OntoJIT: Exploiting CLR Compiler Support for Performing Entailment Reasoning over Executable Ontologies

S. Baset and K. Stoffel

***Abstract*—Most recent efforts on bringing ontologies into mainstream programming languages were hindered by some fundamental issues; mainly the lack of expressiveness of programming languages compared to the declarative nature of ontological languages as well as the different assumptions on which reasoning in these languages is based on. In this paper we give the idea of adopting ontological programming approaches a second thought by proposing a prototype for a C# ontological knowledgebase system where ontologies are expressed directly in an executable form. We present our experience on bridging the semantic gab in general purpose programming languages and on exploiting metaprogramming and the dynamic compilation feature of modern compilers for performing certain entailment queries without the need for the bulky ontology classification step usually required in the case of conventional ontological tools.**

***Index Terms*—Dynamic compilation, executable ontologies, metaprogramming, OntoJIT, OWL, semantic programming.**

I. Introduction

The considerable success of Model Driven Development (MDD) [1] inspired experts working in the neighboring domain of Knowledge Representation (KR) to introduce ontologies into the landscape of software application development as formal domain models. Their efforts yielded the term Ontology Driven Software Development (ODSD) [2], a methodology that is ignited by the semi or fully automatic translation of concepts of an ontology, usually expressed in a specific language such as the Web Ontology Language (OWL), into a general-purpose programming language that can be used throughout the rest of the development process.

ODSD was a step forward in the direction of bringing knowledge representation techniques to the conventional object-oriented modeling and software engineering communities but it only offered little utility of the imported ontologies due to: 1) The semantic gap between the declarative and more expressive nature of ontological languages and the restricted formal general-purpose programming languages. 2) The fact that ontological KB systems are based on a different, and to some extent opposing, Open World Assumption (OWA) compared to the Closed World Assumption (CWA) on which most database and information systems are built.

These two problems need to be addressed for the idea of integrating ontologies into mainstream programming languages to bring its potentials. I.e. beside the rapid application development aspects of the integration, it should be possible to perform logical inferences to entail implicit knowledge from the explicitly stated facts in the integrated ontologies. Hence our motivation is to deploy programming languages as a new means of expressing ontologies all while maintaining their semantic profile. Our interest lies particularly in exploiting language features such as reflection, lambda expressions and dynamic compilation for performing inference queries about the imported ontologies in their executable form.

In this paper we present a prototype for an ontological KB system where ontologies are expressed directly in an executable form (C# code statements) that serves as a programming interface for accessing and using ontologies within an object-oriented programming language. We show through some examples how expressing ontologies as code constructs, with the help of native programming language support, allows for performing certain entailment queries without the need for the bulky ontology classification step usually required in the case of conventional ontological tools such as Protégé or OWL API.

II. Background

As the work proposed in this paper lies at the crossroads between knowledge representation and software languages, we will try in this section to briefly cover from the two paradigms the basic concepts that are essential to the understanding of the rest of the paper.

A. Ontologies

"Ontologies", in plural form, is an engineering term derived from the ancient philosophical Greek term Ontology. Ontologies in the context of knowledge representation are formal abstract models used by computers systems to describe and share knowledge about the real world. This is achieved by explicitly defining the concepts relevant to the domain, as well as the relationships between these concepts, using a formal computer language to avoid ambiguity or incomplete specifications.

These key aspects of ontologies are put together in a concise definition by Gruber *et al* in [3]: "An ontology is a formal and explicit specification of a shared conceptualization".

What differentiates ontological modeling from other modeling paradigms such as UML or Entity Relationship modeling is that ontologies are intended a priori to be shared and they are therefore application-independent to a certain

Manuscript received February 5, 2018; revised May 10, 2018.

S. Baset and K. Stoffel are with the Information Management Institute of University of Neuchâtel, A.L.Breguet 2, CH-2000 Neuchâtel, Suisse (e-mail: sohaila.baset@unine.ch, kilian.stoffel@unine.ch).

doi: 10.18178/ijke.2018.4.1.093

extent. This calls for a standard ontological language to be used when building and sharing ontologies and in response to this call, many ontology languages were proposed during the last two decades [4], [5] Nowadays, ontology modeling is largely dominated by the web ontology Language OWL, the W3C standard language for the semantic web. OWL has two versions OWL and OWL 2 and both versions has got many sub-languages that are varying in expressiveness at an increasing complexity overhead [6], [7] 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 (DL) [8] that offers a good balance between expressiveness and decidability for most KR applications.

