained and the limitations of the current state. We finally discuss some of the related projects and directions for future work.

2 Preliminaries

2.1 Executable Ontologies

Before being able to work on an ontology, inference engines require the ontology to be loaded into memory. This task is achieved by an ontology loader that transforms the ontology from its syntactic form e.g. RDF/XML into an in-memory representation. In the literature, there are two prominent in-memory representations for OWL ontologies: The first one is the abstract syntax tree AST model which is used in OWL API, previously known as the WonderWeb OWL API, [19][18]. The other representation model is the RDF triple and is the format that is adopted in Jena [15].

In our work, we look into the classification of in-memory ontology representations from a different perspective. More precisely we differentiate between two forms of in-memory representation: The passive form and the active form. To illustrate what we designate by each form, we consider the parsing output produced by Jena and OWL API; after the parsing step is completed, both parsing output models, i.e. AST or RDF graph, will eventually reside in the data segment of the program allocated memory waiting to be operated on by the inference engine and in that sense both are examples of the passive forms. In the active form, on the other hand, the output of the parsing step belongs to the code segment of the allocated memory. That is, the syntactic RDF/XML representation is transformed and loaded in memory as a set of executables.

Now projecting the object oriented programming paradigm into this view of active in-memory ontology representation yields the term executable ontology that we first present in this paper. We can now think of OWL concepts and individuals as OOP classes and instances spread over code namespaces that can be compiled and run.

2.2 Meta Programming in Strongly Typed Languages

Parsing RDF/XML into executable ontologies clearly adds another layer of complexity into the already non-trivial parsing task. It requires dynamically generating code statements that are equivalent to the RDF triple being parsed. In such settings, the deployment of meta programming techniques proves advantageous. Meta programming refers to the programming paradigms and the means by which a program has knowledge of itself or can manipulate itself. To that end, meta programs are programs that write programs. Examples of meta programs are optimizers, partial evaluation systems and program transformers [20]. There exists many classification of meta programs, among them is the static vs run-time classification i.e. whether the produced output program is written to disk or dynamically compiled at run-time, the manually vs automatically annotated classification i.e. whether the staging annotations are placed directly by the programmer or produced by an automatic process and finally the homogeneous vs heterogeneous programs which concerns whether or not the meta language is the same as the program output language [20]. Our proposed OntoJIT RDF/XML parser is a manually annotated, run-time heterogeneous meta program.

Like many paradigms in software development, Meta programming is an approach that is not equally supported by all programming languages. Some languages, such as CaML[24], are designed with meta programming in the core of their philosophy. Dynamic languages like Prolog and smalltalk have fundamental meta programming features[21]. Macros in Lisp and Scala also provide strong support for meta programming [22][23], whereas Python programmers usually use meta classes. When it comes to strongly typed languages, however, the emphasis on meta programming features becomes less evident. This does not mean that meta programming is not supported in many of these languages; C++ offers templates for meta programming [31], Java programs have annotations [30] and .Net languages use annotations and/or reflection to produce meta programs[32]. Indeed, the parser presented here was realized using one of the meta programming libraries offered in .Net[33].

3 OntoJIT Parser

Parsing OWL source files into executable source code is the first step of an ongoing effort to bring ontologies into the table of application developers. The overall goal of this effort is not limited to parsing ontologies into compiled source code, the real interesting part is the potential reasoning possibilities over this newly created eco-system of executable ontologies; hence the name OntoJIT refers to just in time ontologies and is inspired from the dynamic "Just in Time" compilation in .Net languages. The OntoJIT parser we present here is written C#. It produces compiled source code in the form of dynamic linking libraries or executables and it can also produce C# source files as an intermediate output. In the following sections we discuss some of the key points in the design and implementation of OntoJIT parser.