Ontologies defined in OWL consists of classes, properties and individuals (instances of classes) all of which are designated by axioms of class, data range, datatype and object property expressions [6].

B. Metaprogramming

Metaprogramming refers to the programming paradigms and the means by which a program has knowledge of itself or can manipulate itself. To that end, a metaprogram is a program that writes, analyses or transforms programs including itself. This self-modifying code feature of metaprograms allows for significant flexibility in handling runtime code changes efficiently without recompilation. It can also help reducing the development time by minimizing the number of lines of code needed to express a solution. In order to support such features, programs in metaprogramming are treated as data; they are usually called object-programs where the term object-program simply denotes a sentence in a formal language. By manipulating object-programs, i.e. constructing, combining or fragmenting them, the metaprogram can evolve. We call the language in which the metaprogram is written the metalanguage and the language of the programs that are manipulated the object language. The ability of a programming language to be its own metalanguage is called reflection [9].

Metaprogramming is an approach that is not equally supported by all programming languages. Some languages, such as CaML [10], are designed with metaprogramming in the core of their philosophy. Dynamic languages like Prolog and smalltalk have fundamental metaprogramming features [11]. Macros in Lisp and Scala also provide strong support for metaprogramming [12], [13], whereas Python programmers usually use meta classes. When it comes to strongly typed languages, however, the emphasis on such features becomes less evident. This does not mean that metaprogramming is not supported in many of these languages; C++ offers templates for metaprogramming [14], Java programs have annotations and .Net languages use annotations [15] and/or reflection to produce meta programs [16], [17]. The proposed prototype in this paper relies heavily on the reflection feature of C# in addition to the .Net libraries for dynamic code compilation.

C. 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). 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. Dynamic compilation aims at getting the advantages of both approaches, i.e. the speed of compiled code with the flexibility of interpretation. A dynamic complier 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 AOT compilers and can therefore enjoy the flexibility of interpretation while maintaining the performance optimization of compilation.

In the context of the work presented in this paper, the ability to compile code at runtime is an essential prerequisite to the parsing component of the proposed prototype. It represents the mechanism by which the asserted axioms from OWL source files are translated into C# code statements and further integrated into the executable of the runtime environment.

III. The Proposed KB Prototype

The alliance between the logic-based approaches of knowledge representation and the powerful techniques of modern programming languages offers some new possibilities for performing entailment reasoning at no price. If ontologies were readily available for the developers as code objects in their programming environment, then exploiting language features such as reflection and lambda expressions allows for more control over entailment query design and execution as opposed to the traditional protocol of loading the ontology into data objects and globally classifying it via a DL reasoner before performing any entailment query.

To validate the afford mentioned ideas, we present OntoJIT Fig. 1.; a working prototype for an ontological KB system where ontologies are directly expressed in an executable form (code statements) that serves as a programming interface for accessing and using an ontology within an object-oriented language. Entailment tasks such as partial query classification and query answering are powered by the built-in programming language support without the need for the bulky ontology classification as a prior step usually required in the case of conventional ontological tools such as Protégé or OWL API.

The proposed prototype and the resulting executable ontologies are entirely written in C#. Ontology transformation into runtime executable is realized using the Common Language Runtime (CLR) compiler of the .Net environment (the Just-in-Time Compiler JIT, hence the name OntoJIT) whereas reasoning tasks rely on the built-in object-oriented inheritance and the metaprogramming techniques of the C# language.