3.1 Parsing OWL files

OWL Graph Traversal Most existing OWL parsing tools use a recursive depth first search to perform a one-pass traversal of OWL source. This seems like an elegant approach for a streaming-like parsing; the DFS serves as a serialization technique and for each construct visited in the source, a corresponding node or edge is attached to the OWL graph being constructed in memory. However, when parsing output is an executable, the pure DFS approach is unfortunately insufficient. Deciding on the corresponding code statements to a syntactic construct requires all information related to this construct to be available at node processing time; which is clearly not the case with the inter-node associativity present in OWL source documents. Here we have two approaches to overcome this limitation, first approach is to use multiple-pass traversal to guarantee that we have complete information before generating the corresponding output. This approach is clearly less efficient compared to the one-pass traversal both in execution time as well as in space complexity since it requires maintaining the intermediate state of nodes being parsed over many passes. The other approach, which is the one used in this paper, is to combine pre-order DFS traversal with look up operations when necessary. The parser presented here is built to read RDF/XML syntax; in that case, the look-up operations are simply forward jumps within to the RDF/XML child nodes and the set of possible look-ups is limited assuming prior knowledge of the associations patterns of OWL nodes.

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 into 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 achieved.

3.2 OWL to OOP Mapping

When comparing the expressiveness aspect of OWL to that of formal programming languages, programming languages rank way below than even the most restricted profile of OWL. The semantic richness of OWL DL ontologies makes it difficult to find an OOP counterpart for each OWL DL semantic construct. Furthermore, there are some fundamental differences between the two schools of modeling such as the notion of disjoint classes, inheritance model and many others. When mapping OWL DL to OOP, our goal was to exploit the native programming language constructs while at the same time trying not to violate the OOP design principles. Although the mapping seems self evident in some parts e.g. `owl:class` as an OOP class, `rdfs:subClassOf` as OOP class inheritance

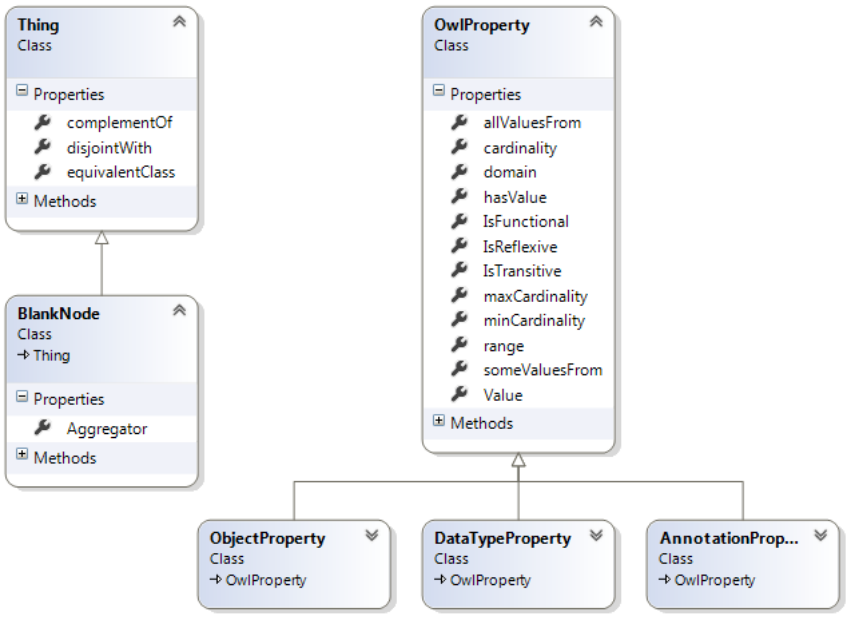

Fig. 1. The initial output scheme in OntoJIT.

relation and OWL individuals as 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`. One could still create native constructs that are semantically equivalent to such terms by enforcing some design patterns and constraints but this approach has some consequences that we will discuss in one of the following sections. One other possibility is to rely on annotations to express all OWL terms that are foreign in the OOP language, but in plain OOP terms this means that most of the modeled information about an object is laid outside of it and is not directly accessible via its properties. Instead, in OntoJIT parser, for the major part of OWL terms, meta properties are created that form the bases for mapping OWL concepts, properties and restrictions. The meta properties are defined in the top hierarchy level and are then inherited by all parsed classes afterwards and masked where necessary. One important thing to clarify is that the term "meta" used here refers to a completely different sense than the programming technique discussed earlier, the usage of the term here is rather functional; the idea is that these meta properties would cover up for the missing explicit semantics in the formal language constructs, and the full interpretation of the meta properties semantics is to be realized by an inference component on top of the parsing layer. Figure 1. shows the initial output scheme in OntoJIT where meta properties are first defined.

Blank Nodes Just like in RDF/XML, OntoJIT uses blank nodes to express a property restriction or class description axioms. Though in our implementation, blank nodes are not anonymous; they are created as class definitions with

Table 1. OWL DL axioms and their OntoJIT 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	
Indivisual	indivisual	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	

automatically (and deterministically) generated names to make them available for subsequent inference tasks. On the other hand, since these nodes are not explicitly part of the ontology class definitions, these classes get the private access modifier and are therefore invisible from outside the namespace they belong to.

Semantic Equivalence The semantic expressiveness of the source ontology is preserved with the aid of meta properties. As stated earlier, the role of meta properties is to cover up for the missing explicit semantics in the formal language constructs, i.e, when there is no programming language counterpart for an axiom in the source ontology or when relying on the programming language to express an axioms would interfere with the Open World Assumption OWA. For example, the property association axiom `rdfs:range` could be easily parsed into the data type of the property in the class definition where it belongs to. While this is the norm from a strict modeling perspective, it does not conform to OWA inference mechanism. According to OWA, having two different fillers for the range property is perfectly fine as long as they are not stated to be distinct; whereas this would certainly not pass type checking performed by an OOP language compiler.

It is also worth mentioning that OntoJIT, in its current state, supports OWL $\mathcal{SHOIN}(\mathcal{D})$ DL profile. Parsing ontologies with OWL 2 DL $\mathcal{SROIQ}(\mathcal{D})$ extensions[6], like for example General Concept Inclusion axioms, has not been tested. Table 1. lists OWL DL axioms and their OntoJIT C# counterparts.

4 Demonstrations

To test the parsing process introduced in the previous section, we used the two famous OWL DL Pizza[1] and wine[2] ontologies. These ontologies are relatively small in size but they are pretty expressive as they were created for the purpose of demonstrating the different capabilities of OWL DL and they would therefore be helpful in validating the parsing routine.

Formally proving the semantic equivalence of an OWL DL ontology and the corresponding executable produced by OntoJIT would require at least comparing results of some inference tasks over the two formats which, at this stage of our work, is not possible yet. Instead in this section we demonstrate some code snippets examples of the parsing results and their OWL counterparts.

```
/// <summary>
/// Any Pizza that is not a VegetarianPizza
/// </summary>
public class NonVegetarianPizza : Thing
{
    public NonVegetarianPizza(){}

    public static object equivalentClass
    {
        get{ return "Blank23";}
    }
    public static object disjointWith
    {
        get{return "VegetarianPizza";}
    }
}
```

Fig. 2. Non-vegetarian pizza class definitions (a).

OWL Classes To start with, we consider the example of non-vegetarian pizza definition in the pizza ontology. The produced code snippet is demonstrated in Figures 2. and 4. and the original OWL source is shown in Figure 3. The following is the DL notation of the same information:

$$NonVegetarianPizza \equiv \neg VegetarianPizza \sqcap Pizza$$
$$NonVegetarianPizza \sqcap VegetarianPizza \equiv \bot$$

In the NonVegetarian class definition in Figure 2. we see that the `owl:equivalentClass` term is expressed by mean of the meta property equivalentClass which returns as object (of the RDF triple) a blank node identifier "Blank23". The "Blank23" stands for the anonymous class representing $\neg VegetarianPizza \sqcap Pizza$ that in turn is defined as the intersection of another blank node "Blank22" with the class pizza. Finally "Blank22" is defined as a blank node class with the "ComplementOf" and "VegetarianPizza" meta properties values. As mentioned earlier, the meta properties used in expressing the definitions are essential for substituting for the explicit semantics that are not available as native language constructs. The examples shown here use a textual representation of the values for these properties, in fact these values are just the handles to the created types in the code namespace and are available for later use by the inference component in runtime via reflection.