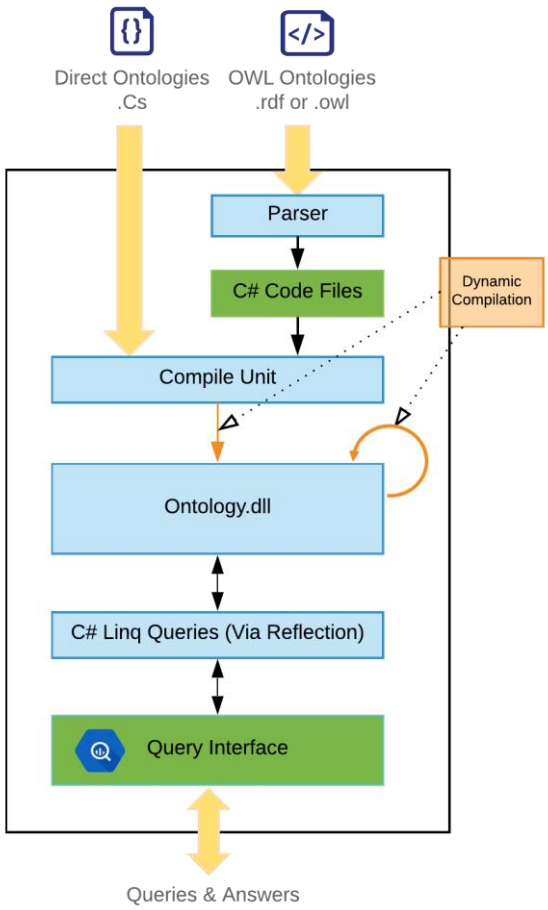

Fig. 1. OntoJIT architecture.

A. *Owl Ontologies or Direct Ontologies*

OntoJIT offers two possibilities for expressing ontologies as executables. First one is to translate existing OWL ontologies via a parsing component that takes as input ontologies in RDF/XML or OWL/XML format. It produces then the corresponding C# code namespace which will be dynamically compiled at runtime as part of a compile unit before being accessible as a .dll or .exe executable. Ontology translating option serves a double objective in our work; first of all, it allows us to bootstrap the KB development process by reusing the readily available ontologies on the web. Also, and more importantly, through this translation step we can validate the hypothesis that in spite of the expressiveness gap between OWL and formal programming languages, it is still possible to maintain the semantic profile of the source ontology in its new executable form. This task is quite difficult because the declarative nature of OWL compared to the less expressive target programming language and the fundamental differences between ontological and object-oriented schools of modeling impose many challenges on the automatic translation process. The OntoJIT parsing component adopts a simple yet effective approach to bridge the problematic semantic gap mainly by relaying on a meta-properties code layer to cover up for the missing explicit semantics in C#. Table I. provides more details on the mapping between OWL axioms and their C# counterpart constructs. OntoJIT also supports blank RDF nodes usually present in OWL to anonymously represent a property restriction or class description axioms without explicitly naming a concept. Though in our implementation, blank nodes are not anonymous; they are created as class definitions with 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, they get the private access modifier and are therefore invisible from outside the namespace they belong to. 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.

```
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();
        subClassOf = MozzarellaTopping;
        hasTopping.someValuesFrom =
            new List<PizzaTopping>()
                { MozzarellaTopping,
                  PeperoniSausageTopping,
                  TomatoTopping
                };
        hasTopping.allValuesFrom = MozzarellaTopping;
        hasCountryOfOrigin.hasValue = America;
        hasTopping = new hasTopping();
    }
}
```

Fig. 2. (a) A code snippet produced by the parser.

```
public class GO_0048311 : GO_0007005
{
    public static string label;
    public static object subClassOf;

    static GO_0048311()
    {
        subClassOf = GO_0051646;
        label = "mitochondrion distribution";
    }
}

public class GO_0000002 : GO_0007005
{
    public static string label;

    static GO_0000002()
    {
        label = "mitochondrial genome maintenance";
    }
}

public class GO_0007005 : GO_0006996
{
    public static string label;

    static GO_0007005()
    {
        label = "mitochondrion organization";
    }
}
```

Fig. 2. (b) A code snippet produced by the parser.

Fig. 2. (a) and (b) show two code snippets of the intermediate C# code produced by the OntoJIT parser for some ontologies that we will introduce in the next section. As

listed in Table I, additional meta-properties are used to bridge the semantic expressiveness gap. In the case of multiple inheritance, for example, the `subClassOf` property is used in addition to the built-in single inheritance supported in C#. The meta-properties corresponding to terminological axioms are static (i.e. shared among all instances) whereas meta properties describing individuals (`owl:AllDifferent`, `owl:differentFrom` and `owl:sameAs`) are non-static. All meta-properties are first defined in the top-level class for both OWL concepts and OWL properties and then inherited, and masked where necessary, by sub classes in the hierarchy.