[1] www.protege.stanford.edu/ontologies/pizza/pizza.owl
[2] www.w3.org/TR/owl-guide/wine.rdf

```
Class: NonVegetarianPizza

    Annotations:
        rdfs:label "PizzaNaoVegetariana"@pt,
        rdfs:comment "Any Pizza that is not a VegetarianPizza"@en

    EquivalentTo:
        Pizza
         and (not (VegetarianPizza))

    DisjointWith:
        VegetarianPizza
```

Fig. 3. Non-vegetarian pizza description in Manchester syntax

```
private class Blank23 : BlankNode
{
    public static object Aggregator
    {
        get
        {
            return OwlVocabulary.intersectionOf;
        }
    }

    public static object CollectionItems
    {
        get
        {
            return "Blank22;Pizza;";
        }
    }
}
```

```
private class Blank22 : BlankNode
{
    public static object Aggregator
    {
        get
        {
            return OwlVocabulary.ComplementOf;
        }
    }

    public static object CollectionItems
    {
        get
        {
            return "VegetarianPizza;";
        }
    }
}
```

Fig. 4. Non-vegetarian pizza class definitions (b).

OWL Properties Just like classes, OWL properties have hierarchical structures. They also have characteristics such as domain, range and cardinality. This is well reflected into OntoJIT executable ontologies. The parser starts with initial hierarchy shown in Figure 1. and expands it as the parsing continues. Parsed classes would then have instances of these properties to express relation between individuals. Figure 5. shows a reduced (incomplete) sketch of the properties hierarchy in the wine ontology. The code snippets in Figure 6. show the definitions for some properties along with their characteristics. The characteristics of a property are supposed to be shared among all its instances and are therefore declared static where as the instance value of the property is a non-static variable.

5 Limitations

5.1 Multiple Inheritance

One of the major differences between modeling in description logic and that in OOP is the different positions the two paradigms have with regards to multiple inheritance. Description logic has a looser interpretation of a class being the subclass of another; indeed, the multiple inheritance term does not really

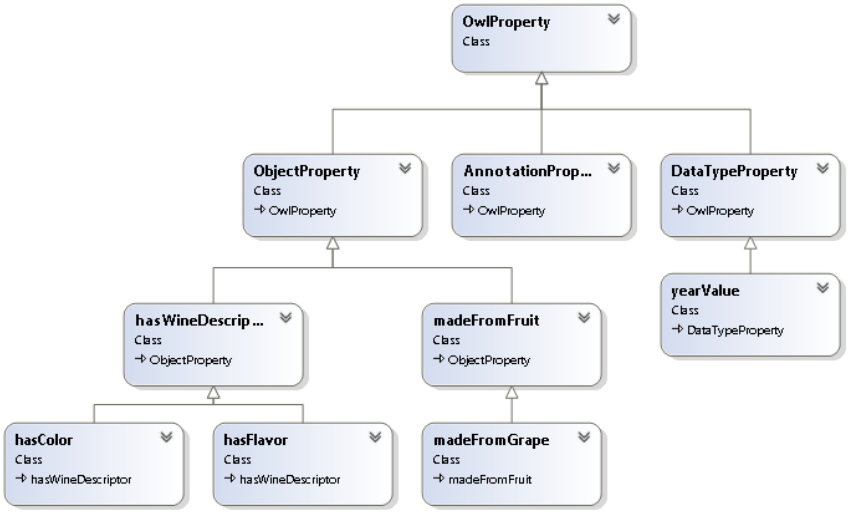

Fig. 5. Reduced sketch of the hierarchy of wine ontology parsed properties

```
public class hasFlavor : hasWineDescriptor
{
    public hasFlavor() {}
    public static object FunctionalProperty
    {
        get { return true; }
    }
    public static object range
    {
        get { return "WineFlavor"; }
    }
}
```

```
public class hasWineDescriptor : ObjectProperty
{
    public hasWineDescriptor(){}
    public static object domain
    {
        get {return "Wine";}
    }
    public static object range
    {
        get{return "WineDescriptor";}
    }
}
```