TABLE I: OWL DL AXIOMS AND THEIR C# COUNTERPARTS IN ONTOJIT.

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	

As an alternative to translating existing OWL ontologies, the experience we obtained writing the parsing component enabled us to establish some grounding for directly expressing ontologies as C# code. As a matter of fact, directly expressing code ontologies boils down to inheriting a certain class hierarchy and implementing the right interfaces. Both OWL and direct C# ontologies will end up in the compile unit to be compiled at runtime. The resulting executable ontology is then available for subsequent queries via the C# built-in Language Integrated Query (Linq) [18].

B. *Reasoning over Executable Ontologies*

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 is orthogonal to the different representations formats of OWL concepts. In other words, what is more interesting in reasoning over executables ontologies is not to perform reasoning using the present logical entailment algorithms but to explore what new possibilities the new executable representation can bring.

For that purpose, we can benefit from the new palette of metaprogramming and dynamic compilation tools offered by the language compiler. For example, we can rely on C# reflection to access into type information of OWL concepts now represented as C# classes. This is an out-of-the-box feature that allows us to retrieve the transitive closure of all sub classes of a given concept (or its ancestors).

C# Linq queries make it no longer necessary to use a query editor or write queries in a separate language such as SPARQL [19]. Instead, developers can directly write their questions as native C# queries against the current runtime assembly containing the ontological KB facts. Complex entailment queries are also possible to formulate using lambda expressions as predicates in the body of the query.

Another important benefit that comes with the new executable representation is the built-in support for object-oriented inheritance that can be exploited to add a procedural extension to the imported ontology. To take a concrete example, let's consider the small interface and the class definition shown in Fig. 3. The class `Thing`, which is the corresponding C# class to OWL top concept, is set to implement the `IClassifiable` interface.

```
public interface IClassifiable
{
    public static void Classify();
}

public class Thing : IClassifiable
{
    public static void Classify()
    {
        // Ontology classification logic
        // goes here....
    }

    // Relevant class meta-properties
    // ....

}
```

Fig. 3. Partial classification via class inheritance.

All translated OWL concepts are classes that inherit either directly or indirectly from the class `Thing`. This means they all implement the `IClassifiable` interface and have, therefore, their implementation of the method `Classify()`. Now it is possibly to call the classify method on a concept class in any level of the type tree hierarchy and to recursively classify all classes below the selected one. This provides the developer with significantly greater flexibility when working with ontologies with large terminological boxes because it is no longer necessary to always perform global classification on the whole ontology but rather on the ontology's "subtree" of interest for the given task.

Finally, the execution of the `Classify()` method may result in entailing new implicit semantics and thus in modifying the code to represent the newly available information. The resulting code modification can be materialized and reflected into the runtime executable by means of dynamic compilation.

C. *Query Interface*

In cases where abstracting the technical details of the lambda expressions and Linq queries is desired, an optional encapsulating query interface layer completes the prototype. This layer makes it possible for users to specify query terms without having to deal with the corresponding Linq expressions.

Some example queries are provided in the following demonstration section.

IV. Demonstration

For demonstrating some of the new possible forms of entailment queries using executable ontologies we chose two well-known ontologies in the domain of knowledge engineering. The first one is the gene ontology (GO)[1], a large ontology that has around 43585 terms and 93265 relations with respect to three aspects: molecular functions, cellular components and biological processes [20].

The second ontology is the Stanford Pizza Ontology[2]; a rather small ontology but very useful in validating logical entailment queries since it was developed as a tutorial for demonstrating the different OWL DL constructs.

After successfully parsing the ontologies into runtime executables, we designed a test to run a set of entailment queries against the executable ontology and to automatically compare the query results with the results obtained by Protégé using both HermiT and FaCT++ reasoners [8]. Most tests results were identical except for some cases involving OWA reasoning that we will discuss in the following section.

A. Some Example Queries

In this section we present some of the queries we used to test entailment reasoning potentials over executable ontologies; we limit the scope in this paper for testing terminological entailment about concepts in an ontology and we don't include queries about assertional axioms. Before presenting the queries and discussing the obtained results, it is worth emphasizing that OntoJIT queries are entirely based on the integrated query mechanism of C# and do not require a separate reasoning step usually inevitable -and extremely heavy in large ontologies like the gene ontology- in any OWL DL entailment query.

To start simple, we will first consider the DL query to find all subtypes of the concept chromosome. In this case, all we have to do is to consider the transitive closure over two relations; the class inheritance (which is readily available via the type information in the programming language) as well as the equivalent class meta property. The results are shown in Fig. 4.

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

Fig. 4. Query results for the term 'chromosome'.

A slightly more complicated query to answer is to find all chromosomes that are part of a cytoplasm. This translates into the conjunctive DL query:

$$\exists x.(chromosome(x) \wedge is\_part(x,y) \wedge cytoplasm(y))$$

The necessary axioms to answer this query from the source gene ontology are translated into C# type information and are accessible via reflection. The C# Linq query we used is shown in Fig. 5. (a) and the results are listed in Fig. 5. (b).

```
var chromosomes = TransitiveClosure(typeof(chromosome));
var cytoplams = TransitiveClosure(typeof(cytoplasm));

var types =
    from type
    in Assembly.GetExecutingAssembly().GetTypes()
    let property = type.GetField("is_part")
    let value = (ObjectProperty)property.GetValue(null)
    where chromosomes.Contains(type) &&
    cytoplams.Intersect(UnwrapValue(value.someValuesFrom)).Count() > 0
    select type;
```

Fig. 5. (a) C# Linq query for chromosomes that are part of a cytoplasm.

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

Fig. 5. (b) Linq query results

For the pizza ontology we could adopt a more systematic approach for query results' validation thanks to its relatively small number of terminological axioms. As a fixed reference model, we used the query of the Pizza Finder[3] application to suggest pizzas that have some desired toppings but none of the specified excluded toppings. We automated query term generation based on a large random subset of possible combinations of included and excluded toppings. We then ran the same queries in OntoJIT as well as in the Pizza Finder Java application.

Query evaluation in Java application relies on creating a temporary concept of conjunctive terms and negated terms and finding all concepts in the ontology that are subsumed by the just created concept. This requires a prior ontology classification step to be performed by the DL reasoner. In OntoJIT, on the other hand, we formulated the query as the relative complement **A\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. The obtained query results from the two approaches were always matching except for the few cases where the open world assumption used in DL entailment forbids entailing the truth value of the corresponding query term. On the other hand, the closed world assumption adopted in OntoJIT allows query evaluation to be more relaxed and to return more results in the result set. 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 in OWL API and OntoJIT are: {"American","FourSeason"} and

[1] The Gene Ontology Consortium: http://www.geneontology.org/
[2] www.protege.stanford.edu/ontologies/pizza/pizza.owl
[3] https://github.com/owlcs/pizzafinder

{“ American”,”Capricciosa”,”FourSeason”,”LaReine”,”Pare mense”,”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.

V. Discussions

While the idea of having all the capabilities of ontological knowledgebase systems at the doorstep of our preferred programming language environment is alluring, the transition between the two different schools is still no free lunch. In the following sections, we shed some light into some of the challenges we encountered when implementing OntoJIT.

A. The Semantic Gap

The semantic richness of ontological languages makes it very difficult to find a programming language counterpart to express all possible OWL axioms. As an example we can consider finding a native programming counterpart for OWL DL terms such as: `owl:disjointWith` used to indicate that a class is disjoint with another class and `owl:equivalentClass` used to indicate an equivalent class. In OOP, all classes are disjoint by default so there is no built-in mechanism to selectively group disjoint ones. Same goes for indicating an equivalent class; in plain OOP there is no point in defining another class if there exists an equivalent one already. The approach we took to overcome these limitations is simply to rely on a meta-properties layer to compensate for the missing semantics [21]. In the literature [22], [23], there exist some other interesting attempts trying 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.

B. Open World Assumption Reasoning

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. The OWA argues that since it is not possible for an agent to have complete knowledge then it is not possible, via deductive reasoning, to infer the truth value of a fact not present in its base (explicitly or implicitly) irrespective of whether or not it is known to be true.

While OWA makes sense in the context of description logics and other pure logic programming languages, it is still rather counter-intuitive and a major source of confusion for most conventional software developers in contrast to the closed world assumption in other common modeling and data paradigms. Furthermore, it has been argued that closed world assumption and local closed world assumption LCWA [24] are largely sufficient in many application domains.