Fig. 6. OntoJIT property classes for hasFlavor property and its parent Property.

fit in description logic vocabulary. In OWL, the `rdfs:subclassOf` term 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 C# or Java – though not all – have a more strict definition of class inheritance, OOP classes are disjoint by design and that is why a class can not be a subclass of two different parent classes and multiple inheritance is thus not supported. To keep record of all parent classes, OntoJIT parser uses meta properties beside the native class inheritance support in C#, whenever multiple inheritance is encountered, the subClassOf property is extended. This workaround suffers from inconsistency but is still preferable over relying on interfaces where one could use interface declarations instead of classes to reflect OWL hierarchies. The problem with the interface approach is that interfaces are abstract and thus are not instantiable and one would need to create a shadow class for each declared interface. This

can quickly become an overkill and unscalable when considering relatively complex ontologies with a lot of blank nodes. Left with these two not really optimal solutions, the pursuit of a more elegant one is still an open question.

5.2 Import Closure

The approach taken to handle the `owl:imports` terms is a little bit a minimalist approach for one reason; it doesn't handle the case where the ontology being parsed is an OWL DL or OWL light ontology and the imported ontology is an OWL Full one. The parser presented here is mainly concerned with OWL DL or Light profiles and more investigations and analysis are necessary before attempting on parsing an OWL FULL ontology. In this case, the parser is not able to process OWL Full constructs and will therefore skip them. This for sure would have an impact on the soundness of the reasoning results but as reasoning is not yet in the scope of the current state of OntoJIT, this is something to be addressed again as the work in the project advances.

6 Related Work

The difficulty of utilizing OWL ontologies in conventional software projects was behind the work presented in [29]: The authors demonstrate some of the fundamental differences between the "subject-predicate-object" school of modeling (with persistent triple-stores) and the object oriented school (with normalized relational databases). According to the authors, the combined use of ontologies with standard programming practices would enable the development of semantic-rich enterprise applications and they suggest a framework for translating some ontology constructs into Enterprise Java Beans. In [28], the primary intention is to provide guidance on how to build real-world semantic web applications. Here, the authors draw analogy between deploying ontologies as high-level models in software development and the approach used in Model Driven Architecture MDA. They also suggest a software architecture for web services and agents for the semantic web driven by domain ontologies. [26] proposes a hybrid modeling software framework that combines the object oriented representation of a domain with its ontological representation. The authors analyze the advantages and disadvantages of such hybrid modeling approach by means of a case study of a large medical records system. There exist as well many API projects to integrate OWL ontologies into application development. The OpenRDF API [3] along with its satellite projects Elmo/Alibaba[4], provides object triples mapping for creation of flexible RDF-based applications. Another object-oriented API for managing RDF is ActiveRDF [25], it offers schema-free manipulation and querying of RDF data while conforming to RDF(S) semantics. Overall, OWL to UML mapping has a good share of papers in the literature. In [27] A UML-based

[3] OpenRDF, http://www.openrdf.org/
[4] https://bitbucket.org/openrdf/alibaba

visualization of OWL DL ontologies is presented. The work done in [34] provides a rigorous comparison between UML and OWL as two flagship languages for artificial intelligence and software engineering communities; the authors argue that based on the core definitions of ontologies and models, none of the common informal distinctions made between the two terms is actually justifiable. Instead, ontologies themselves are to be regarded as models. Further more, without changes to the currently used ways of distinguishing between models and ontologies the confusion around the two terms will continue to arise.

On the technical side, one particular project that addressed the idea of mapping OWL ontologies into JAVA OOP classes is in[16]. The main aim of the project was aiding semantic application development and the approach taken was to try to stretch the expressiveness of modeling in Java to that of OWL DL by enforcing some constraints and design patterns: Interfaces for multiple inheritance, special listeners on property accessors, type checking for domain and range properties, etc. While we see the motivation behind this approach, we believe that it entails some twisting in the interpretation of OO design principles and what is originally supposed to be explicit semantics in OWL is becoming rather implicit and dependent on the interpretation of the "special purposes" patterns used. Another observation is that this approach would work just fine as long as only the modeling part is concerned but if performing inference tasks is part of the deal, then more caution is necessary. Relying merely on native Java constructs to translate OWL DL means in a certain way delegating the responsibility of enforcing restrictions and properties characteristics to the compiler, which is not exactly the point of properties and restrictions axioms from an open world reasoning perspective. Another related project is in [17]. The authors proposed an initial Python metaclass-based representation of OWL ontologies that offer class declaration and instance creation. Their prototype also allows integrating an OWL DL reasoner with their metaclass representation.

7 Conclusion and Future Work

In this paper we presented a novel approach into democratized semantics by bringing OWL ontologies into the context of programming languages. We also reported on our experience in automatically parsing ontologies into executables. Since the project is in its early stage, there is a lot on the road map for Onto-JIT; mainly exploring the reasoning possibilities over executable ontologies and potential advantages or drawbacks this can bring. One idea here is that with run time dynamic compilation of modern programming languages, the generated source code can change and adapt at run time. In that sense, executing the ontologies would result in spanning the source code as more explicit information are inferred from initial implicit semantics. Another interesting possibility is to exploit hierarchical self-organizing models when inferring class hierarchy using meta properties as features of asserted input class definitions.

Apart from reasoning, there is also more to investigate on the subject of the chosen programming paradigm; the OOP paradigm was a close fit from the

modeling perspective but applying the same idea in an imperative paradigm would also be of interest.

In the long run, we believe that even though the presented idea of democratized semantics is in its infancy stage: the more research we do in this direction the more potentials arise in the two universes of application development and knowledge representations alike.

References

1. Ian Horrocks, Peter F. Patel-Schneider, 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, Volume 1, Issue 1, December 2003, Pages 7-26, ISSN 1570-8268
2. F. Baader, D. McGuinness, D. Nardi, P. Patel-Schneider, The Description Logic Handbook: Theory, implementation and applications, Cambridge University Press, 2002
3. McGuinness, Deborah L., and Frank Van Harmelen. "OWL web ontology language overview." W3C recommendation 10.10 (2004): 2004.
4. OWL 2 Profiles, OWL 2 Web Ontology Language Profiles, W3C Recommendation, 2009.
5. Delugach, Harry. "ISO/IEC WD 24707 Information technologyCommon Logic (CL)A Framework for a Family of Logic-Based Languages." Pacific Northwest National Laboratory, Chantilly, VA 7 (2004).
6. Horrocks, Ian, Oliver Kutz, and Ulrike Sattler. "The Even More Irresistible SROIQ." Kr 6 (2006): 57-67.
7. Lenat, Douglas B., and Ramanathan V. Guha. "The evolution of CycL, the Cyc representation language." ACM SIGART Bulletin 2.3 (1991): 84-87.
8. Genesereth, Michael R., and Richard E. Fikes. "Knowledge interchange format-version 3.0: reference manual." (1992).
9. Brachman, Ronald J., and James G. Schmolze. "An overview of the KL-ONE knowledge representation system." Cognitive science 9.2 (1985): 171-216.
10. Horrocks, Ian. "DAML+OIL: A Description Logic for the Semantic Web." IEEE Data Eng. Bull. 25.1 (2002): 4-9.
11. Niles, Ian, and Adam Pease. "Towards a standard upper ontology." Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001. ACM, 2001.
12. Michael Kifer and Georg Lausen and James Wu, Logical foundations of object-oriented and frame-based languages, JOURNAL OF THE ACM, 1995, volume 42, pages 741–843
13. Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp and James P. Rice, OKBC: A Programmatic Foundation for Knowledge Base Interoperability, AAAI-98 Proceedings.
14. Clark, Peter, Bruce Porter, and Boeing Phantom Works. "KMThe knowledge machine 2.0: Users manual." Department of Computer Science, University of Texas at Austin 2 (2004): 5.
15. Carroll, Jeremy J., et al. "Jena: implementing the semantic web recommendations." Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters. ACM, 2004.