VI. Related Work

Scanning literature in both areas of knowledge representation and software engineering for work related to executable ontologies reveals a relatively scattered body of research. The majority of related papers are concerned with the modeling aspects of OWL and how it could be used to enrich application semantics without referring to reasoning possibilities.

The difficulty of utilizing OWL ontologies in conventional software projects was behind the work presented in [25]: The authors demonstrate some of the fundamental differences between the "subject-predicate-object" school of modeling and the object-oriented school. 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 [2], the primary intention is to provide guidance on how to build real-world semantic web applications. 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.

The authors of [26] proposed a hybrid modeling software framework that combines the object-oriented representation of a domain with its ontological representation after analyzing the advantages and disadvantages of such hybrid modeling approach.

OWL to UML mapping has also a got good share in the literature: [27] presents a UML-based visualization of OWL DL ontologies while the work done in [28] 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. Furthermore, without changes to the currently used ways of distinguishing between models and ontologies the confusion around the two terms will continue to arise.

Finally, the OpenRDF API, 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 [29], it offers schema-free manipulation and querying of RDF data while conforming to RDF(S) semantics.

[4] OpenRDF project http://www.openrdf.org/ and Elmo/Alibaba https://bitbucket.org/openrdf/alibaba

VII. Conclusion and Future Work

In this paper, we reported our experience on using main stream programming languages to represent ontologies and to perform some entailment reasoning query over the executables. We proposed a prototype for an ontological knowledgebase system where ontologies are directly represented via C# classes and instances. We also demonstrated through examples some of the new possibilities to exploit metaprogramming features to answer certain entailment queries out of the box and eliminating the need for the bulky pre-step of ontology classification.

We are also working on analyzing partial ontology classification algorithms by spanning the tree of type information using the language built-in support for inheritance.

For the future work, we are interested in extending the set of potential entailment tasks to support assertional queries as well as more terminological queries besides the ones demonstrated in this paper.

References

[1] C. Atkinson and T. Kuhne, "Model-driven development: A metamodeling foundation," *IEEE Software*, vol. 20, pp. 36-41, 2003.

[2] H. Knublauch, "Ontology-driven software development in the context of the semantic web: An example scenario with Protege/OWL," in *Proc.* chez *1st International Workshop on the Model-Driven Semantic Web (MDSW2004)*, 2004.

[3] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, pp. 199-220, 1993.

[4] I. Horrocks, "DAML+OIL: A description logic for the semantic web," *IEEE Data Eng. Bull.*, vol. 25, pp. 4-9, 2002.

[5] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object-oriented and frame-based languages," *Journal of the ACM (JACM)*, vol. 42, pp. 741-843, 1995.

[6] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz, "Owl 2 web ontology language: Profiles," *W3C Recommendation*, vol. 27, p. 61, 2009.

[7] B. Motik, P. F. Patel-Schneider, and B. C. Grau, "Owl 2 web ontology language direct semantics," *W3C Recommendation*, vol. 27, 2009.

[8] F. Baader, *The Description Logic Handbook: Theory, Iplementation and Applications*, Cambridge University Press, 2003.

[9] P. Maes, "Concepts and experiments in computational reflection," *chez ACM Sigplan Notices*, 1987.

[10] F. Pottier, "An overview of Cml," *Workshop on ML*, 2005.

[11] H. Abramson and M. H. Rogers, *Meta-programming in Logic Programming*, MIT Press, 1989.

[12] E. Burmako, "Scala macros: Let our powers combine! On how rich syntax and static types work with metaprogramming," in *Proc. the 4th Workshop on Scala*, 2013.

[13] D. Hoyte, *Let Over Lambda*, Lulu. com, 2008.

[14] D. Abrahams and A. Gurtovoy, "C++ template metaprogramming: Concepts, tools, and techniques from Boost and beyond," *Pearson Education*, 2004.

[15] K. Czarnecki and T. U. W. Eisenecker, "Generative programming," in *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, p. 15, 2000.

[16] W. Schult and A. Polze, "Aspect-oriented programming with c# and. net," in *Proc. Fifth IEEE International Symposium on* chez *Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002)*, 2002.

[17] C. Ganz Jr, "Runtime Code Compilation, "chez *Pro Dynamic. NET 4.0 Applications*, pp. 59-75.