16. Kalyanpur, Aditya, et al. "Automatic Mapping of OWL Ontologies into Java." SEKE. Vol. 4. 2004.
17. Babik, Marian, and Ladislav Hluchy. "Deep integration of python with web ontology language." In Proceedings of the 2nd Workshop on Scripting for the Semantic Web. 2006.
18. Bechhofer, Sean, and Jeremy J. Carroll. "OWL DL: Trees or Triples?." Proc. of the Thirteenth International World Wide Web Conference (WWW 2004). 2004.
19. Bechhofer, Sean, Raphael Volz, and Phillip Lord. "Cooking the Semantic Web with the OWL API." International Semantic Web Conference. Springer Berlin Heidelberg, 2003.
20. Sheard, Tim. "Accomplishments and research challenges in meta-programming." International Workshop on Semantics, Applications, and Implementation of Program Generation. Springer Berlin Heidelberg, 2001.
21. Abramson, Harvey, and M. H. Rogers. Meta-programming in logic programming. MIT Press, 1989.
22. Burmako, Eugene. "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming." Proceedings of the 4th Workshop on Scala. ACM, 2013.
23. Hoyte, Doug. Let Over Lambda. Lulu. com, 2008.
24. Pottier, Franois. "An overview of Cml." Electronic Notes in Theoretical Computer Science 148.2 (2006): 27-52.
25. Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. 2007. ActiveRDF: object-oriented semantic web programming. In Proceedings of the 16th international conference on World Wide Web (WWW '07). ACM, New York, NY, USA, 817-824. DOI=http://dx.doi.org/10.1145/1242572.1242682
26. Puleston, Colin, et al. "Integrating object-oriented and ontological representations: a case study in Java and OWL." International Semantic Web Conference. Springer Berlin Heidelberg, 2008.
27. Brockmans, Sara, et al. "Visual modeling of OWL DL ontologies using UML." International Semantic Web Conference. Springer Berlin Heidelberg, 2004.
28. Knublauch, Holger. "Ontology-driven software development in the context of the semantic web: An example scenario with Protege/OWL." 1st International Workshop on the Model-Driven Semantic Web (MDSW2004). Monterey, California, USA.[WWW document] http://www. knublauch. com/publications/MDSW2004. pdf, 2004.
29. Athanasiadis, Ioannis N., Ferdinando Villa, and Andrea-Emilio Rizzoli. "Ontologies, JavaBeans and Relational Databases for enabling semantic programming." Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International. Vol. 2. IEEE, 2007.
30. Czarnecki, Krzysztof, and Ulrich W. Eisenecker. "Generative programming." Edited by G. Goos, J. Hartmanis, and J. van Leeuwen (2000): 15.
31. Abrahams, David, and Aleksey Gurtovoy. C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond. Pearson Education, 2004.
32. Schult, Wolfgang, and Andreas Polze. "Aspect-oriented programming with c# and. net." Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002). Proceedings. Fifth IEEE International Symposium on. IEEE, 2002.
33. Ganz Jr, Carl. "Runtime Code Compilation." Pro Dynamic. NET 4.0 Applications. Apress, 2010. 59-75.
34. Atkinson, Colin, Matthias Gutheil, and Kilian Kiko. "On the Relationship of Ontologies and Models." WoMM 96 (2006): 47-60.