[18] P. Pialorsi and M. Russo, *Introducing Microsoft® linq*, Microsoft Press, 2007.

[19] S. Harris, A. Seaborne, and E. Prud'hommeaux, "SPARQL 1.1 query language," *W3C Recommendation*, vol. 21, 2013.

[20] G. O. Consortium, "Expansion of the Gene Ontology knowledgebase and resources," *Nucleic Acids Research*, vol. 45, n °%1D1, pp. D331--D338, 2016.

[21] S. Baset and K. Stoffel, "OntoJIT: Parsing Native OWL DL into Executable Ontologies in an Object Oriented Paradigm," chez *International Experiences and Directions Workshop on OWL*, 2016.

[22] A. Kalyanpur, D. J. Pastor, S. Battle, and J. A. Padget, "Automatic Mapping of OWL Ontologies into Java.," chez *SEKE*, 2004.

[23] M. Babik and L. Hluchy, "Deep integration of python with web ontology language," in *Proc. the 2nd* Workshop *on Scripting for the Semantic Web*, 2006.

[24] P. Doherty, W. Lukaszewicz, and A. Szalas, "Efficient Reasoning Using the Local Closed-World Assumption," in *Proc. of the 9th International Conference on Artificial Intelligence: Methodology, Systems, and Applications, AIMSA 2000 Varna, Bulgaria, September 20--23, 2000*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 49-58.

[25] I. N. Athanasiadis, F. Villa, and A.-E. Rizzoli, "Ontologies, JavaBeans and Relational Databases for enabling semantic programming," in *Proc. 31st Annual International Conference on Computer Software and Applications, 2007. COMPSAC 2007.*

[26] C. Puleston, B. Parsia, J. Cunningham, and A. Rector, "Integrating object-oriented and ontological representations: A case study in Java and OWL," *International Semantic Web Conference*, 2008.

[27] S. Brockmans, R. Volz, A. Eberhart, and P. L{\"o}ffler, "Visual modeling of OWL DL ontologies using UML, " *International Semantic Web Conference*, 2004.

[28] C. Atkinson, M. Gutheil, and K. Kiko, "On the Relationship of Ontologies and Models.," *WoMM*, vol. 96, pp. 47-60, 2006.

[29] E. Oren, R. Delbru, S. Gerke, A. Haller, and S. Decker, "ActiveRDF: Object-oriented semantic web programming," in *Proc. the 16th International Conference on World Wide Web*, 2007.

S. Baset is a Ph.D candidate at the information management institute in the university of Neuchâtel, Switzerland. Before joining the institute in 2015, she had a career as a software engineer working for several innovative and technology-oriented companies including her last position at Frontiers, an open access publishing house based in the scientific park of EPFL, Switzerland.

She completed her 5-years bachelor's degree in computer science from Albaath university in 2009 with a specialization in software engineering. In 2013, she obtained a Certificate of Advance Studies in information systems from the Swiss Federal Institute of Technology - ETH Zurich. Between 2013 and 2015 she pursued a master of science degree in management of information systems from the university of Lausanne where she worked on the subject of resolving close-domain entity ambiguity using hybrid similarity measures.

Her current research interests are mostly focused on enforcing the synergies between modeling in knowledge representation and conventional software engineering by exploiting metaprogramming and dynamic compilation to automatically translate and incorporate ontologies into the software development lifecycle.

K. Soffel is a senior researcher in the domain of data mining and knowledge discovery. He has been rector of university of Neuchâtel, Switzerland since August 2016. Prior to this position, he was a full professor at the faculty of business and economics for the years between 1997 and 2016.

He received his bachelor of science degree from the university of Fribourg, Switzerland in 1989 with a double-major in mathematics and computer science. He pursued his Ph.D in computer science from the same university and graduated in 1994. Later this year, he moved to the United States to join College Park at the University of Maryland as a research associate and fellow at the Johns Hopkins hospital where he contributed to the realization of PARKA-DB, a scalable knowledgebase system for very large ontologies.

K. Soffel has many scientific contributions ranging from theoretical aspects to design and implementation of data mining and knowledgebase systems. He has collaborated actively with researchers in several other disciplines of artificial intelligence, namely on fuzzy logic and methods for decision making process.